Technical University of Crete

SCENEWIZARD: A UNITY API FOR EMBODIED INTERACTIVE PROJECTION MAPPING FOR REAL-TIME PERFORMANCES

Author: Artemis Georgakopoulou
(ageorgakopoulou@isc.tuc.gr)

Dissertation Thesis

Committee:

Supervisor: K. Mania, Associate Professor,
K. Ouggrinis, Associate Professor,
M. Zervakis, Professor

April 10, 2019

Acknowledgements

I would like to express my deep gratitude to Professor Katerina Mania, my thesis supervisor, for giving me the chance to explore an amazing field in Computer Science and manifesting my idea into reality. I would also like to thank Professor K.Ouggrinis, for his patience, guidance and enthusiastic encouragement during my research in SceneWizard. My grateful thanks are also extended to M. Christoulakis for his advice and assistance in keeping my progress on schedule, as well as his useful critiques of this research work. I would also like to thank Professor M. Zervakis for accepting to be in my committee and approved the idea of SceneWizard.

My grateful thanks are also extended to my laboratory colleagues for their help in conducting the gesture database evaluation, as well as their valuable assistance, as means of exchanging information and ideas. Finally, I wish to thank my friends, my partner and family, especially my parents, standing by my side and supporting me throughout all the years of my study.

Abstract

In this thesis we present an Interface in Unity for Interactive Projection Mapping using Kinect Depth Sensor. The main idea is to create a link between Unity3D, Kinect and a Video Mapping Software(Projection Mapping Software), so that Projection Mapping artists and designers can include interactions with projected computer-generated visual effects within each Projection Mapping project they want to perform. The project could work also as a Software Development Kit for Unity, which provides tools that assist in developing Kinect applications and Interactive Projection Mapping installations.

In addition to that, we provide a database of gestures that are pre-trained and ready-to-use, as well as tools that ease the Projection Mapping set-up process. This way designers and artists not only have the possibility to perform Interactive Projection Mapping, but also expand the impact of their performance, in an immersive, recreating and entertaining experience.

SceneWizard is an Assistance Library for Unity, which purpose is to facilitate the process of creating and presenting Interactive Projection Mapping performances. Using SceneWizard, the user can easily recreate the real scene into Unity's virtual environment, simply by creating and aligning the projection's components (surfaces and projector) according to the geometric relationship between them. The user also has the possibility to create various versions of a specific scene and present a variety of visuals, as means of storytelling performance. Additionally, the user can automatically associate the surfaces with pre-composed visual effects, created and rendered in the VideoMapping Software of his/her choice. Furthermore, the user has the possibility to correlate each surface he/she desires with a corresponding gesture, provided from the custom Gesture Database we designed for our tool. Finally, using a Kinect Depth Sensor, the user can perfom an Interactive Projection Mapping Show, with SceneWizard's capabilities of: iterating through the created scenes, resetting the initial state of each scene's surfaces, so he/she can perform the gestures of a scene more than once and saving the performance settings as stand-alone asset files (transferable data in any Unity project).

SceneWizard was developed in Unity 2018.1.4 running in Windows 10 Operation System. For the implementation we needed to install Kinect MS SDK for Windows as well as import the Kinect unitypackage in Unity so that the platform can collect data from the sensor. Additionally, we needed to install Spout Protocol and its unity package to initialize the communication between Unity and the Video Mapping Software, to achieve ambiguous sharing of video data. The project was developed on an MSI GP62M series.

Contents

List of Figures

Listings

List of Tables

1 Introduction

Projection Mapping, simply put, is to use technology to project imagery (video, animation and other colorful displays) onto a three-dimensional surface within venues, such as sport stadiums, nightclubs and concert halls. The projections can be as simple as indoor stage effects or as complex as video onto buildings and industrial landscapes. Instead of using a boring flat white screen, it can be used to bring objects into life, create immersive environments and provide exciting experiences through visual stimulation. Projection Mapping can be used for advertising, live concerts, theater, gaming, computing, decoration and anything else you can think of. The idea is to use technology, like video projectors, to manipulate lightning onto varying surface types and turn common objects into 3D displays. In essence, it's like painting with light- a way to add textures, colors and even feelings to an environment.

While many augmented reality approaches involve rendering graphics over a live video (video see-through), handheld, or head-worn (optical see-through), Projection Mapping integrates the augmenting graphics on the real environment surfaces, so it does not divert the user from the real world. Therefore, Projection Mapping provides the users with augmented reality experiences without needing to wear bulky equipment that can hinder face to face personal interactions. In addition to that, even though traditional augmented reality techniques, such as, video see- through, could provide similar results, they currently support a limited field of view. Contrariwise, Projection Mapping provides the possibility of a wide field of view, giving the chance for better quality images, multiple users attendance, or even interaction.

Although there are various types of powerful software widely available, in order to achieve Projection Mapping, adding interactivity with the end-viewer quite complicates the process. Such software, already require a lot of work along with technical and artistic skills, as the user (artist or designer) needs to become proficient with it. Simultaneously, making Projection Mapping interactive requires high level of programming skills and patience. Interactive Projection Mapping installations could include, gestures or movement interactions, specific object, or marker-based interactions, interactions with custom-made led controllers and many more. These kind of interactions include reading and manipulating data from any kind of sensor, such as Kinect Depth Sensor, or even a simple RGB camera. To use sensors like these, you need to create custom programs, in order to collect and manipulate the appropriate data to achieve interaction.

Over the past few years, academic study has been conducted focusing on controllers and motion-based video games, in order to add interactivity. For many years, the motion sensing input device technology was something that people only saw either in movies or on their television sets. However, technological advancements in the gaming industry have enabled motion sensing input devices to become reality. In November 2010, the Kinect Motion Sensor was introduced to the market as an input device for XBOX360 gaming console and was very successful product with more than 10 million devices sold by March 2011. In this thesis, we will thoroughly discuss implementing gesture recognition via Kinect Motion Sensor, in order to achieve interactive Projection Mapping spatially.

1.1 Brief Description of our Approach

SceneWizard is a UnityPackage, an Assistance Library in Unity, which provides developers and designers with tools to create Interactive Projection Mapping projects, installations and performances. The main focus of the project is to create an interface of the connection between Unity and Kinect and between Unity and any VideoMapping software (ex. AfterEffects, ResolumeArena5, etc.). Unity3D will be the main tool of this project, to simulate the scene and the events.

Through SceneWizard's User Interface, the user can control the two necessary Software for the Performance: Unity and a VideoMapping Software of his/her choice, as well as the two necessary hardware for Interactive Projection Mapping: the Projector and a Kinect Depth Sensor. The Projector is represented as a Camera in Unity's virtual scene and needs to be parameterized according to the real Projector's characteristics (FoV, throw-ratio). Additionally, the user can set up the virtual Scene using SceneWizard's UI, in correspondence to the geometric relationship between the Projector and the Scene's Surfaces (distance, projection angle). Furthermore, the user can rectify the inevitable image registration inaccuracies of the Surfaces, by manually aligning the virtual Surface's edges to the real geometry Surface, during PlayMode and before the Performance. Besides this, the real Scene, can have multiple versions of itself, since the user has the ability, to create and customize one Scene with various and possibly different characteristics (Visuals, or Gestures) for each of the containing Surfaces.

Moreover, the user can customize the Scene's Surfaces, by selecting a Gesture, provided by our custom Gesture Database, that will correspond to a specific Surface and will control the Visuals displayed on that Surface. More specifically, the user can select an initial Visual that will be displayed on the Surface to begin with, and a next Visual that will be displayed after the Gesture's occurrence. In continuous, when the selected Gesture has been performed (during Run-time) the corresponding Surface will switch to the selected next Visual, set from SceneWizard. During the Run-time Performance, the user can loop through the Scene Versions, in order to "tell" a story without words, and also can reset the Surfaces of a current rendered Scene to their initial state, in order to perform the associated Gestures again.

1.1.1 Gesture Recognition

Initially, this project aims at recognizing gesture events from the "designer" or user and list them, so that the "designer" can use them for his project to trigger events. The "gesture" here, means position or motion of the body joints. To succeed this function, several packages need to be imported in Unity3D. To acquire the skeleton data, we can use OpenNI open source library, or modify the scripts of MS Kinect SDK. To recognize the gestures, we have created a list of custom gestures, a custom Gesture Database, which will be detected from the movement of skeletal joints. To succeed this, we trained the Gestures to a Kinect Depth Sensor, using the Visual Gesture Builder Kinect feature. The latter, can be utilized to train the Depth Sensor into recognizing specific Gestures, by training Machine Learning Algorithms, which classify the gesture data and can be trained to recognize specific patterns. Moreover on this, we will use an API provided from Github, which allows us to communicate with Kinect Depth Sensor via C# Scripts.

1.1.2 Textures via VideoMapping software

To create mesmerizing results of manipulating visual effects with the body, the visual effects need to be imported in Unity3D, so that they later can be triggered. To succeed this, we concluded on using Spout Protocol, a video frame sharing system for Microsoft Windows, which allows applications for Microsoft Windows to share OpenGL textures. Spout Protocol SDK will be the connection between Unity3D and any VidepMapping software of our choice. Basically, it will import the visual effects from the VideoMapping

Software to Unity3D as textures. In this way, the "designer" will be able to trigger these texture events, as a result of a gesture. An editor window will guide the "designer" into setting up the appropriate parameters of the events occurring.

1.2 Thesis Structure

In this chapter we gave a brief description of what Projection Mapping is, as well as explore its potential in a nutshell. We also provided a brief description of what our application contemplates.

In Chapter 2, we provide an in-depth introduction to Projection Mapping. We begin with defining Projection Mapping from its history, to further investigate the fundamental techniques and concepts of Projection Mapping, realize the limitations and issues that result from the latter and adduce solutions relying on the latest advances in research on this field. Additionally, we showcase some essential Projection Mapping examples, including research papers and state-of-the-art software and applications. We, later, focus on Human-Computer interactions, to get familiarized with an interactive approach in Projection Mapping, which is directly tied to SceneWizard. We, then examine the impact of Projection Mapping on human perception and provide an overview of the latest applications and installations on Interactive Projection Mapping. Finally, we offer a brief review of the VideoMapping software, to conclude with the one we used for SceneWizard.

In Chapter 3 we assemble the necessary requirements, to consider SceneWizard a complete tool, as well as point out the reasons that lead to its creation. This analysis includes also all the Hardware and Software requirements, essential for developing and/or using SceneWizard. Furthermore on this chapter, we present the basic use-cases framework for SceneWizard. We divide the framework into three notations, underlying the different perspectives of the utilization of SceneWizard. Even though our tool is intended to Projection Mapping artists and designers, we do also analyze the use-cases from the End-Users prospect, as a performer, who runs SceneWizard and as an audience, who enjoys the results. Finally, we adjoin the use-cases of a developer, as it is very likely to be necessary due to the potential of expansion of SceneWizard.

Chapter 4 constitutes of an integrated manual of the implementation of SceneWizard. In this chapter, we provide a full analysis of all the components of SceneWizard (Kinect, Unity, and TouchDesigner),as well as how we implemented the connection between these components. Initially, we present SceneWizard's User Interface and the basic steps for its utilization. Firstly, we present our work on Kinect Depth Sensor and the building of our Gesture Database. We, then introduce the Sensor to our main software (Unity), which afterwards link to the Video Mapping software (TouchDesigner). We demonstrate all the critical scrutinies of SceneWizard's development, its architecture and rationalize how and why everything is linked in the way it is. SceneWizard is basically the interface between the used components, headquartered in Unity and we present the detailed implementation of its creation and functionality. Finally, we exhibit the utilization of SceneWizard, while preparing for our Demo presentation.

Chapter 5 provides a user study of SceneWizard, divided into two parts: initially the evaluation of SceneWizard as a tool and secondly the evaluation of the efficiency of our Gesture Database. We evaluate SceneWizard's tools and functionality by conducting the "Think Aloud" testing technique and present the adjustments performed on SceneWizard after the end of the evaluation process.

Finally, in Chapter 6, we summarize everything we delved into in the previous chapters. We concentrate more on what we achieved, as well as, what is yet to be accomplished with SceneWizard, as future work of development, enhancement and optimization of our tool.

2 Technological Background

2.1 Introduction

Projection Mapping, also known as, video mapping, or Spatial Augmented Reality (SAR), is a projection technique used to turn objects and generally surfaces, often irregularly shaped, into displays for video projection. This technique augments real world objects and scenes, by changing the look of the physical environment with projected light. Using a, or multiple projectors Projection Mapping renders virtual objects direct within or on the user's physical space. A key-benefit of Projection Mapping is that the user does not need to wear any equipment. Real objects can be physically handled and naturally manipulated to be viewed from any direction, which is essential for ergonomic evaluation and provides a strong sense of palpability. Unlike Virtual Reality, which immerses a user into a computer-generated environment, Projection Mapping (also knows as Spatial Augmented Reality) joins together physical and virtual spaces by creating the illusion that computer-generated objects are actually real objects in a user's environment. Below we can see a figure of the Continuum of advanced computer interfaces, based on Milgram and Kishino (1994):

The most common way to achieve this is, after the object which will be projected on is chosen or created, software is used to map the corners of the video output to the surfaces. First, one must choose the images or video to project. Then, place each video on its designed surface. Alternatively, one may choose to map the entire scene in 3D and attempt to project and mask the image back onto its framework. The next step is defined as "masking", which means using opacity templates to actually "mask" the exact shapes and positions of the different elements of the building or space of projection. In 3D Mapping, coordinates need to be defined for where the object is placed in relation to the projector, the XYZ orientation, position and lens specification of the projector have to result to a determined virtual scene. To achieve this end, there are various video mapping tools. Large projectors with 20,000 lumens output, or greater are used for large-scale projections such as city skyscrapers. A 2,200 lumen projector is adequate for projections under indoor light or theatrical lighting in most cases.

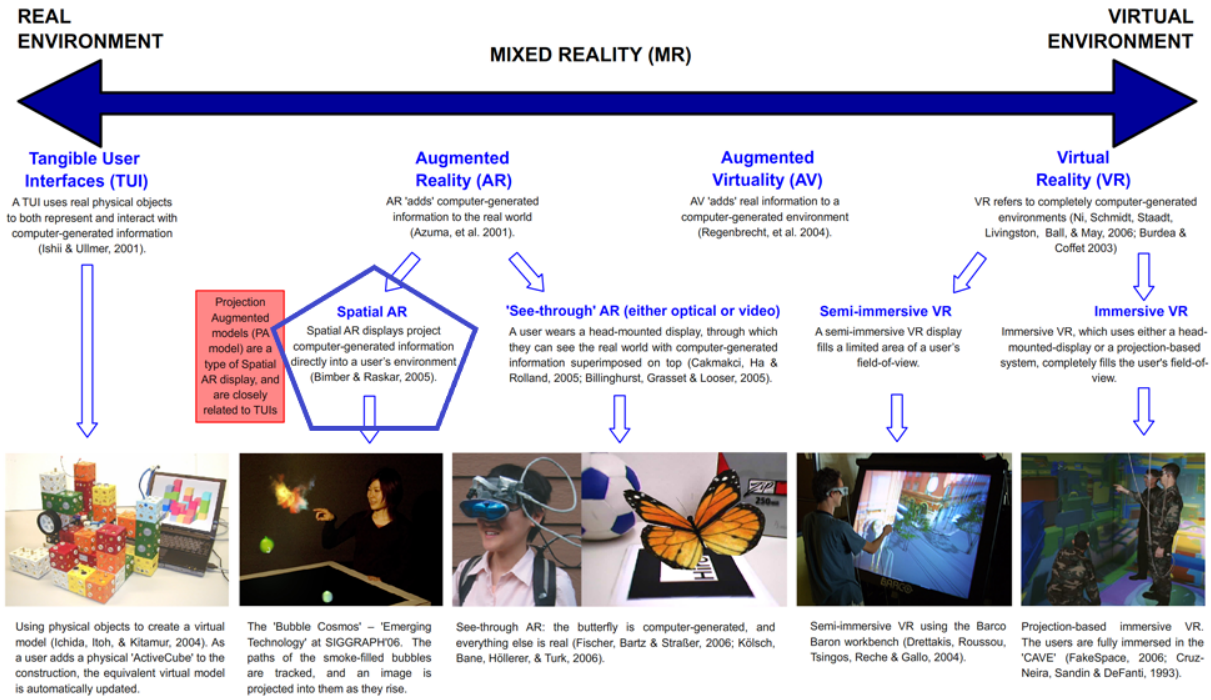


Figure 1: Continuum of advanced computer interfaces, based on Milgram and Kishino (1994)

Before we explore the methods of Projection Mapping, we will firstly refer to a history retrace of this field, by presenting few of the early applications that introduced Projection Mapping into academia and research. Then we will investigate the main issues addressing Projection Mapping and refer to propositions suggested to overcome these limitations. Next, we will talk about the present state of Projection Mapping while staging the current applications being utilized, mainly focusing on adding interactivity. Afterwards, we will postulate future directions of Projection Mapping, to continue with evaluating the benefits of including Interactivity into Projection Mapping. Additionally, we will focus on understanding specific gesture based Human-Computer Interactions and present the state-of-the art applications that inspired us into creating SceneWizard.

2.2 History of Projection Mapping

While Projection Mapping has recently exploded into the consciousness of various artists, designers, performers and advertisers everywhere, the history of Projection Mapping dates back to the 1990s, when it was referred to as "Video Mapping", or by its academic term "Spatial Augmented Reality", or "Shader lamps".

2.2.1 First Occurrence in 1969– Disneyland- "Haunted Mansion".

The story begins in 1969, when the first known instance of Projection Mapping onto a non-flat surface takes place, at the opening of the "Haunted Mansion" ride in Disneyland. The dark ride featured a number of interesting optical illusions, such as the "Grim Grinning Ghosts"(Figure:2), and "Madame Leota". The first included 5 singing busts that were singing the theme song of the ride. To accomplish this, they recorded

real singers on 16mm film and then used a projector to project the recording onto three-dimensional busts, which had the faces of the individual singers. This way the busts appeared as if they had been alive.

Figure 2: Grim Grinning Ghosts:Disney

Further down the ride the same technique was applied to present "Madame Leota". A mold of a face was encapsulated into a crystal ball, while a film loop played the recording back onto the mold.

2.2.2 1980-Michael Naimark-"Displacements"

The next record of Projection Mapping is from 1980, when artist Michael Naimark, created an immersive film installation, using Projection Mapping techniques. In this art installation, an empty(of people) room was projected, creating the illusion that people were actually there, interacting with objects. To do this, he first created and designed a room with physical objects, where he placed two actors. At the middle of the room, he placed an 16mm camera, on a rotating platform, which recorded continuously the actors while interacting with the objects in the room. He then replaced the camera with a rotating projector, so that the film is exactly projected on the place it was previously recorded. In addition, he spray-painted white the whole room, with the objects in place, so that the surfaces could act as displays. During the rotating Projection Mapping, all the object were perceived as very tangible, apart from the actors that appeared flat, due to their lack of physical representation.

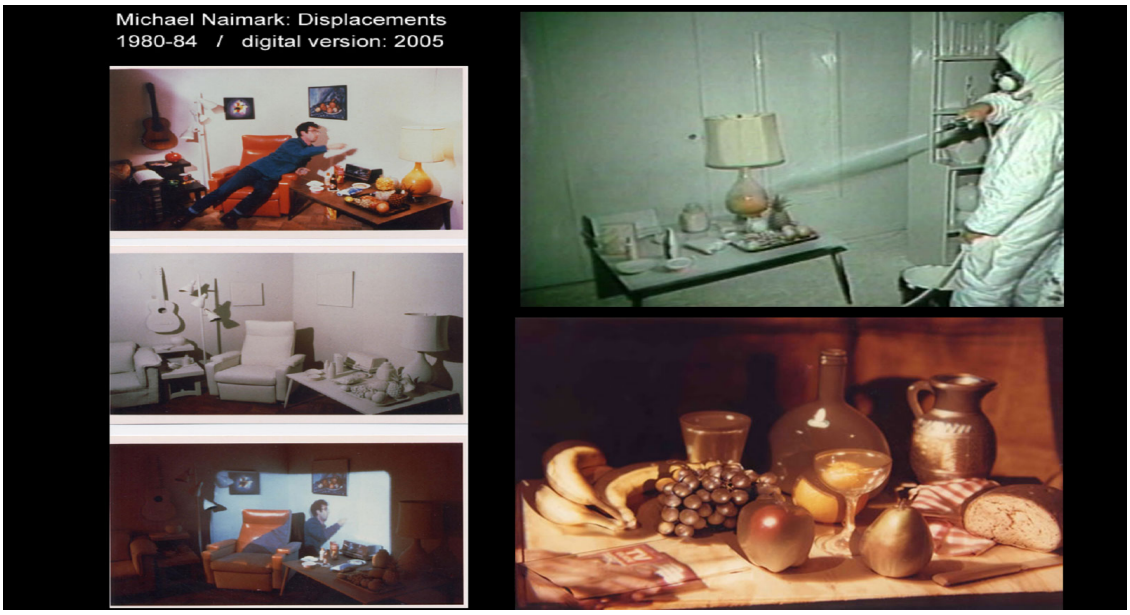
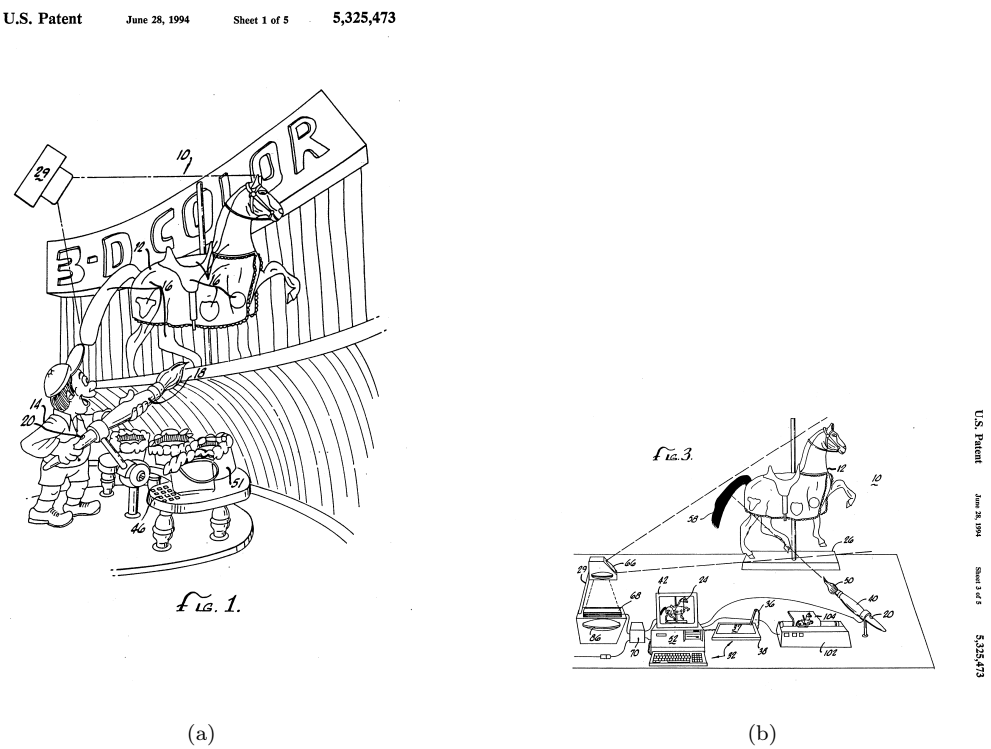


Figure 3: Michael Naimark-"Displacements"

2.2.3 90s Patents

- Disney – 1991- Apparatus and method for projection upon a three-dimensional object.
  Disney, not only pioneered the technology for Projection Mapping, they also have the earliest patent, a system designed for digitally painting an image onto "a contoured three-dimensional object".



(a) (b)

Figure 4: Disney Apparatus – 1991

- General Electric Co – 1993 - Projection of images of computer models in three- dimensional space GE also has a patent for Projection Mapping. More specifically a system and method for precisely superimposing images of computer models in three-dimensional space to a corresponding physical object in physical space. The system includes a computer for producing a computer model of a three-dimensional object. A projector means, projects an image of the computer model onto the physical object. A spatial transformation accurately maps the computer model onto a projection stage at the projector which projects a formed image in a three-dimensional space onto the physical object in a one-to-one correspondence.

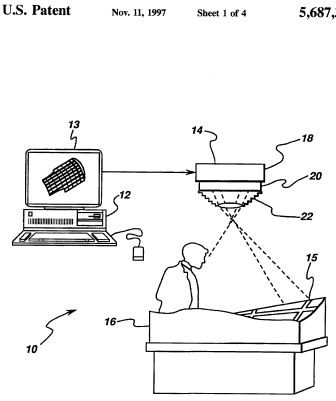


Figure 5: General Electric Co Apparatus– 1993

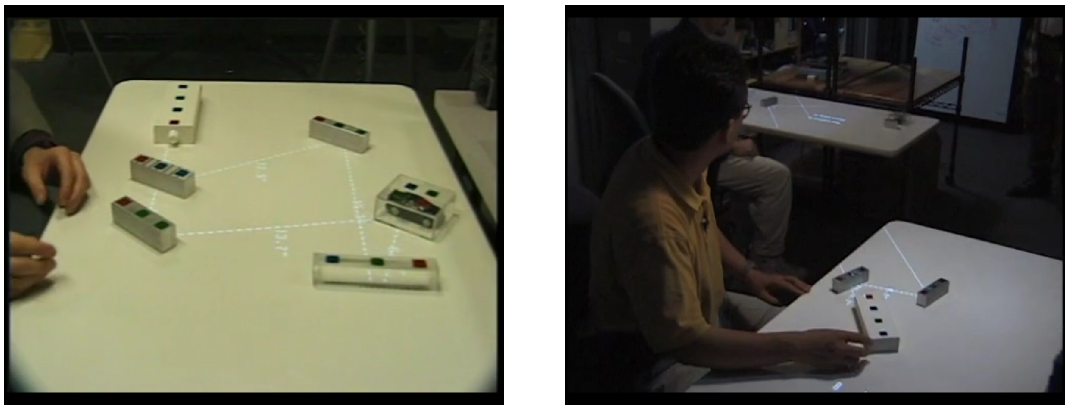
2.2.4 1998-University of North Carolina (UNC)-"The office of the Future"

Projection Mapping really started to get in traction when it was pursued academically. "Spatial Augmented Reality" was born out of the work of RaMesh Raskar, Greg Welch, Henry Fuchs and Deepak Bandyopadhyay, who had been students at the University of North Carolina. "The office of the Future" envisioned a world, where projectors could "cover" any surface. The concept was, to connect distant physical offices to each other by virtually projecting onto the other office. To accomplish this, they used depth camera's as well as normal cameras to initially record the distant office. Afterwards, with the use of multi projectors the office at the different location would be projected. This also captured the position of the spectator to adjust the projected image for perspective view. In the end, they made it possible to Skype for example, with life-sized versions of office mates or view life-sized virtual 3D models. This work even featured an early real-time imperceptible 3D scanner (like Kinect).

Figure 6: "The Office of the Future". A conceptual sketch of the office of the future. By replacing the normal office lights with projectors, one could obtain precise control over all of the light in the office. The image is intended to help differentiate between the projected images and the real objects in the sketch.

2.2.5 1999-John Underkoffer – I/O Bulb and Luminous Room

John Underkoffer developed his idea Input/Output Bulb, along with Luminous Room, which are the central ideas of his project. The project aimed in the pervasive transformation of architectural space, so that every surface is rendered capable of displaying and collecting visual information. According to J.Underkoffer the I/O bulb is the conceptual evolution of the ordinary light-bulb: one which not only projects high resolution information but also simultaneously collects live video of the region it's projecting onto. A Luminous Room is the structure that results from seeding an enclosed space with a multiplicity of coordinated I/O Bulbs÷enough, specifically, so that every location is treated by at least one I/O Bulb.



(a) (b)

Figure 7: I/O Bulb and Luminous Room

2.2.6 2001 – Shader Lamps

As a follow on to "Spatial Augmented Reality", RaMesh Raskar, student at the University of North Carolina at Chapet Hill, also invented a method called Shader Lamps. This computer graphic technique, is used to change the appearance of physical objects, using one or more video projectors, that project static

or animated texture or video stream. A 3D graphic rendering software is typically used to compute the deformation caused by the non perpendicular, non-planar or even complex projection surface. Complex objects (or aggregation of multiple simple objects) create self shadows that must be compensated by using several projectors. The objects are typically replaced by neutral color ones, the projection giving all its visual properties, thus the name shader lamps. The technique can be used to create a sense of invisibility, by rendering transparency. The object is illuminated not by a replacement of its own visual properties, but by the corresponding visual surface placed behind the object as seen from an arbitrary viewing point.

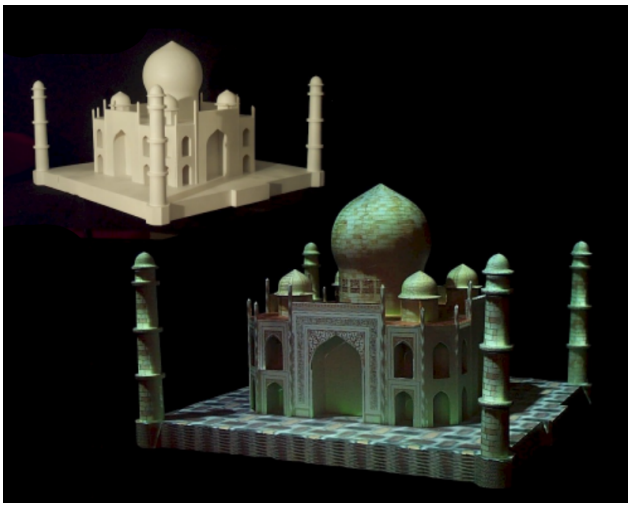


Figure 8: The underlying physical model of Taj Mahal and the same model enhanced with Shader Lamps

2.3 Registration Artifacts

2.3.1 Introduction

In this chapter we review the fundamental issues, produced while forming and projecting images using a projector. To present this review, several papers have been used in order to acquire the necessary information, based on the research that has and is being conducted on Projection Mapping. Each paper is referenced accordingly.

2.3.2 Image Registration Methods

In general, Image Registration is the process of transforming different sets of data onto a coordinate system. In Projection Mapping, it refers to the degree in which the visual image data is accurately presented on the real environment. Traditionally, a projector is used to create flat or rectangular two-dimensional images. However, a projector and a three-dimensional surface can be used in a variety of geometric configurations. In order to create a realistic look of the projected model, the image-to-geometry registration should be accurate enough. The problem of image-to-geometry registration has been thoroughly studied, and according to *Matteo Dellepiane* and *Roberto Scopigno* [1] a number of different solutions have been proposed:

- Semi-automatic Methods: A very robust approach is based on setting several 2D-3D correspondences: the correspondences are then used to estimate the cameras parameters using a minimization algorithm. The approach can be very time-consuming, especially when tens or hundreds of images need to be

aligned. A way that could minimize image acquisition and remove the need for registration is automatic planning of the images required, but this approach can only be used in controlled environments.

- Features and Silhouette-based Methods: The geometric features of the object can be used to find the registration of images. These features include points, lines, rectangles edge intensities, or even the silhouette of the object. Unfortunately though, these methods rely on the mandatory presence of these features on the object, hence they could be best applied on peculiar types of artifacts, as for example architectural scenes, if using clusters of orthogonal lines.

- Color-based Methods: Another feature that can be used (if present) is the reflection value (laser intensity) or color information that some 3D scanners acquire. This method could be used to define controllers that map the virtual scene onto the real scene, by detecting a specific color or LED, its position, direction and velocity.

- Statistical Methods: Other approaches try to catch the nonlinear correlations between the image and the geometric properties of the target surface. A measure which is extensively used in medical imaging is called Mutual Information (MI). It was pioneered by Viola and Wells and by Maes et al.[2] They suggested comparing the gradient variations of the image with the rendering of the 3D models, while showing the surface normals. Corsini et al.[3] in continuous, extended this algorithm by including other geometric properties, such as ambient occlusion and reflection directions, in the alignment algorithm.

- Multi-view Methods: Almost all the above mentioned methods are based on the alignment of a single image on the geometry. An alternative work relies on the fact that if a group of images has to be aligned on a model, it is possible to take advantage also of the relations between images. Zhao et al. Work,[4] amongst others, exploit Structure From Motion(SFM), during 2D/3D registration process.

2.3.3 Geometric and Photo-metric Concepts

On Projection Mapping, the image synthesis and projection using a projector can vary in conceptual framework. Rendering images of three-dimensional virtual objects onto three-dimensional geometry have various possibilities. To achieve a desirable accuracy in projected-based environments, the geometric relationship between the display components needs to be defined. According to *Oliver Bimber, RaMesh Raskar*, in their book "Spatial Augmented Reality-Merging Real and Virtual Worlds" [5], to describe this relationship, the rendering process, given a specific projector involves three geometric components that define the display configuration:

- Projector Model and Set-up: Hardware configuration plays an important role in Projection Mapping, since the projector's characteristics define the projected image. These characteristics include the type of the projector, as well as the position and orientation of it. The type and model of the projector determine the lens characteristics such as the field of view, focus, shift and intensity of the projected image. Simultaneously the position and orientation stipulate the shape, intensity as well as the distortion of the image. The projector set-up can be front projection or rear-projection. Knowing how to adjust this parameters results in better quality image.

  In addition to that, the number of the rendering projectors (using multiple projectors), also alternates the type of the resulted image as the field of view is widened. Adjusting the layers and positions of the images could rebound to the quality and size of the image.

- Shape Representation of the display portal: Matching a real world scene with its virtual counterpart, in order to mix them seamlessly, requires the designation of the projection target. The display surface can vary in types, shapes, sizes, states, materials and colors, fact that complexes the process of mapping straight beam lines accurately onto the 3D geometry. Surfaces are potentially planar, non-planar,

textured, deformable and/or moving, in a dark, bright or environment light. These non-optimized surfaces perplex the situation and create several technical issues, that need to be overcome in order to display desired appearances.

- User/ Users: In Projection Mapping an accurate spatial relationship between the user (usually user's head), the projector and the scene needs to be maintained, so that the user receives the full experience. Often, the concept involves the visual perception of an illusion, situation where the position, movement/velocity and direction of the user are taken under consideration. Moreover, Projection Mapping performances are usually targeting multiple users, since Spatial Augmented Reality provides AR experiences of a wide field of view.

2.3.4 Limitations and Technical issues

As a result of the geometric components, according to each application scenario, there are several technical issues and difficulties that result into categorizing Projection Mapping in the following sections:

- Static/Mobile Projectors: Projectors have traditionally being used as fixed devices, that render fixed images. However, nowadays projectors are small enough to be transported easily, as well as handheld. This compactness of new-generation projectors is generating new modes of use, away from fixed projector installations. Although, an arbitrarily placed projector should muddle up the image registration problem, advances in self-calibration camera techniques can automatically produce a desirable projection.

- Orthographic/Oblique Projection: Orthographic projection or sometimes orthogonal projection is a type of parallel projection, in which all the projection lines are orthogonal to the projection plane, resulting in every plane of the scene appearing in affine transformation on the viewing surface. Parallel projection is a rare and tedious concept, usually the projectors position is fixed and demands some kind of rotation to the projector to project on the desired surface. In general, when a projector is roughly positioned, it is likely to be oblique. Even if the projector is orthogonally positioned in large display systems, after a period of time it can become oblique due to mechanical or thermal variations. On the other hand, the projectors in oblique projection intersect the projection plane at an oblique angle to produce the projected image, as opposed to the perpendicular angle used in orthographic projection. The problem with oblique projection appears when three-dimensional objects need to be projected onto two-dimensional surface, in other words when displaying depth on the flat surface. When the projector's optical axis is not perpendicular to the display screen, the resultant image is keystoned and appears distorted. For example if you project the interior of a room on a wall, it can all look very realistic until, the viewer start to move from side to side. Then the illusion quickly becomes busted. Perspective changes without viewpoint, so a static projection, is not enough to keep up the illusion. Below is a figure image (Figure:9) to visualize the problem obtained from Teejo Renaud's Projection Mapping presentation [6].

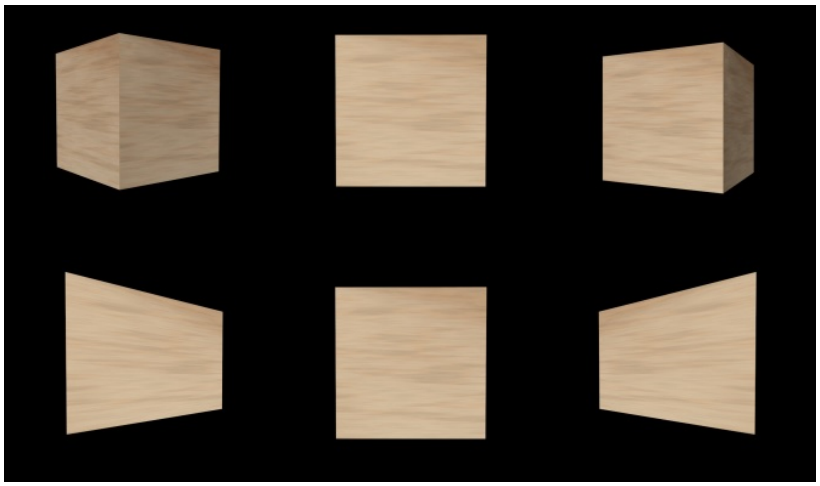


Figure 9: Perspective. The first row is how we would need to perceive a cube if we areplaced from its left side, centered in front of it and if we are standing at the right of it. The second row shows how we would perceive if the image is not adjusted to the perspective of the spectator.

- Short-throw/Long-throw Projection: To continue with, the projector setup can include a short or long distance throw, depending on the lumens of the projector, as well as the size of the projected scene. The distance throw creates a physical limitation called Depth of Field. Just like a camera, the projector has a projection area where the projected image is in focus, everything before or after that area is blurred, just like we can see in the figure below, obtained from the same source as the figure above:

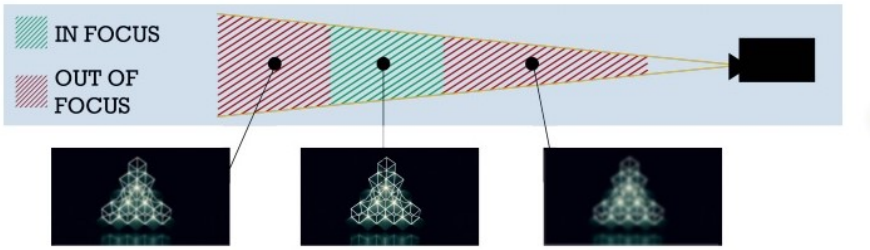


Figure 10: Depth Of Field of a Projector

- Single/Multiple Projectors: Ordinarily, people use projectors in order to complete a simple task like projecting a movie, or presentation. For this task, usually the required field of view does not over-go the field of view provided by a single projector. Bigger displays, though, like domes or architectural buildings, could not possibly be covered by the field of view of a single projector. Thus, the need of using multiple projectors to cover the display area is created. However, using multiple projectors, there are areas in the display area, that are illuminated by multiple projectors. As a result these areas appear brighter and cause the projector overlap intensity blending, as we can see in the figure below:

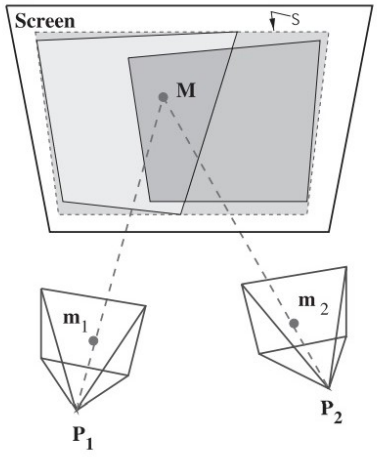


Figure 11: Overlapping Projectors

- Planar/Non Planar/Curved/Dome Displays: In Projection Mapping the type of the projected model surface can diversify in shape, providing numerous concepts. If the concept is simple enough, a planar (flat) surface could be adequate. For planar surfaces, projectors are typically mounted so that their optical axis is perpendicular to the planar display surface, however, when the projector optical axis is not perpendicular to the display screen, the resultant image is keystoned and appears distorted. As aforementioned in previous paragraph, we call this type of projection an oblique projection and the traditional projection an orthogonal. We can address the problem of rendering perspectively correct images with oblique projectors, by adjusting technically the image keystones from the projector. Additionally, we will describe in the next chapter, using Homography between the points on the display screen and the projector pixels, induced due to the planarity of the screen. A technique introduced by RaMesh Raskar, showing that oblique projectors can be used to easily create immersive displays. When projecting onto an arbitrary 3D surface, no matter how the projector is positioned and oriented towards the surface the resulting image will mostly look distorted. Although, from the projector's point of view the projected image looks perfectly aligned. So in order to achieve an undistorted look on a non-planar surface the projector needs to broadcast an image that depicts a view onto that surface from its own (the projector's) position. Creating a virtual replica of the real scene is a technique used by most Projection Mapping software, in order to align and register the surfaces correctly. We will extensively discuss the process of implementing this technique in our approach in Chapter 5. Furthermore, even though current software provide the possibility of manually aligning virtual to real objects, another technique was introduced by RaMesh Raskar in the late 90s, which involved calibrating the scene with a camera, to achieve the best surface estimation. We will discuss the efficiency of this method on the next chapter.

  Simultaneously, curved screens, or even colossal domes, are increasingly being used for high-resolution visualization environments. According to Raskar, when three-dimensional scenes are displayed on a curved screen, the images are perspectively correct from only a single point in space. As a result, depending on, if the user is moving or not, the scene calibration technique must recalculate the parameters, according to the user's position.

- Transparent/Semi-transparent Displays: When scouting markets for architecturally unique and interesting structures, we often encounter buildings that are primarily glass. Although these buildings may

appear architecturally compelling, glass surfaces do not properly reflect the light needed to achieve 3D projection clarity. This does not necessarily mean that buildings with glass windows are impossible. *ContraVision* [7] has found a solution to this conundrum, such as applying white adhesive to the projection surface of that structure to properly reflect the light from the projectors, as we can see in the figures below:

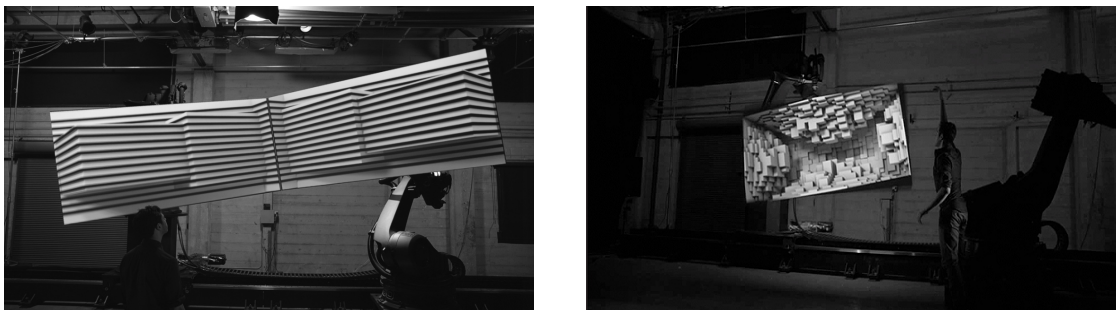
(a) (b)

Figure 12: ContraVision : Window Film for Digital Projection

Another option is, combining video Projection Mapping with custom cut Plexi with high end rear projection film. Such materials, can easily be accessible due to their availability (in hardware stores) and low cost. For instance, regular float glass or Plexiglass can be coated with a half-silvered film, in order to avoid transparency. The advantage of half-silvered film is that it can be coated onto a flexible carrier, such as thin Plexiglas, that can easily be bent to build curved displays. The drawback of such a material is its non-optimal optical properties. Instead of having light transmission and reflection factors of 50%, these materials normally provide factors of approximately 70% reflection and 30% transmission, due to their sun-blocking functionality. An alternative material is the so-called spyglass that offers better and varying transmission/reflection factors without sun-blocking layers. However, the reflective film is usually impregnated into float glass—thus it is not well suited for building curved mirror displays.

- Static/Moving Display components: Dynamic Projection Mapping, is a field that attracted much attention the recent years. In Dynamic Projection Mapping the projection surface is never at the same spot thus, the projector's output need to be adjusted, so that the dynamically-changing real world and virtual visual information are completely merged into the level of human visual perception. One way to achieve this is measuring the position of the surface and adjusting the projector output to that location. Using a camera and software that analyses the position of the surface multiple times per second, could give us the desirable result (constant scene calibration). Another way is, when knowing the surface's future position. For example, when a robot with a predefined path, holds a surface, we can predict the surface's position at any point in time, taking into account the speed and direction of the surface (robot). However, conventional approaches on Dynamic Projection Mapping have faced

some limitations, such as the target objects being limited to rigid, or high-speed camera calibration requirements, or high-speed projector requirements, which enable high-frame-rate and low-latency projection. In the next chapter we will mention the latest approaches to overcome this limitations. Below we can see an example of a Dynamic Projection Mapping:



(a) (b)

Figure 13: Box : An example of Dynamic Projection Mapping
https://gmunk.com/BOX

- Projector-based Illumination/ Environment-light Illumination: One of the main challenges in Projection Mapping systems, is the generation of correct occlusion effects between the real and virtual objects. In most cases, the existing environment illumination is applied, in order to lighten the scenery. Therefore, this light is also reflected off the objects' surfaces, interfering with the optically overlaid graphics and resulting in unrealistic appearances. In addition to that, the real environment is illuminated by physical light sources, while the virtual environment is illuminated based on synthetic light sources. This results in inconsistent shading and shadow effects unless the virtual light sources approximate the properties (such as position, direction, intensities, color, etc.) of the physical ones. Consequently, the physical contribution of the real light sources has to be neutralized to illuminate the entire environment based on the virtual lighting situation. This is only possible with controllable physical light sources, such as video projectors. For indoor situations, the environment illumination can be controlled and synchronized with the rendering process. The concept is called projector-based illumination and works, if the simple light bulbs are replaced by video projectors (sometimes referred to as light projectors). Additionally, it requires a physical environment that is initially not illuminated.

- White surface Augmentation/Textured Surface Augmentation: The management, processing and visualization of color information is a critical subject in the context of the acquisition and visualization of real objects. Diffuse objects, with arbitrarily reflection properties (color and texture), need a virtual model that will reflect the wavelengths of light, appropriate for the desirable or realistic appearance and convincing approximations. Matt surfaces are better than polished, since with matte surfaces the object itself seems more illuminated. With high reflective surface there is a chance that most of the light is reflected, so there will be less illumination of the object itself. For example, a mirror reflects nearby all light, but the mirror itself isn't more visible if a flashlight is shone onto. There are various research fields that have exploded since the up-rise of Projection Mapping, that focus in solving photo-metric issues, such as radiometric compensation, defocus compensation, shadow removal, inter-reflection compensation, which we will discuss thoroughly in the next Chapter.

- Single/Multiple Users: Projector-based spatial displays often suffer from the restriction of a single user (viewer) in case virtual objects are displayed with non-zero parallax. In order to solve the aforemen-

tioned perspective limitation, we need to adjust the projection according to the user's point of view, as long as we have a single user. In case of multiple users we can solve the perspectives problem, using multiple-projectors configurations. Additionally, for rendering occlusion shadows for multiple users, the viewpoints of each user, the intrinsic and extrinsic parameters of each light projector, as well as the virtual and the real scene must be known. If the same real surfaces are simultaneously visible from multiple points of view, individual occlusion shadows that project onto these surfaces are also visible from different viewpoints at the same time. Consider two observers: for instance, Observer A might be able to see the occlusion shadow that is generated for Observer B and vice versa. In addition, the shadows move if the viewers are moving, which might be confusing.

- Static/Moving View port: One of the fundamental challenges in Projection Mapping today is tracking moving user/users. The precise, fast, and robust tracking of the observer, as well as the real and virtual objects within the environment, is critical for convincing Projection Mapping applications. After mechanical and electromagnetic tracking, optical tracking became very popular. While infrared solutions can achieve a high precision and a high tracking speed, marker-based tracking, using conventional cameras, represent a low-cost option. Tracking solutions that do not require artificial markers, called marker-less tracking, remains the most challenging, and at the same time, the most promising tracking solution for future Spatial Augmented Reality applications.

2.3.5 Summary

In this subsection, we reviewed various Image Registration methods, the fundamental Geometric and Photo-metric concepts in using a projector for displaying images, as well as, point out the limitations that appear when applying these concepts. Our goal is to understand how a projector can be used beyond creating a flat and rectangular image. As we realized, a projector can be treated as a dual of a camera to create a myriad of applications on planar or non-planar surfaces, or even moving surfaces, thus we concluded on categorizing the concepts based on: 1. The Projector's Model and Set-up, 2.The Shape Representation of the Projected Portal, 3. The state of the User/Users and analyzed the resulting limitations, of each concept. On the section below, we will propose the latest advances and research, as state-of-the-art solutions, to overcome these limitations and problems.

2.4 Overcoming Limitations-Latest Advances in Projection Mapping

2.4.1 Introduction

For non-mobile tasks, novel approaches have taken augmented reality beyond traditional eye-worn or hand-held displays, thus, enabling new application areas. Projection Mapping or Spatial Augmented Reality (SAR) is a great way to create visual experiences, that can be enjoyed by large numbers of people at the same time in a way that brings people together – the exact opposite of how VR head-sets work. As a result, during the last decade, Projection Mapping has been tremendously widespread over the world. The goal is to seamlessly merge physical and virtual worlds, by superimposing computer generated graphics onto real surfaces. Projection Mapping approaches exploit components such as video projectors, optical elements, holograms, radio frequency tags, and tracking technologies, which due to the decrease in cost and graphics resources, has brought a considerable interest in using such augmented reality systems in universities, labs, museums, and in the art community. However, due to the diversification of Projection Mapping concepts and applications, as we mentioned before, several restrains have arisen that may obstruct Projection Mapping from being effortlessly performed. To meet the demands of the expanded bandwidth of options, researchers developed computational algorithms to project geometrically and photometrically correct images, by applying projector-camera systems (procams). According to the state-of-the-art-report from *Grundhöfer* and *D.Iwai* [8], we will briefly summarize the recent advances (in particular, the last 10 years) of Projection Mapping

algorithms and hardware solutions as means of displaying desired appearances onto non-optimized surfaces and introduce emerging applications that apply these new technologies.

In agreement with Grundhöfer and D.Iwai[9], we will mainly subdivide the algorithms into methods related to the geometric calibration of procams, as well as photo-metric calibration of the lightning parameters of the environment scene, since the same distinction we formed for presenting the issues produced by the variety of conceptual performance of Projection Mapping. Geometric calibration describes the task to estimate the three-dimensional properties of the devices, their relationship with respect to each other, and their relationship with the surface, in contrast to a projector-camera pixel correspondence estimation and mapping into a two-dimensional space. On the other hand, photo-metric calibration describes the task of calibrating and compensating the radiometric properties of the procams. Furthermore, the upcoming research field of Dynamic Projection Mapping will be briefly discussed. Below we can see a figure which represents the distinction of the algorithms, obtained from the aforementioned state-of the-art report:

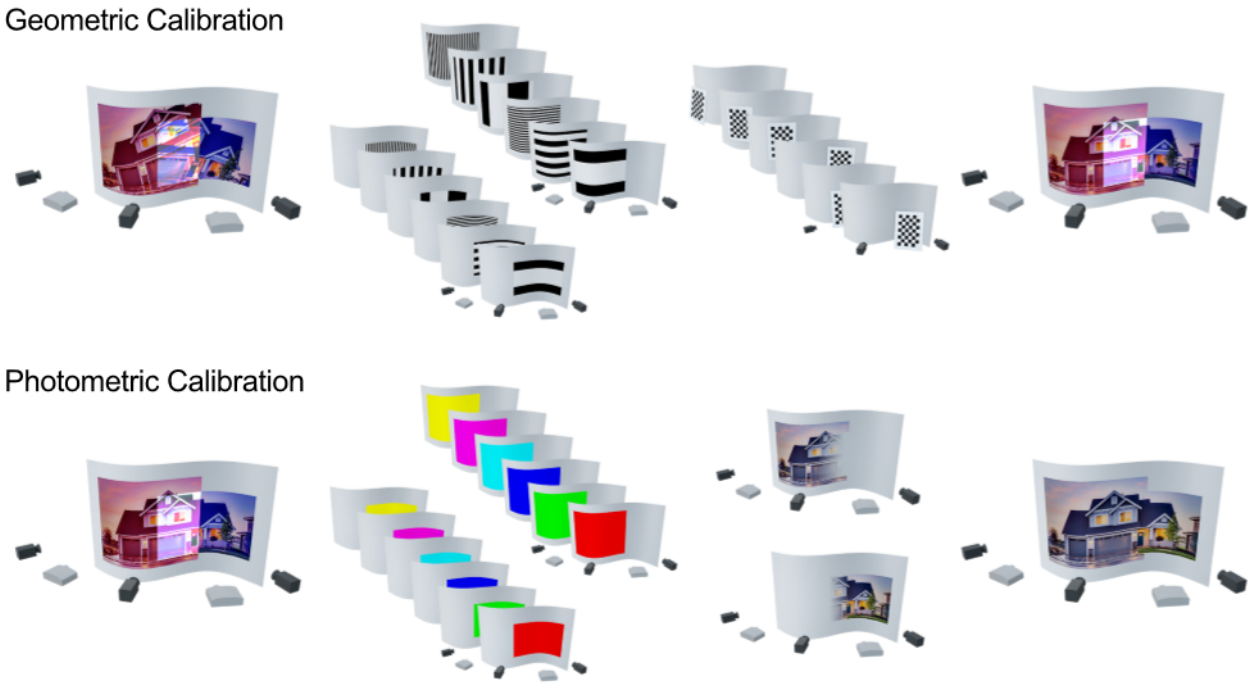


Figure 14: :Schematic overview of the fundamental calibration steps required by most procams applications. The process of geometric calibration is visualized exemplary on the top where cameras are used to capture projected structured light patterns, in this case gray codes,to generate projector to camera pixel correspondences. In combination with a multi-camera calibration, here, for example, based on planar checker boards, this is used to geometrically register the projectors to the surface and to enable the generation of a consistent projection onto a complex surface geometry as shown on the upper right. To also generate a colorimetrically consistent projection, additional color patterns are projected and acquired by cameras or other spectral sensors like colorimeters to photometrically calibrate the devices, and,in combination with information about the individual projector overlaps gathered from the geometric calibration, a completely consistent multi-projection system can be achieved as schematically shown in the lower right.

2.4.2 Geometric Calibration of Projector-Camera Systems

During approximately the last five years Projection Mapping has been greatly hyped and used in an enormous diversity of concepts, from being presented in venues, museums, park attractions, live festivals, to small abstract artistic projects. As a result, many companies, as well as individual programmers, or programming teams have developed tools to facilitate the process of Projection Mapping, pledging quick and efficient set-ups, with gratifying accuracy, that save much time and course of work. Microsoft is one of these companies, which produced the freely available Kinect-based RoomAlive Toolkit[10]. For procams calibration we briefly present the existing methods in correspondence with the image registration methods, as seen below:

- Semi-Automatic Procams Calibration Methods: Most methods to calibrate projectors usually start with one or multiple cameras, which are initially, either pre-calibrated, or not calibrated at all. The most common way to calibrate the intrinsics of procams involves capturing multiple images of a planar marker board, with unknow orientation, in order to estimate the focal depth in addition to other necessary parameters that will determine for example the lens distortion. Zhang et al.[11] presented the most commonly used method, as seen in his paper "Flexible camera calibration by viewing a plane from unknown orientations". His work is often used as a baseline to compare other calibration methods.

- Self-Calibration Methods: Instead of using calibration methods based on homography for calibration estimation, several semi- and full auto-calibration techniques for procams have been presented. Yamazaki et al.[12] presented one of the first methods that fully automate the calibration process of procams. However his suggestions have been hard to achieve in real-world conditions. A recent method as proposed by Garrido-Jurado et al.[13] offers a more flexible self-calibration method, although it assumes that the intrinsics of procams are known beforehand, which is usually not the case. Garcia et al.[14] proposed a method to calibrate a system that contains multiple cameras-projectors. The most recent method presented for calibration of multiple procams (MPCS) was proposed by Simon Willis and Anselm Grundhöfer[15]. This method uses at least two or more cameras to apply a full self-calibration without the requirement of any prior.

2.4.3 Dynamic Projection Mapping

The expeditious technological advancements in computational power of CPUs and GPUs have evolved to achieve super high-speed projections, which in their turn allow the tolerance of more complex set-ups and concepts. Thus, the field of Dynamic Projection Mapping, where procams are constituted of potentially moving components, has been increasingly studied within the research fields. According to Grundhöfer's report we classify the methods of Dynamic Projection Mapping, according to dynamic components' degree of freedom. Most systems define dynamic, in the sense that either the real-environment scene is rigidly transforming during the projection, or any of the procams components is relocating (camera or projector). Several approaches support applications of known rigid geometry and available tracking information. Inferior research has been conducted and published about real-time projections onto non-rigid, unknown moving projection surfaces. Below we will briefly summarize the recent developments on Dynamic Projection Mapping, focusing on rigid dynamic surfaces and non-rigid augmentations.

- Rigid Dynamic Projection: When focusing on Dynamic Projection Mapping of rigid objects, we require the pose estimation of the projector with respect to the 3D geometry, in order to render the graphics correctly. A very common method for pose tracking or estimation is applying visual markers. However, markers attached on a projection surface disturb projected results, as we can see the markers as a texture of the surface. To resolve this issue, a combination of techniques is required with radiometric compensation. Other suggestions include replacing the markets with tiny photosensors. Another

method was proposed, supporting a system for Dynamic Projection Mapping by Hashimoto et al.[16] He and his team suggested a system effective machine-learning and high-speed edge-based object tracking using a single IR camera. Similar methods have been proposed utilizing Kinect Depth Sensor.

Moreover, in Dynamic Projection Mapping, there is a perceived lagging, resulting from the inevitable end-to-end latency, which further pushes the boundaries of research in this field. One of the recent papers presented has been the one from Yoshihiro Watanabe, Toshiyuki Kato and Mashatoshi ishikawa[17] , who extended the work based on deformable dot marker clusters for tracking, combined with high-speed projector to achieve astonishing results in high quility outputs and lower latencies, of the order of milliseconds.

- Non-Rigid Dynamic Projection : A projection-based mixed reality system was proposed by Punpongsanon et al.[18] for tangential deformation of nin-rigid surface by estimating the deformation using infrared textures, which we can translate as invisible markers. Alternatively, Fujimoto et al.[19] proposed the usage of retro-reflective markers, also invisible, to measure the surface's deformation.

  A system to dynamically augment human faces using projection was presented by Bermano et al.[20]. Their system performs human face tracking without markers, in order to detect expressions of the face and adapt the projection. For further state-of-the-art applications on non-rigid Dynamic Projection Mapping, concern our source report[9].

2.4.4 Photometric Calibration (Radiometrical Control of Projector Camera Systems)

In this subsection we will discuss about the difficulty of accurately mesuring and controlling the lighting parameters of the environment of a projection. The light being studied is the light emitted from the projectors and reflected from the surfaces. As we know from the properties of light, depending on the surface material properties, the light that reaches them can be reflected, refracted, scattered and absorbed. Hence, only a fraction of the light's original intensity and color is depicted. For Lambertian surfaces (completely diffuse surfaces), the amount and color of reflected light depends on several parameters, such as the surface's material color (M), the light color and intensity that leaves the source (I), as well as the distance (r) and the incidence angle (a) of light rays with respect to the surface, together called form factor (F). For perfectly diffuse surfaces, Lambert's law approximates the diffuse reflection of light for each spectral component with

$$R = IFM,$$

where

$$F = \cos a/r2.$$

Even though, basic linear models for describing local color transformations already have been thoroughly studied, there are recent advances in enhancing the radiometric compensation techniques, to overcome the lighting limitations. Below we will briefly present few of them.

- Local Per-Pixel Radiometric Compensation: The basic methods of radiometric compensation, study the light configurations between camera and projector as a multiplication of the corresponding linear matrixes. The former researches in this field have been updated by new, more advances researches, such as the one presented by Chen et al.[21], who proposed a method to describe the property of how color channels of procams interact with each other, based on a single color mixing matrix, rather than a different matrix for each component.

  When projecting onto dark or strongly saturated surface pigments, we receive several compensating errors, regarding the frailty of the projector to produce a larger color gamut. Numerous methods have been suggested, on automatically targeting and adjusting colors outside of the available gamut.

Christoffer Menk and Reinhard Koch[22] presented their work on physical-based augmentation of real object under the influence on ambient light, by calculating the spectral data for each component of the procams independently, as well as their influence in the simulation, in order to later adjust the projection.

Furthermore, using non-linear thin-plate-spline interpolation technique is a novel approach presented by Grundhöfer[23], where high-quality photometric compensation is generated, without the requirement of pre-calibrated procams. In addition to this, he along with Iwai[24] extended this approach and came up with a method to reduce perceived artifacts at strong texture edges, which result from the unavoidable fact that projector pixels might hit multiple surface pigments with varying reflectance properties at texture edges, leading to artifacts due to overcompensation.

Moreover, the pose of the available projectors according to the scene has a significant impact on the visual quality of the projection. Thus, various approaches have been suggested on automatically calculating the optimal placement for the projectors, in order to achieve the ultimate compensation effects, such as the work presented by Alvin J. Law, Daniel G. Aliaga and Aditi Majumber[25].

- Inter-Reflection Compensation: Some methods that have been suggested in order to improve the image quality, achieve this only locally, since the compensation is carried out independently for each pixel or, incorporating locally neighboring pixels. However, these methods are not able to handle and neutralize any kind of global illumination effect. When light reaches a surface, diffuse inter-reflections will happen at any concavity, which results in a degradation of the images quality. Bimber et al.[26] initially presented a method of compensating the influence of these inter-reflections, by calculating the global illumination. His method have been later optimized in the terms of inverse radiosity techniques. Siegl et al.[27] presented more recently a publication studying the same issue, compensating stray-light in real-time dynamic, multi-projections. A more specific study was done by Habe et al.[28], who published a paper on an inter-reflection compensation algorithm specifically designed for dome-shaped projection surfaces. A more generall suggestion was posed by Wetzstein and Bimber[29], performed inter-reflection compensation by inverting the light transport matrix between procams.

  These aforementioned techniques on inverse radiosity share the same principle of reducing illumination in areas where strong inter-reflections occur. However, this results in the reduction of the overall intensity of the projections. Further research has been conducted, as a result of overcoming this issue. [30] By using for example UV-reactive photo-chromic surface materials, a more efficient inter-reflection compensation can be achieved, due to the fact the these materials change their reflection propertied from bright to dark when illuminated by high power ultraviolet radiation.

- Closed-loop Radiometric Compensation : Several researches focus deeper into the appearance control of moving objects by applying closed-loop radiometric compensation techniques, while using a coaxial procams as proposed by Amano and Kato[31]. The proposed method employed MPC algorithm for the projector camera system and enabled arbitrary appearance control, such as photo retouching software, in the real world. Advocating the ease of the image registration, they continued their research[32] on the optical calibration of a scene with coaxial procams and proposed a technique where a spatial grid pattern is projected onto a differently shaped grid surface such that the alignment accuracy can be directly estimated by observing the camera, while moving the latter. The same principle of a coaxial procams is used to visually manipulate material appearance[33], microscopic specimens[34], shading illusions[35], and context aware illuminations[36].

- Supporting Multiple Users In order to reduce confusing occlusion effects when the projection includes multiple viewports (many viewers), Rashkar et al.[5] suggested some methods:

- Method 1: In order to generate correct lightning effects on the real scene's surfaces, we can utilize the fact that, occlusion shadows generated for other viewpoints, are the umbral hard shadows that are cast by the virtual scene with a light source positioned at the other viewpoints' locations, by attaching a point light to each viewpoint, in combination with matching hard shadows on the real scene's surfaces.

- Method 2: The interference between individual occlusion shadows can be minimized by ensuring that they are generated only on those real surfaces that are visible from the corresponding viewpoint. However, in the end all view-dependent computations are finally done for the projector's perspective, since they get rendered from its viewpoint.

- Method 3: Each projector is complemented by a video camera that is used to dynamically scan reflectance information from the surfaces of the real objects. Knowing this reflectance information allows us to render occlusion shadows for each observer into a stencil buffer first, by rendering the real scene's geometry from each observer's perspective, as well as adding the corresponding occlusion shadows via projective texture mapping. After the stencil buffer is filled in the appropriate way, the real scene's reflectance map is rendered into the frame buffer, shaded under virtual lightning and finally the virtual objects can be added to the observer's view. For more information on this technique, readers can trace our referenced book[5].

2.4.5 Hardware

Projection Mapping faces several technical limitations, as seen in the previous chapter, due to the hardware's restricted capabilities. However, the rapid rate of development and the consequent reductions in prices of hardware electronics, developing in the sphere of automations, encourages researchers and academics on studying technologies that go beyond algorithmic improvements, in order to further expand the quality of the images for arbitrary surfaces by applying an emerging approach: Computational Projection Displays - a joint design of display hardware, optics and computational algorithms.

- High Dynamic Range Projection: The dynamic range or contrast of a projection display is defined as the difference between the darkest and lightest tones in a projected image(ratio of the maximum to minimum luminance). In the real world, however, the dynamic range is extremely wide (>1,000,000). As a consequence, in order to realistically render computer generated images, while displaying them in real environment, higher dynamic range is required. The present state of projectors, however, offers restricted support of dynamic range of between 1,000:1 and 6,000:1.[37] Several solutions have been presented to overcome the contrast limitations. These solutions can be subdivided, according to the methods utilized trying to achieve this goal, either by reducing the black-level without additional mechanical adjustments, or by adding hardware which locally amplifies the amount of photons.

  HDR projection has been mainly achieved by applying the double modulation principle, according to which the emission of a light source is spatially modulated twice at cascaded light, blocking spatial light modulators (SLMs), such as in a digital micro mirror device (DMD) or a liquid christal display (LCD), to reduce the luminance of dark pixels (or black level), while maintaining those of bright pixels.

  A recent and novel energy-efficient method, was proposed by Reinald Hoskinson et al.[38], where the double modulation method is applied. In this research they presented a proof of concept projector with a secondary array of individually controllable, analog micromirrors, added to improve the contrast and peak brightness of conventional projectors. The micromirrors reallocate the light of the projector lamp from the dark parts towards the light parts of the image, before it reaches the primary image modulator. Each element of the analog micromirror array can be tipped/tilted to divert portions of

the light from the lamp in two dimensions. By directing these mirrors on an image-dependent basis, they can increase both the peak intensity of the projected image as well as its contrast. The big advantage of the light-reallocating double modulation methods is the significantly limited light loss and the extremely high local intensities, which can be generated by focusing all available light onto a single pixel region. However, these methods produce further difficulties on the content generation, since a specific light budget needs to be spatially distributed in such manner without any flickering.

Even a projector with an ideal, unlimited dynamic range would get disturbed by global illumination effects, such as inter-reflections, which increase the black-level, consequently decreasing dynamic range. As a result, it is wiser to focus on optimizing the overall projection system, in spite of targeting dynamic range. To this end, researchers have proposed to spatially modulate the reflectance pattern of a projection surface to suppress the elevation of black level[39, 40, 41].

An alternative method, to generate HDR locally, can be achieved by using galvanoscopic laser projectors. In addition to this, in order to optimize the spatio-temporal scanning order and speed of such laser projectors, another technique was introduced by Grundhöfer[42].

- High Speed Projection: High speed projection systems enable a much higher frame-rate than a normal video rate (60 Hz). It has been achieved using DLP projectors that represent an 8-bit pixel intensity by controlling a MEMS mirror flip sequence at thousands of frames per second. However, nowadays researchers focus more on the high speed projection of images, that can be essential to humans, within Dynamic Projection Mapping(described in a paragraph above). When projecting onto moving components, it is crucial to maintain the virtual-to-real object alignments, since any errors on the image registration can significantly degrade the sense of immersion. Ng et al.[43] proposed a hybrid system that provides low-fidelity visual feedback immediately, followed by high-fidelity visuals at standard levels of latency. They showed that misalignments have been perceived, when the latency between the inputs and displaying has been greater than 6.04ms, which translates into the minimum desired frame rate.

  To resolve the latency issue Grundhöfer[9] addressed two solutions, as they have been presented. On the first, the direction of the image produced by the projector is controlled rapidly using a dual-axis scanning mirror galvanometer system, in order to perform Dyamic Projection Mapping. The second solution involves applying high-speed projectors that can display 8-bit images at several hundreds frames per second with low latencies. Watanabe et al.[44] developed a device that achieves up to 1,000 Hz, whereas Regan and Miller[45] proposed a technique to reduce motion blur artifacts in such situations using a high speed projector.

- High Resolution Projection : During the last decades, researchers have come long way on the resolution issue, from SD to HD, and now making a quantum leap into 4K, 8K resolutions and beyond. Resolution optimization is a topic in vogue, since the human's eye resolution, is quite high, approximately between 0.4 – 1 arcminute. The human visual perception of detail does not solely depend on the ability of the eyes to resolve the smallest features in a given subject, but also depends on the distance between him and the object being projected. There is a difference between perceived resolution and actual pixel resolution – the only thing that matters to the viewer is "perceived" sharpness and detail. Additionally, the human vision is divided into central and peripheral angles of view and as the eyes focus on the central view, they have the ability to move in order to fill in the missing information of the general picture. Mathematically, there is no doubt that increasing the pixel count gets the job done, but beyond a certain threshold the returns drop off towards zero while exponentially increasing data rates. However, when projecting visual content, the case often includes large crowd of people, therefore increasing the resolution, as in image size is highly in demand.

  There are two main approaches on achieving this goal, one, by using multiple projectors, adding-up

each and every projector's resolution in order to generate a higher resolution and second by optimizing a single projector's system. In a multi-projector approach several techniques have been proposed to resolve issues, such as differences of luminance and chrominance, caused by overlapping areas amongst multiple projectors. An interesting approach that considers the perplex functionality of the human vision, considering the varying property of retinal acuity, was proposed by Dalsuke Iway et al.[46] Their method utilizes two overlapping projectors, corresponding the central and peripheral human vision, to produce a super-resolution and reduce motion blurring effects. Projection-based super resolutions have been thoroughly investigated by several groups[47][48].

In contrary to the latter approach, Will Allen et al.[49] presented a single-projector approach, called "Wobulation". This is a pioneering technique, which shifts sub-frames of a projected image by a fraction of a pixel and projects them in a rapid succession, so that the slightly shifted images (generated sub-frames) are overlapped, seamlessly appearing as a superimposed image. The technique is being used as a standard technique for consumer- grade 4K projection systems. We recommend readers interested in these techniques to refer to a book [8]for more details.

- Increasing Focal Depth : As we have already discussed in the section about limitations of Projection Mapping, the focal depth of projectors is an additive difficulty that needs to be eliminated. The hardware design of the projectors leads to a shallow depth of focus and in concepts like Dynamic Projection Mapping, the focal depth can cause serious blurring issues. As a result, extending the projector's depth of focus (DOF) is crucial for the future of Projection Mapping.

  This field's techniques fall into utilizing single or multiple projectors as well. Single projector approaches apply techniques of calculating the defocus blur of the image before the projection, by convoluting a point spread function (PSF) with the original image, resulting into a digital sharpening of the original image. Then by inversing the process, therefore by deconvoluting (usually with the Wiener filter), we can correct the original image and display a defocus-free image. Zhang and Nayar[50] formulated image correction as a constrained optimization problem.

  During the last decade, however, new optical design systems have come to introduce novel approaches on extending the DOF of a projector. Dalsuke Iwai et al.[51] presented a cost-efficient approach, which extends the DOF of a projector by fastly sweeping forwards and backwards the focusing distance, which results in optimizing the PSF of a pixel over a sweep period and allows the projection focus at any point within the given range.

  In a multi-projector approach, deconvolution is not required, unless dynamic state of objects is not included. A novel approach was proposed by Nagase et al.[52] , where PSF is calculated automatically after calibrating the geometric relationships of the components, followed by a selection of the optimal projector for each moving surface.

- Light Field Projection : Glasses-free 3D or light field projections, are systems that provide projections that are physically correct for a wide range of perspectives views, hence observers do not need to wear special glasses. These systems, however, suffer from the loss of spatial resolution and light throughput. Several solutions have been proposed to overcome these limits, by utilizing a single or multiple projectors.

  Jurik et al.[53] proposed a novel principle, where instead of a projection screen being used, each projector acts as one angularly-varying pixel, that emits view dependent colors on the same object.

  In a single projector approach, researchers proposed to use an array of LEDs that illuminates a digital micro mirror device (DMD) from different directions at a high-speed, sequentially to time[54][55]. Additionally, Hirsch, Wetzstein and Raskar[56] suggested a compressive light field projection system, where they built an LCoS-based dual layer light field projector, along with an angle expanding screen,

where non-negative light field factorization is applied at a high-speed range to decompose an original light field, in order to achieve super-resolved and high dynamic range 2D image display

- Multi spectral projection: The present hardware state of the projectors allow only a limited color gamut, thus an optimization of the available colors is highly in demand. In the early stages of this research field pioneering designs presented an extended projector color gamut, either by increasing the number of color primaries, or the number of color filters.[57],[58] In other work, a flexible gamut projector was built, which allows color pimaries to be dynamically selected by non-negative matrix factorization that decomposes a target image into a set of adaptive color primaries and corresponding pixel values.[59] In a different concept, Li et al.[60] suggested a method that focused on a good spectral reproduction, while enriching the color gamut.

2.4.6 Summary

This report, according to Grundhöfer's Projection Mapping State-of-the-Art Report[9], covered the recent advances in the research fields related to Projection Mapping applications. We summarized the novel enhancements, to simplify the 3D geometric calibration task, which now can be reliably carried out either interactively or fully automated, as well as, present further improvements regarding radiometric and photometric calibration and compensation techniques. Moreover, we discussed about the latest hardware innovations concerning the hardware utilized for Projection Mapping, including dynamic range, frame rate and spatial resolution and depth-of-field, light-field and multi-spectral projections. The figure below, as obtained from Grundhöfer's report, summarizes any potential remaining issues of Projection Mapping for future research.

	Topic	Issue
Algorithms	Geometric calibration	Full online-self calibration
	Dynamic projection mapping	Dynamic full body augmentations
	Radiometric control	More accurate control for non-lambertian surfaces
Hardware	High dynamic range projection	Dynamic HDR under environmental light
	High speed projection	Efficient rendering and data transferring
	High resolution projection	Dealing with close-up situations in interactive systems
	Increasing focal depth	Avoiding contrast reduction
	Light field projection	Light field manipulation on arbitrary surfaces
	Multispectral projection	Full spectral color reproduction by a single projector

Figure 15: Table 1 : Future of Projection Mapping

2.5 Examples of Spatial Augmented Reality

2.5.1 Introduction

Tremendous advancements in the fundamental technologies, such as geometric calibration and radiometric compensation, have been expanding the application fields of Projection Mapping. This chapter provides a link between the previous, more technical, chapters and an application-oriented approach and describes several existing spatial AR display configurations. We cover applications in which geometric registration and radiometric compensation are core components, as well as outline examples, that utilize the projector-based augmentation concept in both a small desktop (e.g., RoomAive[10] approach) and a large immersive (e.g.,

MORI Building Digital Art Museum[61]) configuration.

Another goal of this chapter is to show that Spatial Augmented Reality display configurations can be used successfully and efficiently outside research laboratories. The L'Atelier des Lumières, created by Culturespaces[62], for instance, has been presented to more than 600.000 visitors, in the year 2016. Using 140 video projectors and a spatialised sound system, the multimedia equipment can cover a total surface area of 3,300 m2, extending from the floors to the ceilings and over walls up to ten metrers high. L'Atelier des Lumières is an indicator of the fact that it is possible to make the technology (soft and hardware) robust enough to be used by museums and in other public places.

Figure 16: L'Atelier des Lumières, Exhibition: Van Gogh, Starry Night, created by Gianfranco Iannuzzi, Massimiliano Siccardi with the musical collaboration of Luca Longobardi

Figure 17: L'Atelier des Lumières, Exhibition: Dreamed Japan, Images of the floating world, created by Lucia Frigola, Cédric Péri, Sergio Carrubba, Paola Ciucci.

2.5.2 2D Camera

A team of researchers introduces a projection-based system where users can build their own physical world, map virtual content onto their physical construction and play directly with the surface using an IR detected stylus[63]. In a similar manner, a theme park ride utilized procams registration that allows players to change the surface textures with gun-like input devices[64]. Moreover with this technique, an interactive architectural daylight model has been achieved, where a user performs the action of moving the walls[65],[66]. The system measures their poses by an overhead camera, and updates the projected lighting simulation[67]. Willis et al.[68] utilized a hand-held projector that can project IR and visible images from the same projection lens for displaying IR/AR markers and visible images at the same time.

Another emerging research trend is to alter perceptions other than visual perception by changing the colors or textures of real surfaces including human bodies. An experiment was conducted, where it was shown that by changing the appearance of a hand, some haptic sensations could be controlled[69],[70]. In addition to this, the sense of taste could also be altered by changing the color of food by Projection Mapping[71].

2.5.3 Depth Camera

A variety of applications have been proposed, to facilitate the registration problem, by using depth camera systems (procams). One of the most active research domains of procams applications is Projection Mapping games. Several game applications have been developed, such as IllumiRoom[72], which extends the display area of TV games over surfaces around the TV screen[73]. Other applications combine multiple depth cameras and projectors for interactions on, above and between surfaces, that are being projected. Furthermore, Yamamoto et al.[74] turned the hand and arm of a user into an interactive surface, using an RGB camera, while the depth camera supported the interactions.

2.5.4 Radiometric Compensation

Radiometric compensation (RC) allows the display of desired colors on arbitrary surfaces that have spatially-varying reflectance properties. Grundhöfer et al.[75] proposed an idea of turning everyday surfaces into blue screens by controlling the surface colors for chroma keying in video composition, whereas Aliaga et al.[76] applied an RC technique to virtually restoring the color of historically important objects in Projection Mapping. RC has been also applied in interactive applications, as a means of converting physical documents/books into pseudo-transparent to support document search on a physical desktop.[77] A group also applied an RC technique to optically embed graphical information in shadows[78]. Finally, Amano et al.[79] applied RC techniques, in order to build an appearance control tool to support visually impaired people.

2.5.5 Room Alive

In recent years the way people play video games changed, from the VR immersive game field to interacting more naturally with the game in a real world by moving our body (i.e., Nintendo Wiimote, Microsoft Kinect and PlayStation Move). Coupled with new display technologies (e.g., Oculus Rift), we can now feel more "in the game" than ever before. RoomAlive[10] is a proof-of-concept prototype that transforms any room into an immersive, augmented entertainment experience. RoomAlive's system enables new Interactive Projection Mapping experiences that dynamically adapts content to any room. Users can touch, shoot, stomp, dodge and steer projected content that seamlessly co-exists with their existing physical environment. The basic building blocks of RoomAlive are projector-depth camera units, which can be combined through a scalable, distributed framework. The projector-depth camera units are individually auto-calibrating, self-localizing, and create a unified model of the room with no user intervention, in order to adapt the augmented content to the particular room, for instance spawning enemy robots only on the floor of the room. The content also reacts to the user's movement. RoomAlive uses a distributed framework for tracking body movement and touch detection using optical-flow based particle tracking and pointing using an infrared gun.

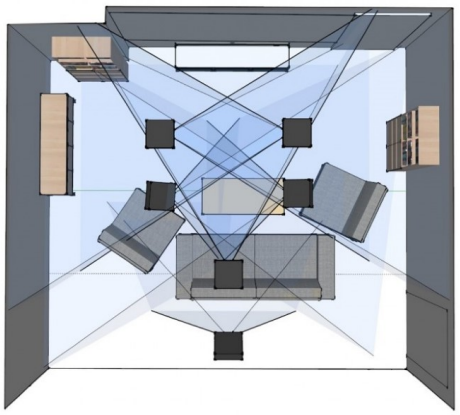


Figure 18: RoomAlive Installation: Procams installation in a room.

2.5.6 Summary

In this subsection we summarized several Projection Mapping applications, as latest in the research field of Projection Mapping. We covered novel enhancements, as well as presented on-point applications in different domains, such as museums, home entertainment, science and industry. The robustness, attractiveness, and

efficiency of technology are essential for a successful application of Projection Mapping in public places, such as museums. Some of the configurations that have been described in this subsection tell the latest success stories and provide a promising perspective for upcoming installations.

2.6 Human – Computer Interactions

2.6.1 Introduction

Humans interact with computers in many ways; the interface between humans and computers is crucial to facilitating this interaction. Desktop applications, internet browsers, handheld computers, and computer kiosks make use of the prevalent graphical user interfaces (GUI) of today. Voice User Interfaces (VUI) are used for speech recognition and synthesizing systems, and the emerging multi-modal and Graphical user interfaces (GUI) allow humans to engage with embodied character agents in a way that cannot be achieved with other interface paradigms. The growth in human–computer interaction field has been in quality of interaction, and in different branching in its history. Instead of designing regular interfaces, the different research branches have had a different focus on the concepts of multimodality rather than unimodality, intelligent adaptive interfaces rather than command/action based ones, and finally active rather than passive interfaces.

The Association of Computer Machinery (ACM) defines human–computer interaction as "a discipline concerned with the design, evaluation and implementation of interactive computing systems for human use and with the study of major phenomena surrounding them". An important facet of HCI is user satisfaction (or simply End User Computing Satisfaction). "Because human–computer interaction studies a human and a machine in communication, it draws from supporting knowledge on both the machine and the human side. On the machine side, techniques in computer graphics, operating systems, programming languages, and development environments are relevant. On the human side, communication theory, graphic and industrial design disciplines, linguistics, social sciences, cognitive psychology, social psychology, and human factors such as computer user satisfaction, are relevant. And, of course, engineering and design methods are relevant." One of the basic elements of the human – computer interaction is the user interface, which is supposed to bring the feeling of comfort while using computer, and to facilitate the usage of computers. The goal of human – computer interaction on the user interface level is to perform operations effectively, the control of the device operations, and to get feedback from machine that helps operator in decision making process. User interface is the system upon which the users interact with the machine. It involves hardware and software components. User interfaces often reflect the general philosophy of the operating systems makers, and are also the trademark that distinguishes them on the market. The development of user interfaces, also with the development of all components of operating systems, is determined by and in course with the development of computers themselves[80].

One of the most important moments in software development is turnover from design that was primarily targeting intensive computations to the design targeting intensive presentation. The history of this development process can be divided in three eras: batch processing (1945 – 1968), command line usage (1969 – 1983), and graphical user interface (GUI) era (after 1984). However, in the subsections below we will discuss an alternative Gesture-based HCI, to continue with an example of implementing Gesture-based HCI with Kinect Depth Sensor. Furthermore, we will thoroughly analyze Kinect's technology and components, in order to understand the utilization of the sensor, for our further implementation. Next, we will inspect the gestures recognition methods, so we can later justify our implementation methods. Finally, we will present several Interactive Projection Mapping applications to conclude the state-of-the-art chapter.

2.6.2 Gesture-based HCI

One of the most important research areas in the field of Human-Computer-Interaction (HCI) is gesture recognition as it provides a natural and intuitive way to communicate between people and machines. Gesture-based HCI applications range from computer games to virtual/augmented reality and is recently being explored in other fields. Gestures are a type of nonverbal communication, where visible body movements take place and deliver certain messages along with speech. Gestures can be expressed by suitable body movements of face, hands, head which can be used for controlling specific devices or transferring significant knowledge to the environment. Human gestures can originate from any body movements. Gestures can be of static type (certain pose or configuration, or still body posture) or dynamic type (movements of different body parts).

The human being, communicates with the machines via direct contact mechanism. Nowadays, instead of pressing switches, touching monitor screens, twisting knobs, raising voices, work can be simply done by pointing fingers, waving hands, movements of bodies and so on. Gesture recognition concerns about recognizing meaningful expressions of human motions, as for example embroiling hands, arms, face, head and body. It is an interesting topic in computer vision and pattern recognition technology which deals with the mathematical interpretation of human gestures via a computing device. It enables human being to interact with machines without any mechanical contact, as for example by signaling a finger on a computer screen such that the cursor would move accordingly, which makes such conventional input devices like mouse, keyboard obsolete. Thus, after presenting how a system can work, we will implement it in our project.

2.6.3 Gesture Recognition with Kinect Depth Sensor - From GUI to NUI

The last fifty years have brought a vast number of technological accomplishments and progress like no other period in human history before. All of that wouldn't be possible without transistors and semiconductors. Since then, the development of computers and computer technology has been increasing exponentially and reached scales that were possible only with the science fiction writers from the beginning of last century. Modern Hollywood blockbusters have in some way influenced the designers and engineers of modern computers and user interfaces. One of them is Minority report by Steven Spielberg from 2002. In this movie, the protagonist played by Tom Cruise controls the computer with gestures. A few years later, the vision of this movie's director came true and the device that enables the control of computer by gestures has been introduced. In this case, it was the game console Nintendo Wii, with Wii Remote controller, publicly shown in 2006. This controller was able to track the position of the player and control the virtual character by user's gestures. Its presence was the turning point in the development of this type of devices. Later on, in 2010, Microsoft has announced Microsoft Kinect, supplement to the XBOX 360 console. Its popularity is based on the possibility of connection to the computer, thanks to the published drivers and Software Development Kit (SDK)[81]. In this way, it is possible to use it in the areas that are not strictly related to the entertainment industry. Since the devices that Kinect belongs to can be classified as gesture tracking devices, this type of user interface is known as NUI (Natural User Interface).

The shift from GUI to NUI (natural-user interface) technologies has transformed what a computing environment is. Today, computing environments are increasingly immersive experiences in which it is difficult to distinguish between a user and a system, which means that the ways in which we achieve a critical and reflective stance is changing. It is important to note that some NUI technologies have been around for several decades, but it's the relatively recent explosion of interest in and profitability from products(ex. [82]) including Apple's iPhone and iPad and Microsoft's Kinect that has led to the uptrend in positive discourse and outright evangelism about them.

2.6.4 Kinect's Technology

Kinect is a sensor created and first published by Microsoft as a side gaming devise for Xbox360 gaming console.(see Figure:19 below) Its category named RGB-D (Red-Green-Blue-Depth) sensor. It combines several different devices working together. These are an RGB camera, a depth sensor, four microphones and a motorized tilt. The RGB camera provides a colorful image and it is used as a simple RGB camera. The difference with Kinect, is the depth sensor that provides depth information for many of the items capture in 2D camera plane. The depth sensor consists of an infrared projector and an infrared camera. The infrared camera captures the reflections of the infrared waveform and provides information for the distance of the reflected items, if they are in capturing range.[83] Together the RGB camera and the depth sensor are a powerful tool in Computer Vision science. The RGB image provides information for chromatic characteristics and for the template of an item. Combined with the distance, given from the depth sensor, it can provide the width and the height of the item.[84],[85] The output of the depth sensor is a gray-scale image, called depth map. It is same size with the RGB image and each pixel scale represents the distance from the camera. The darker points are closer to the camera. If a point is out of the range and the depth cannot be measured, the pixel in this point is black. The range that the sensor gives trusty depth information depends to the operation mode and it is 0.8 to 4 meters for the default mode and 0.4 to 3 meters for the near mode. Objects in greater distance cannot provide appropriate infrared reflections for measurement and nearest objects are not in the plane of view of the two cones of both infrared projector and infrared camera.

The gesture tracking is the based on series of observations of human activities and environmental conditions in order to achieve main goals of the recognition process. The gesture recognition requires sensor that generate and receive signals. The software that can interpret sensor readings is required as well. The gesture recognition is based on past events. Through the learning process, the software can predict future positions. The goal of gesture tracking is the detection and observation of objects in motion via the sequence of recorded images. When multiple images are recorded using camera, the first step is to differentiate moving objects from static background. The working principle is following:

- The projector emits known pattern of infrared (IR) light to the surrounding objects.
- The sensor observes the scene and detects changes in the projected pattern, which depend on the distance and shape of an object.
- The sensor sends received data to the control logic (computer system) for further processing.
- The received data are processed and 3D map of object is formed (3D map in this context represent the collection of 3D coordinates that form the surface of observed object).

We have to mention that, before starting to recognize the gestures, a digital treatment phase has been taken place. This pre-processing part of the workflow is the process where an image is taken and a modified version of it is produced in order to obtain specific information from it. The treatment that the image has to go through consist of 5 stages[86],[87],[88] ,[89]:

1. Image acquisition: An image of the object of interest is obtained.
2. Restoration: Digital filters are applied, in order to remove noise, increase the image contrast or brightness, etc. in order to improve the visualization of the obtained image.
3. Segmentation: The object of interest is separated from the background and from the rest of the objects.
4. Description: Describe the object of interest through the measurement of certain attributes of the object. The parameters or measures that are extracted represent and characterize the object.

5. Recognition: To classify objects from the previous descriptors pre-defined. This means, to assign each object to a class which is a family of shapes (associated to objects) that share common characteristics. For example, the shape of a human body or the shape of a closed hand.

6. Interpretation: According to the values obtained in the measurement phase, is carried out an interpretation of the gesture.

Figure 19: Kinect Depth Sensor

Brief History of Kinect: With the SDK for the Kinect for Windows, Microsoft has provided an easy manner for any developer to create their own applications using the Kinect. After the release of the Kinect for Xbox, the device became very popular with application developers and educational institutions to experiment with. After this became apparent to Microsoft, they released a first beta version of their SDK to support these people in June 2011 and a second beta version in November 2011. Both versions were for non-commercial use only, but in February 2012 a commercial version (v1.0) was released. Starting with v1.0 it was now possible for developers to create applications using the Kinect for Windows that could be deployed for commercial use. Microsoft has eventually released four major updates for their SDK plus one further release.

1. v1.5, May 2012 – Introduced Kinect Studio, Near Mode, Seated Tracking and Face Tracking.

2. v1.6, October 2012 – Introduced a range of minor improvements and support for Windows 8 and Visual Studio 2012.

3. v1.7, March 2013 – Introduced Kinect Fusion. This essentially turned the Kinect into a 3D scanner and made it possible to create 3D models.

4. v1.8, September 2013 – Introduced the features to easily remove the background from scenes and access to the Kinect data through the web.

5. v2, July 2014 – Further Improvements we will discuss in the sections below.

Features of Kinect : The Kinect for Windows SDK comes with a wide array of features and tools to be used for the development of Kinect enabled applications. A list and short summary of the most notable features and tools are described in this subsection.

·*Near Mode*

The Near Mode feature is a Kinect for Windows specific feature, which sets it apart from the Kinect for Xbox since it is built into the firmware of the sensor. This features enables the Kinect for Windows to track the human body from a very close range of about 40 centimeters. Near Mode is a setting that can be turned on or off within the code of the application.

·*Capturing the infrared stream*

Capturing images from the infrared stream opens up the possibility to use images in low light conditions.

The images captured from this stream are 16-bits per pixel infrared data at a resolution of 640x480. The frames come in with a speed of 30 FPS. Since the infrared image is captured as part of the color image, it is not possible to capture both at the same time.

·*Tracking human skeleton and joint movements*

The tracking of the human body is one of the most interesting parts of the Kinect. Up to 20 joints can be tracked on a single skeleton and up to six persons can be tracked at the same time. Due to resource limitations however, only the joints of 2 skeletons can be returned. If the application only requires the joints of the upper body, it is also possible to track a person in seated mode, which only returns 10 joints. This method saves resources which can either be used elsewhere or left alone to make the application run better on lighter machines.

·*The audio stream and speech recognition*

With the four microphones in the sensor, the Kinect is capable of capturing raw audio, but the SDK also makes it possible to clean up the audio signal by applying features such as noise suppression and echo cancellation. It is also capable of speech recognition for use in voice commands and it can detect the direction and distance of the source of the sound.

·*Human gesture recognition*

Although the SDK has no direct support for an API for human gestures, the availability of skeleton tracking and depth data opens of the option of creating gestures which can be used as input method for any Kinect application.

·*Tilting the Kinect sensor and using the accelerometer*

The SDK gives direct access to the motor of the sensor. With this the developer can change the elevation angle of the sensor. The elevation angle range lies between +27 degrees and -27 degrees. The motor only tilts the sensor vertically. The elevation of the sensor is also used to determine the orientation of the sensor by the accelerometer. The data from the accelerometer can be accessed as well.

·*Controlling the infrared emitter*

The infrared emitter can be turned on or off. This might seem like an insignificant feature, but it is useful when using multiple sensors since the infrared emitters from the different sensors interfere with each other. This feature is also only available when using the Kinect for Windows and not the Kinect for Xbox.

·*The Kinect for Windows Developer Toolkit*

The Kinect for Windows Developer Toolkit is a collection of sample applications and documentation to help the developer to create sophisticated applications quickly.

·*The Face Tracking SDK*

By using the Face Tracking SDK applications can be created to track the positions and orientations of faces. It is also possible to animate brow positions and the shape of the mouth in real time which can be useful for detecting facial expressions for example.

·*Kinect Studio*

Kinect studio is able to record and playback the data stream of the sensor. This is very useful during the testing of applications. With this tool the developer does not have to perform in front of the camera him or herself every time the application needs to be tested. Instead, a recording can be used to playback with the application. The recorded file contains all the data that can be captured by the sensor.

Brief Description of Kinect's Functionality: Using the Kinect SDK it is possible to program in the languages Visual Basic (VB) .NET, C# and C++. With VB .NET and C#, programmers can use the managed code provided by the SDK which comes in the form of a Dynamic Link Library (DLL) called Microsoft.Kinect.dll. By adding this library to their application the programmers get access to all the features of the Kinect sensor. In this case programming means that the programmer can only use the unmanaged code provided by the SDK but it gets the programmer closer to the source code since the C++ accesses the Native Kinect Runtime APIs directly, whereas VB .NET and C# code first needs to be handled by the

.NET Kinect Runtime APIs.

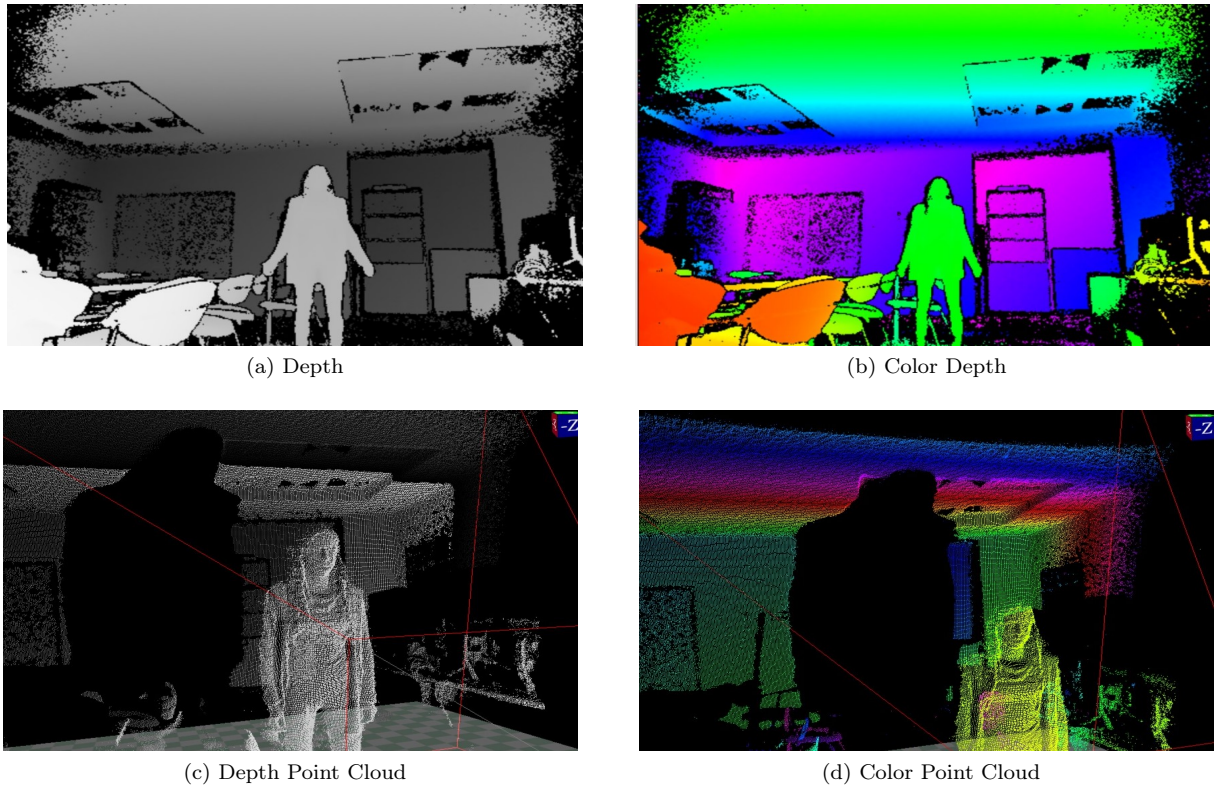
Access to the sensor is usually required to obtain sensor data. There are three types of data that are streamed from the sensor to the application. These are the color data stream, the depth data stream and the audio data stream. These data streams are directly available from the sensor through its drivers. The Kinect for Windows SDK makes it possible to access two more data streams, which are the infrared data stream, which gets subtracted from the color data stream, and the accelerometer data from the sensor. Thanks to Kinect's provided libraries programmers can easy get access to the following features: capturing and processing the color image data stream, processing the depth image data stream, capturing the infrared stream, tracking human skeleton and joint movements, human gesture recognition, capturing the audio stream, enabling speech recognition, adjusting the Kinect sensor angle, getting data from the accelerometer and controlling the infrared emitter.

·*Capturing and Processing Color Image Data Stream*:
The color image date stream is one of the two types of image stream formats that are supported by the SDK, the other one being the depth image stream. Both streams are transferred through different pipelines. All image frames from the image data stream are stored in a buffer in a first in last out fashion. The application makes use of one or more image frames by retrieving them from this buffer. The FPS speed depends on how fast the application can retrieve image frames from the buffer. This buffer is constantly refreshed with new images frames however. So if the application is not able to retrieve image frames in time, the user will see a stuttering video. Kinect camera is capable of capturing 32-bit RGB images at the resolutions 640 x 480 and 1280x960, where it supports 30 FPS at 640 x 480 and 12 FPS at 1280 x 960 (see Figure: 20 below). Besides the RGB images the SDK also makes it possible to retrieve 16-bit YUV images, raw Bayer images and 16-bit infrared data over the same pipeline. The YUV images are only available with a resolution of 640 x 480 at 15 FPS, while the Bayer images offer the same two formats as the RGB images. The infrared data is only available with a resolution of 640 x 480 at 30 FPS. An important note to remember is that only one image type and format can be requested at a time because all of these image frames are transported over the same channel. Actually retrieving images within an application can be done in two ways, either by using the event model or the polling model. When using the event model we need to subscribe to the specific event handler where we process the incoming frames within our code. When subscribing we must specify the image type and format we would like to receive. Once this setup is complete the Kinect sensor will keep streaming image frames to the application until it gets stoped. When using the polling model a request is sent to the Kinect SDK every time an image frame is required. This is handled by the SDK by first looking whether or not there is an open color channel and if no one has subscribed to that channel. If that is the case, it will automatically pick an image frame from that channel. If there are no open channels, a new channel will be created to retrieve the image frame. Below (Figure: 20) we can see a captured image of the depth and color stream:

(a) Depth (b) Color

Figure 20: Kinect Studio

·*Capturing and Processing Depth Image Data Stream*:
The depth data the Kinect sensor returns consists of a 16-bit gray scale image. The Kinect sensor throws a series of algorithms at this image which will return the distance of each pixel in millimeters. The depth data stream can return images with 30 FPS at resolutions: 640 x 480, 320 x 240 and 80 x 60. The image with the depth data is created by the IR emitter and IR depth sensor of the Kinect. In short, the IR emitter constantly emits infrared light in a pseudorandom dot pattern. The IR sensor reads these dots and with the data being received a depth image is constructed. The method used to determine the distances in a depth data image is called stereo triangulation. This is an analysis algorithm for computing the 3D position of points in an image. With stereo triangulation, two images from two different views of the same scene are used. The relative depth information is then calculated by comparing these two images. This is about the same way humans are able to perceive depth and why this is difficult for people with only one eye. With Kinect this method is applied to the image generated by the IR sensor, which is the image that is actually captured, and an 'invisible' second image. This second image is actually a pattern of the IR emitter. Since the emitter and the sensor have some distance between them, their viewpoint of the scene is slightly different, making it possible to apply the stereo triangulation algorithm. One limitation to keep in mind concerning this feature, is that it only works optimally between 80 centimeter and 400 centimeter, or 2,6 feet and 13,1 feet, not taking near mode into consideration. The Kinect is able to capture beyond 400 centimeter but the quality of the resulting depth image frame suffers dramatically. After a depth image frame has been created a method called bit-shifting is used to calculate the distance from each pixel in the image frame. Since each depth image frame is a 16-bit gray scale image, each pixel contains 16-bit worth of data. Of these 16-bits, the first three are used to represent the player(s) identified by the Kinect. The other 13 bits represent the distance in millimeters. By performing a right shift of three bits the distance of that particular pixel can be calculated. Distance is measured for every pixel in a 2D addressable array, resulting in a depth map which is a collection of 3D points (each point also known as a voxel). 2D representation of a depth map is a gray-scale image, where the brighter the intensity, the closer the voxel. Alternatively, a depth map can be rendered in a three-dimensional space as a collection of points, or point-cloud. The 3D points can be mathematically connected to form a Mesh onto which a texture surface can be mapped. If the texture is from a real-time color image of the same subject, a life-like 3D rendering of the subject will emerge. One may be able to rotate the image to view different perspectives. Below (Figure: 24) we can see captured images from the depth stream, presenting depth, depth with color, point cloud gray and color point cloud in the same sequence.



(a) Depth

(b) Color Depth

(c) Depth Point Cloud

(d) Color Point Cloud

Figure 21: Capturing Images with Kinect Depth Sensor

Besides the distance, the player index can also be calculated with the first three bits. In order capture this, the skeleton stream needs to be enabled since it contains the player information, which will be discussed further on. The player index can have a value from 0 to 6 where the value 0 states that the sensor does not recognize any player. The player index value is calculated by performing a logical AND operation with the value of the pixel and a constant called the PlayerIndexBitmask. This constant is defined in the class DepthImageStream and always has the value 7. The figure below (Figure:22) demonstrate the body index:

Figure 22: Capturing the Body Index with Kinect

·*Tracking Human Skeleton and Joints' Movements*:

Tracking a person with Kinect sensor is achieved by using the raw depth data. Initially the person that is standing in front of the sensor is just another object on the depth image frame. By using the human pose recognition algorithm an attempt is made to recognize the person from the other objects in the depth image frame. The human pose recognition algorithm uses several base character models that vary from one another on several factors such as height, size, clothes, etc. From these base characters machine-learned data is collected, where each individual body part is identified and labeled. These are then matched with the depth image frame based on probabilities. Initially the human body is just one of the many objects in the space the sensor sees. In order to identify this object as a human body the sensor matches each individual pixel of the depth data with the data it has learned. Of course the match is seldom perfect, thus a successful match is based on probability. When a match has been found the process of identifying the human body immediately continues, by creating segments to label to the body parts. This segmentation is done by using a Decision Forest, which consists of a collection of independently trained decision trees. With this method the data is matched to specific body types. Every single pixel is passed through this forest, where all the nodes represent different model character data labeled with body part names. When a match has been found, the sensor marks the pixel to eventually divide the body into separate segments. When this process is finished the sensor positions the joints on the places that have the highest probability of matching with the referenced data. The coordinates (X, Y and Z) of each of the joints are calculated by three views of the same image namely, the front view, the left view and the top view. This is how the sensor defines the 3D body proposal. The Kinect SDK gives the developer access to all 20 skeleton joints. Each joints is identified by its name (head, shoulders, elbows, etc.) and during tracking are each in one of their three states; Tracked, Not Tracked or Inferred. The skeleton as a whole can achieve three states: Not Tracked, Position Only or Tracked. Figure:23 below shows a complete skeleton with all 20 joints visible.

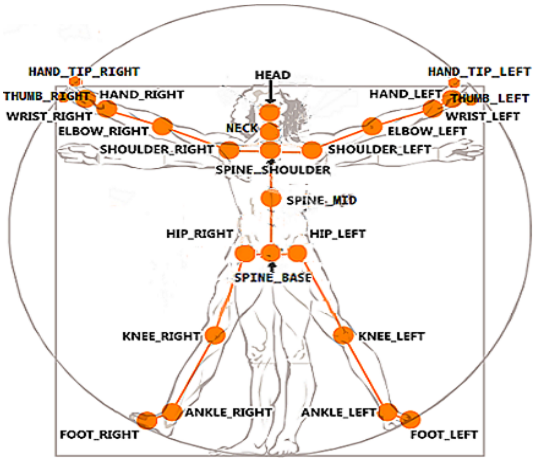
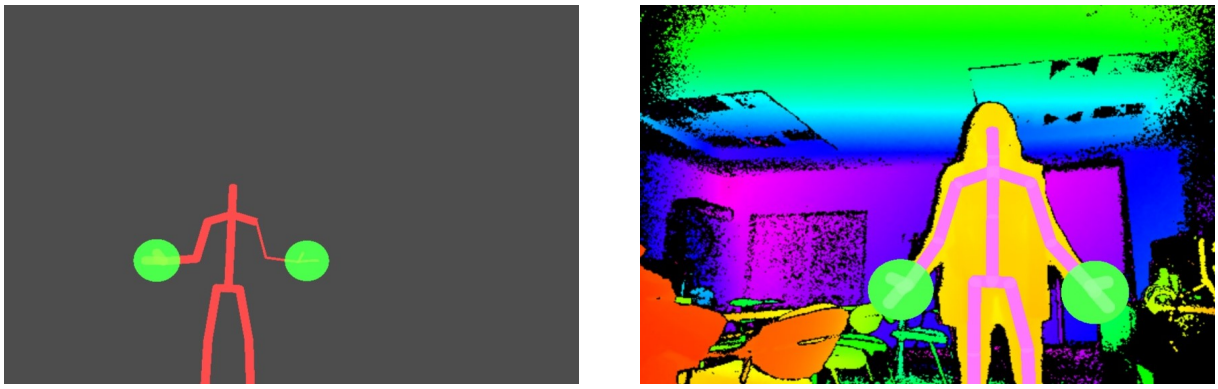


Figure 23: Body Joints as seen from Kinect

The sensor is able to detect a maximum of six users. It is also possible to track a seated skeleton, instead of the default full skeleton. This mode only tracks and returns 10 joints points. If the developer only needs to track these joints, using the seated skeleton mode can save a lot of unnecessary data being sent from the sensor to the application. When tracking a user, the data of the skeleton is returned to the application in a SkeletonStream object. The joints in the skeleton are organized hierarchically. This means that every joint can be a parent and a child unless the joint is a leaf such as the head and hands. The highest joint in the hierarchy is called the root joint and every skeleton only has one of these. With the full, default skeleton the root joint is thee Hip Center joint. For the seated skeleton this is the Shoulder Center joint. Below we can

see a capture of the body stream:



(a) Body with joints

(b) Body with Color Depth

Figure 24: Tracking Human Skeleton with Kinect

·*Recording Audio*:
Additionally, Kinect provides also an audio stream implemented for voice recognition,which however will not be our main concern, as seen in the figure below:

Figure 25: Recording Audio with Kinect

The Kinect v2: On July 15th 2014, Microsoft began shipping their new Kinect sensor, simply named Kinect v2. They also released the public preview version of the Kinect SDK 2.0. The new Kinect has been improved on many fronts compared to the first Kinect. Below we can see a figure demonstrating Kinect v2 Depth Sensor:

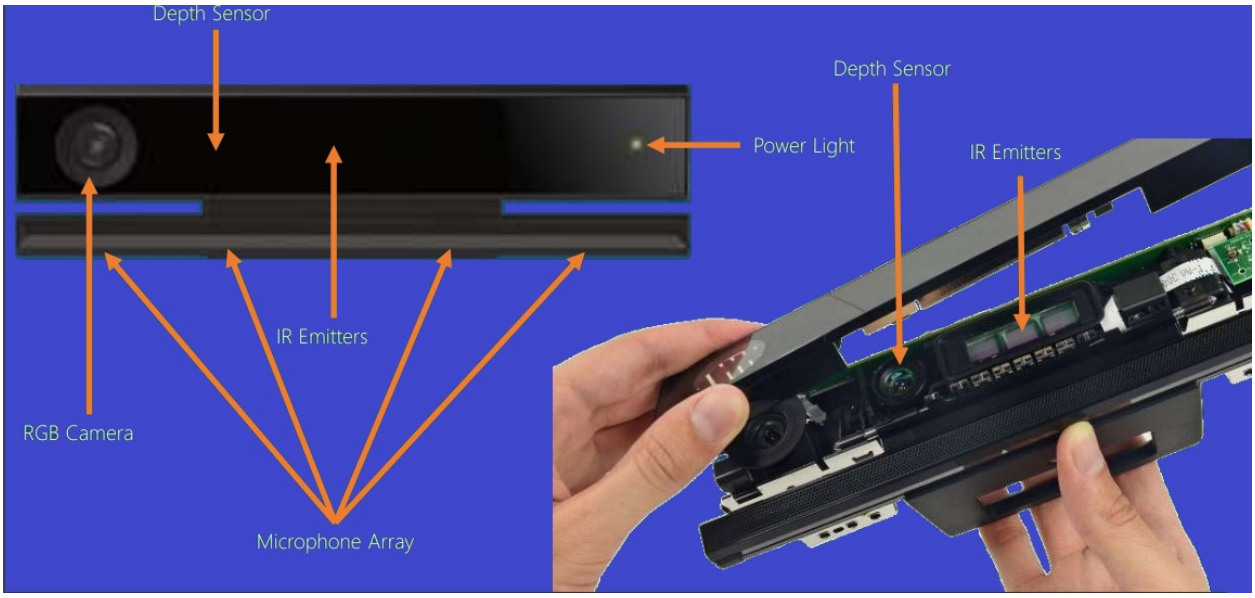


Figure 26: Kinect's v2 Hardware

The Hardware: The color camera of the Kinect v2 now supports a resolution of 1080p, running between 30 FPS and 15 FPS depending on the lighting conditions. The depth sensor has an increased resolution as well, from 320x240 with the v1 to 512x424 with the v2. This has been achieved with what Microsoft calls Time of Flight. Where the v1 used an infrared grid pattern for depth sensing, the new infrared beam sensor is modulated and measures how long it takes for photons to travel to any object in its view. Additionally this new infrared technology ensures that the Kinect v2 sensor does not depend on the lighting conditions of the environment it is used in. The infrared image stream runs at 30 FPS, as well as, the range and field of view have also been improved. The field of view has been expanded in height and in width, which in turn improved the range as well. The Kinect can now track between the 0,5 meters to 4,5 meters. The depth sensor can even measure up to 8 meters. Concerning skeleton tracking, the Kinect v2 can track 6 full skeletons simultaneously, where each skeleton consists of 25 joints. A few of these extra joints were placed in the hands, which makes it possible to track the thumb and four more fingers, as two separate joints and the ability to discern between an open and closed hand. The skeleton has been corrected anatomically, particularly where the hips, shoulders and spine were concerned. These joints have been previously located too high in the body. Additionally, in order to connect the new sensor to a laptop, the Kinect Adapter for Windows is necessary, displayed on the image below:

Figure 27: Kinect's adapter

The Software: On the programming side, it is now easier to develop applications for the Kinect v2, because it now takes less lines of code to initialize functions of the Kinect sensor, or retrieving data from stream objects such as the skeleton or color stream. The SDK has also been changed in a way, such it is now possible to develop a Kinect application and compile it as a desktop application, Windows Store application or Xbox One application. Out of the box the new SDK also includes a Unity plugin to develop Kinect enabled games in the Unity 3D(this plugin we will also use for our implementation).[90] Microsoft also delivers a tool called Kinect Common Bridge. Kinect Common Bridge is essentially a toolkit that enables developers to access the Kinect API from within other programming environments. This could be an application, such as MATLab. The Kinect is now able to recognize faces accurately and track its position and orientation. Out of the box it can also track mouth and eye movement. The second major feature concerning the face, is an extension of the Fusion feature that was introduced in the SDK v1.7 and is called HD Face. This feature allows us to scan the face in very high resolution, providing the possibility to create detailed 3D models of people. The tool-set of Kinect has been improved and extended as well. The new Kinect Studio has an improved UI, which can be organized to an individual's own taste. It also provides better features, as means to choose which data types we want to record, making it easier to just record the data we need, in addition to saving bandwidth at the same time. When testing Kinect v2 applications by playing back recordings with Kinect Studio, having a Kinect sensor attached to the computer is not required anymore. This makes it easier to develop applications anywhere without having to bring along sensor as well. A new tool has been added as well, called Visual Gesture Builder. Utilizing this tool, it becomes easier to create and train complex gestures. These gestures can then be used to interact with the programmer's application. We will be using this tool, to implement Gesture Recognition for out tool. For further comparison between the previous version of Kinect and Kinect v2, we recommend the readers to study this research [91]

2.6.5 Methods for Gesture Recognition

We will be using Visual Gesture Builder, to implement Gesture Recognition for our tool, but we need to specify the two types of methods, that can be used for Gesture Recognition.

- **Heuristic-based Gesture Detection**: Heuristic-based Gesture Detection is, basically, to shape an algorithm that is customized to describe a certain gesture. That algorithm is designed to recognize a single gesture or a single class of gestures relying on the context in which the gesture is performed using the skills and experience of a human to build that algorithm.[92]
  - Benefits of using heuristics for gesture detection:

1. Less expensive than machine learning. It does not relay on external code basis or external libraries.
2. No infrastructure costs.
3. Helps reconstruct missing information.
4. Reduces the expected states, or frames, of the gesture from the user's movements during detection and comparing them heuristically to expected values from a predetermined template.

□ Disadvantages of heuristics for gesture recognition:

1. Difficulty in describing algorithmically a gesture (Requires high programming skills).
2. Takes longer time to figure out an efficient algorithm that describes with high accuracy a complex gesture.
3. Heuristics may produce results by themselves so they have to be commonly used in conjunction with other optimization algorithms to improve their efficiency.

■ **Machine Learning for Gesture Detection**: Machine learning is, in brief, an algorithm developed to note changes in data and evolve in its design to accommodate the new findings, using techniques that identify patterns within given data-sets, based on a likelihood matching with pre-existing patterns. This signifies that, a machine learning approach involves the defining of gestures, as an ordered set of gesture states, based on a general expectation of what that state (gesture segment or pose) looks like mathematically in 3D space [93].

□ Benefits of using machine learning for gesture detection:

1. Sophisticated pattern recognition. Along with noting relationships, can determine the type and quantify as well. This is not just happening with key, or even secondary variables, but on every relationship that takes part in the pattern.
2. Intelligent decisions along with the capability to note irrelevant data, and rank the relative importance of variables. Machine learning methods can make decisions either aided by a human or not. The solution can distinguish sub-classes and make determinations on what data should be included and which should not with very little instruction.
3. Time saving since the total amount of time spent in training and testing the examples is much lower than trying to figure out a heuristic description method.
4. Gesture detection is created and handled as content, rather than being a time-consuming coding task.
5. Gestures can be quickly prototyped and evaluated before any code is written.
6. High accuracy for detecting gestures can be achieved—even in cases where skeletal data is very noisy, such as sideways poses.
7. By tagging data appropriately, perceived latency can be made very low.
8. The database size is independent of the amount of training data.
9. There are very few thresholds to tweak and maintain.

□ Disadvantages of machine learning for gesture recognition:

1. As the model becomes more refined as much examples are given, there is needed to record the same gesture several times in different situations, speed, people, background, etc. The test of multiple iterations will produce a final version that delivers the highest level of accuracy.
2. The recorded gesture would be specific to the actor who recorded it and only those with a similar frame would successfully have the gesture detected.
3. It is time consuming to tag data.
4. A powerful PC is required for training sessions.
5. Much disk space is required for storing raw (.xrf) and processed (.xef) recordings.

2.6.6 Gesture Recognition using Visual Gesture Builder

As camera and sensor technology evolved, many HCI(Human ComputerInteraction) technology that appeared in multiple science fiction movies in the past have emerged. Related research in gesture recognition has been studied through the DTW (Dynamic Time Warping), and HMM (Hidden Markov Model) algorithm with temporal and spatial variations. Study of 3D gesture recognition using the HMM is increasing, however we will focus on another ML Algorithm, the Adaptive Boosting Algorithm, which is integrated in Kinect SDK for building gesture databases that are recognized by the sensor in a semi-automatic way.

Visual Gesture Builder is a tool included in Kinect v2 for Windows SDK, that uses Machine Learning Algorithms to create Gesture Databases for Kinect Applications. These databases are used to perform gesture detection at Runtime using the VGB API. VGB uses the AdaBoost(Adaptive Boosting) Algorithm to determine when a user performs a certain gesture and RFRProgress(Random Forest Regression Progress) Algorithm to determine a gesture's progress. These gestures are captured with Kinect Studio and fed into VGB.

VGB "learns" gestures by taking in clips and having the developer label the portions of the clips where the desired gesture is visible. On the flip side, the untagged parts are used as negative footage to indicate what does not constitute a gesture. Finally, the developer runs an existing Machine Learning (AdaBoost) algorithm to synthesize gesture recognition code that can be included in a program.

Adaptive Boosting Algorithm

After it sequentially generates classification rules, the AdaBoost algorithm readjusts the distribution of sample data from the observed values obtained, by applying the previous classification rules. Weight of the sample data will start in the same state as early learning. As each round progresses, the misclassified data is given a high weighting from observations obtained, by applying the previous classification rules. In contrast, distribution of the sample data is re-balanced in a way that gives low weights the correct classification data[3]. Pseudo-code of the Ada-Boost algorithm is given in in Figure:28. The algorithm takes as input a training set $\left(x_1, y_1, ..., x_m, y_m\right)$, where each x_i belongs to some domain or instance space X, and each label y_i is in some label set Y. We assume Y = $\{$-1, +1$\}$; Adaboost calls a given weak or base learning algorithm repeatedly, in a series of rounds t = 1, ..., T. The weight of this distribution on training example i on round t is denoted D_t(i). Initially, all weights are set equally, but on each round, the weights of incorrectly classified examples are increased, so that the weak learner is forced to focus on the hard examples in the training set.
The weak learner's job is to find a weak hypothesis h_t:

$$X \to \{-1, +1\}$$

appropriate for the distribution D_t. The goodness of a weak hypothesis is measured by its error:

$$\epsilon_t = \Pr_{i \sim D_t}\left[h_t(x_i) \neq y_i\right] = \sum_{i:h_t(x_i) \neq y_i} D_t(i).$$

Notice that the error is measured with respect to the Distribution D_t, on which the weak learner was trained. In practice, the weak learner may be an algorithm that can use the weights D_t on the training examples. Alternatively, when this is not possible, a subset of the training examples can be sampled according to D_t, and these (unweightened) resampled examples can be used to train the weak learner.

Given: $(x_1, y_1), \ldots, (x_m, y_m)$ where $x_i \in X$, $y_i \in Y = \{-1, +1\}$
Initialize $D_1(i) = 1/m$.
For $t = 1, \ldots, T$:

- Train weak learner using distribution D_t.
- Get weak hypothesis $h_t : X \to \{-1, +1\}$ with error

$$\epsilon_t = \Pr_{i \sim D_t} [h_t(x_i) \neq y_i].$$

- Choose $\alpha_t = \frac{1}{2} \ln \left(\frac{1 - \epsilon_t}{\epsilon_t} \right)$.
- Update:

$$\begin{aligned} D_{t+1}(i) &= \frac{D_t(i)}{Z_t} \times \begin{cases} e^{-\alpha_t} & \text{if } h_t(x_i) = y_i \\ e^{\alpha_t} & \text{if } h_t(x_i) \neq y_i \end{cases} \\ &= \frac{D_t(i) \exp(-\alpha_t y_i h_t(x_i))}{Z_t} \end{aligned}$$

where Z_t is a normalization factor (chosen so that D_{t+1} will be a distribution).

Output the final hypothesis:

$$H(x) = \text{sign} \left(\sum_{t=1}^{T} \alpha_t h_t(x) \right).$$

Figure 28: Adaptive Boosting Algorithm - Pseudocode

Once the weak hypothesis h_t has been received, AdaBoost chooses a parameter a_t, as in the Figure above. Intuitively, a_t measures the importance that is assigned to the h_t. Note that $a_t \geq 0$, if $e_t \leq 1/2$ and that a_t gets larger as e_t gets smaller.
The Distribution D_t gets updated using the rule shown in the figure. The effect of this rule is to increase the weight of examples misclassified by ht and to decrease the weight of correctly classified examples. Thus, the weight tends to concentrate on hard examples.
the final hypothesis H, is a weightened majority vote of the T weak hypothesis, where at is the weight assigned to h_t.
For more information about the algorithm and the training error analysis, we recommend the readers to recur the references.[94], [95], [96]

As a result, VGB is an interface provided from Kinect SDK, with AdaBoost algorithm integrated in its building process for gesture detection. Once the trainer loads and labels the clips, the system analyzes the inputs given from the trainer (for example, which joints are significant for a specific gesture and which to ignore). Afterwards, the algorithm generates the weak classifiers for each characteristic (for example joint positions x-y-z, or velocity, angle, muscle force etc) of a specific gesture and weights them. According to the weights, a strong classifier is being built, after having evaluated the data of the weak classifiers and following the pseudocode we described above. We will analyze the functionality of VGB in chapter 5.3

2.6.7 Applications

Gesture recognition technology has been employed for several domains, like hand gesture, dynamic hand gesture tracking, posture recognition, sign language recognition, robot control, person identification, healthcare etc. Lai et al.[97] in their approach, utilize a Kinect sensor to develop a skeleton,using data acquired from human postures. They developed two different methods to leverage Kinect; one by calculating vectors

in the space of features using the Euclidean distance metric and the second, while the other method does this in the space of feature covariances using a log-Euclidean metric. Wang et al.[98] proposed a Kinect based dynamic gesture trajectory identification method, using OpenNI, where a palm is defined as a Kinect node, to get the palm position coordinates, and then use the Hidden Markov Model technology to do gesture model training in order to optimize the recognition rate. Le et al.[99] have proposed a human posture recognition method, which exploits skeleton data, to define an interested posture with high accuracy. Support vector machine is applied to classify the human postures from the extracted features.

Oszust and Wysocki[100] have developed a system for signed expressions identification, using Kinect. Skeletal structures of the body and hand positions are used to characterize Polish sign language words, based on seven features for each hand, that have been analyzed using k- nearest neighbor algorithm. Malima et al.[101] have designed an algorithm for fast hand gesture recognition through vision, where robots are controlled by hand pose signs provided by human being captured through a camera. Sadhu et al.[102] have designed a highly accurate system for person identification using their gait posture by Kinect sensor. Features from nine body joint movements of 25 different people have been extracted with respect, followed by coordinate normalization. Parajuli et al.[103] have developed an senior health monitoring system using the Kinect device to monitor elderly people. Support vector machine is used to analyze the gait and posture data, in order to detect when they are likely to fall by measuring their gait.

2.6.8 Summary

The embodiment of computers in places, such as work places, educational institutes or entertainment venues, has had a tremendous impact on the field of user-system interaction. We spend huge amounts of time and energy to transfer information between those two worlds. Mice and graphic displays are everywhere, in order to define the frontier between the computer world and the real world. This could be reduced by better integrating the virtual world of the computer with the real world of the user, providing User Interfaces based on gestures for example, instead of hardware inputs. In this subsection, we discussed about the importance of user friendly front-end systems and how gestures could provide appealing human-computer interactions. Furthermore, we discussed about the implementation of such interactions with Kinect Depth Sensor, as well as present the sensor's technology and the methods that achieve gesture recognition. In addition to this, we summarized several state-of-the-art applications, that apply gesture-based technology.

2.7 Interactive Projection Mapping

2.7.1 Introduction

We humans have been fascinated with creating art that fools our eyes and transports us physically since the beginning of civilization. From perspective tricks in murals, to the fake windows on blank building walls of the Trompe l'oeil tradition, we delight in deceiving our senses. Now, in the 21st century, 2D and 3D Projection Mapping are the latest tools of this grand tradition, and are available to transform events in ways that are only limited by imagination. Projection Mapping isn't around for that long, in the form that we know it today it has maybe existed for 10-15 years. It is only in the last 5 years that the technique is greatly hyped and gets used a lot for a great range of different kind of projects. These projects can be somewhat grouped in to four main categories. 1) Commercial product presentations 2) Live music visuals 3) Festivals 4) Abstract artistic projects. It started out as an art form, but as with many things in the creative world, if it is successful in the non-commercial field it won't take long until it gets imported into the commercial projects. It can be seen on buildings, vehicles, festivals, live music performances, clothing and just about anything we can think of. It is a technique that is in its core not that difficult, but because the execution differs extremely within each project, it doesn't get dull quickly.

In this subsection, we will discuss the great impact Projection Mapping, after we apprehend the human visual perception of light. We will further discuss how Projection Mapping can be enhanced when we add the interactivity concept, as well as summarize the latest applications that take advantage of Interactive Projection Mapping, in order to lure, captivate and cultivate the viewers.

2.7.2 Influence of Projection Mapping

Driven by a new, affordable generation of projectors, mapping can completely cover not just flat walls and traditional projection screens but also irregular shapes, objects, and even entire building facades. These projectors are not typical boardroom AV devices. Among the many capabilities of Projection Mapping nowadays are:

•Affordable, high definition projection of images, including video and animation.

•Synchronization of projected images with high-quality sound.

•Portable placement of projectors.

•tacking and linear linking of projectors for panoramic and large-scale, seamlessly blended images.

•Innovative and easy-to-use software for creating and driving original Projection Mapping immersions.

For nightclub owners, sports producers, and corporate/private event managers, Projection Mapping draws customers and entertains clients. For this reason, many Las Vegas nightclubs invest heavily in AV technology, including Projection Mapping, to pack their dance floors. Now, with the latest generation of liquid crystal display (3LCD) projectors and off-the-shelf software, most clubs and other venues can deliver electrifying motion, immersion, and action into virtually any setting. Projection Mapping now permeates several public events—from product launches to concert halls. Wherever producers want to bring excitement to the next level, Projection Mapping plays a key role in emotionally engaging audiences, increasing the "wow" factor, and promoting greater attendance and attention.

Light Shapes Human Emotions

Neurological research has shown a direct correlation to light and hormonal activity in the human brain.[104] Lighting, or lack of it, can actually trigger hormonal and other electrical impulses. Seasonal Affective Disorder (SAD) is perhaps the best known direct correlation to the absence of natural sunlight and emotional response[105]. Patients with SAD exhibit prolonged depression during the long, cloudy periods of winter. More recent studies have found that Red diffuse light suppresses the accelerated perception of fear[106]. Moreover, blue light, especially the type emanated from electronic devices, can ward off sleep.[107] The pineal gland in the brain normally releases melatonin, the brain's "sleeping pill," a few hours before a person's regular sleep pattern. Exposure to blue light, however, postpones or diminishes the brain's production of melanin. However, other studies have shown that blue-enriched white light in the workplace improves self-reported alertness, performance and sleep quality[108].

Light Excites

Meanwhile, studies at the University of Toronto [109] have determined that bright lights intensify emotions. Participants in the study were exposed to various light levels when judging a variety of sensations, such as taste, word choices, and fictional characters. High light levels correlated to heightened responses. Researchers concluded, that marketers should make certain, that light levels are high when literally spotlighting potentially emotional product purchases, such as jewelry. Dim light is relaxing, the perfect venue for a romantic

dinner. Bright light excites the brain on many levels.[110]

Attracted to the Light From Lime to Projection

It's no wonder that we're attracted to light on many levels and that light has the proven potential to influence our mood and behavior. For these reasons, public venues have used lighting for great effect throughout the ages, from the first lime lights that illuminated early theatrical productions to the multimedia extravaganzas of popular music performances today. Clubs in the Roaring 20s competed for customers with lavish dance-floors sporting light-diffracting chandeliers and mirrors. In the 1960s black lights and morphing, amoebae-shaped light shows pulsed to the sound of psychedelic music. The 1970s brought Saturday Night Fever and a raft of nightclubs sporting disco balls and multicolored panel dance-floors that blinked to the beat of dance music. Laser lights were next on stage in the late 1980s and 90s to excite crowds from dance clubs to sports arenas, piercing the air with crisscrossing, multi-colored beams. During the Super Bowl XLIV half-time show, The Who staged the largest outdoor concert production incorporating laser lights. That display was broadcast to over 100 million viewers. Since then, even smaller night clubs have installed laser lights to draw crowds.

Light + Story = Greater Emotional Engagement

Projection Mapping is a natural progression in audience engagement through the use of light. Lasers, dance-floors, and other lighting techniques still have their place. But Projection Mapping can paint entire visual stories and create immersive environments, while delivering even additional impact combined with stage-craft lighting effects. While we are all affected and energized by light, storytelling and imagery are also a captivating way to excite crowds, tell a product story, or bring an additional creative dimension to a concert performance. Stories and images provided by large, immersive projected displays add an energy and emotional connection.

(a) An image of the Hindu goddess Kali was projected onto the Empire State Building as part of a display meant to spark awareness of massive wildlife extinction.

(b) Live Art Projections by Android Jones in Sydney, Australia

Figure 29: Projection Mapping live performed by digital painter Android Jones

On the Dance Floor and in the Concert Hall

Projection Mapping has revolutionized dance floors, concerts, and live events. Carrie Underwood, during the 55th Grammy Awards, performed live as her dress, as a Projection Mapping surface, transformed into butterflies, roses, and other brilliant images as she sang "Two Black Cadillacs."(See Figure:30) "They can do a lot of amazing things with projectors these days," she said backstage, holding her trophy for best country solo performance. Furthermore, Music and Art Festivals, such as Electronic Music and Art Festival "Momento Demento" in Croatia, is pushing boundaries to deliver spectacular multisensory experiences(See Figure:32 below). The "Burning Man" Festival in Nevada's Black Rock Desert, which hosts numerous art installations dedicated to community, art, self-expression, and self-reliance, provided a mesmerizing 3D, 360°Projection Mapping Show, on a faceted skull installation, constructed in wood and covered cloth(See Figure:31 above).

Figure 30: Carrie Underwood, during the 55th Grammy Awards

Figure 31: 3D, 360°Skull Projection Installation in Burning Man Festival.

Figure 32: Electronic Music and Art Festival "Momento Demento" in Primišlje, Croatia.

Morphing Public Spaces

Using multiple stacked and linked projectors, video artists can morph virtually any curved, cornered, or irregularly shaped surface, including entire building facades. These are the same techniques producers are using to enliven outdoor events, such as Lyon's "Fête des Lumières," or the Festival of Lights in Berlin (See figure:33 below).

(a) Lyon: Festival of Lights(Fête des Lumières),Cathédrale Saint-Jean

(b) Berlin: Festival of Lights, Brandenburg Gate

Figure 33: Festival of Lights in France and Berlin

Energizing Fans and Audiences

Projection Mapping of stadiums boosts crowd energy during pregame and halftime shows. At Pensacola Christian College, producers there developed and staged a full-court Projection Mapping presentation for its basketball team for one-thirteenth of than the price quoted to them by professional production companies.

(a) Left: The Pensacola Christian College basketball court before Projection Mapping.

(b) Right: After mapping: Pre-game Projection Mapping pumps up the basketball crowd at Pensacola Christian College.

Figure 34: Projection Mapping at Pensacola Christian College.

Making Events Truly Eventful – Adding another dimension – Light + Story + Interaction

From fund-raisers to product launches, Projection Mapping bathes familiar objects and environments in a whole new light. Animation, motion, and video draw greater attention, make events more memorable, and create a lasting, positive impression. Imagine, though, adding an extra dimension in Projection Mapping: Interactivity.

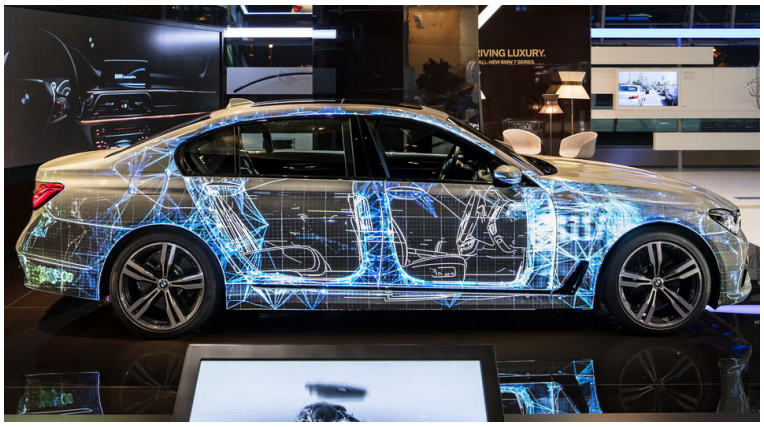


Figure 35: BMW Gets Creative With Projection Mapping On 7-Series.

2.7.3 Adding Interactivity

There are various ways we can think of the user interacting with the show. In November 2010, the digital 3D Projection Mapping technology made Ralph Lauren's flagship building, one in the US and the other in the UK, seem to suddenly disappear, then reappear, block-by-block, before they each opened up like a dollhouse to unleash a 3D parade of four-story tall models, a virtual polo match, and gigantic Ralph Lauren products. As a collection of perfume bottles appeared, the air was filled with Ralph Lauren's fragrance, adding another dimension (3D –> 4D) to the experiential performance. In 2015, Andrien M. and Clair B. designed "Hakanai" project(See Figure:36). Inside a cube fashioned from translucent veils, a dancer takes a visual journey into a 3D space between dreams and reality. The choreographed performance installation combines video Projection Mapping, CGI, and sensors to dynamically respond to the movements and proximity of its performer. Its visuals and sounds are generated and animated live, offering a uniquely different performance for each and every iteration. The audience remained stocked as the dancer appeared to be bending the light with her dance.

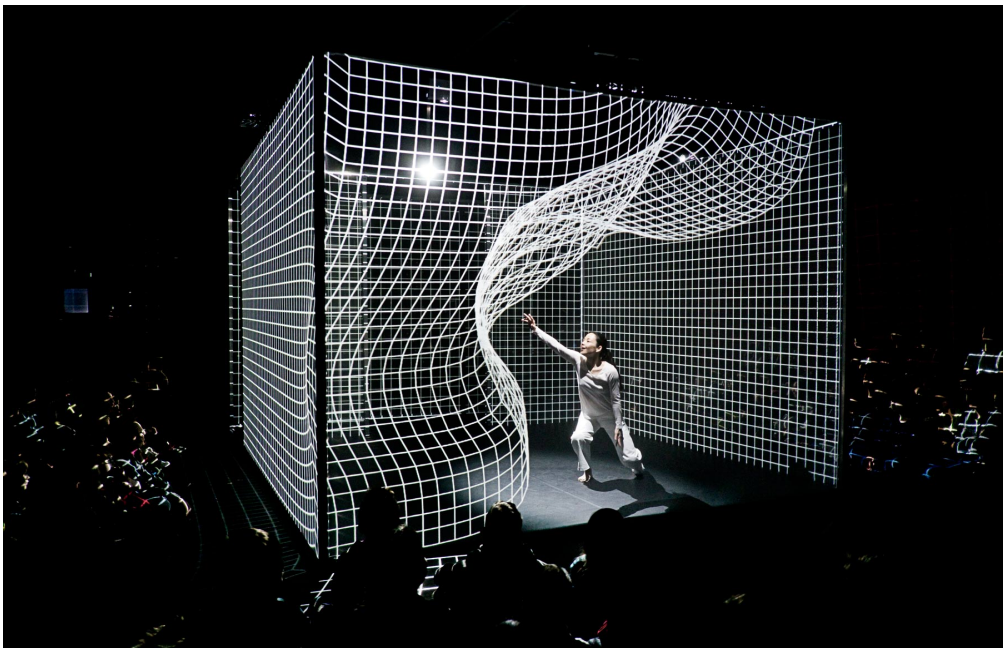


Figure 36: Hakanai: A Conceptual Dance Performance by Adrien Mondot and Claire Bardainne

As a result of the state-of-the-art Projection Mapping performances, we can introduce interactivity with the visualizations using the following techniques[111],[5]:

·Optical Tracking with Markers: In this method, a target is fitted with markers which form a known pattern to computer vision algorithms. A camera or multiple cameras constantly seek the markers and then use these algorithms (for example, POSIT algorithm) to extract the position of the object from the markers. Initially, the method requested sources of infrared light (active and passive), to track the visible markers like QR codes(circular/triangular/rectangular/crosses markers). Circles are efficient for tracking motion capture, since a circle has no edges.(See an example Figure:37 below) "Strictly speaking, a simple circle provides more robust tracking since it is rotation invariant, the tracking point remains the same even if there is rotation in the shot," says Discreet's Marcus Schioler[112]. A circle also tracks better with zoom and focus changes. Things with angles (triangles, squares, crosses etc) are better for tracking how something is deforming and track its scale as it changes size - rotation, skew etc. The optical tags can be pattern images that can be identified under certain parameter conditions, but few years ago LED markers appeared to be in the vogue. Digital Fusion's Rony Soussan notes that he's " found that glowing or illuminated objects can give undesired results to the camera lens, like glowing and focus issues" However, Discreet's Schioler points out that "using an LED marker could be a big help in some tough lighting conditions as the marker would remain visible even under dark shadows."

Figure 37: Movie: The Avengers, Special Effects with Computer Generated Imagery(CGI).

·Optical Tracking Marker-less: With the recent emergence of better cameras and more accurate sensors in soon-to-be mainstream devices, the tracking systems are transitioning from image or QR code based activations to marker-less experiences. Current implementations of marker-less AR (often referred to as 'dead reckoning') use sensors in devices, like Depth Cameras to accurately detect the real-world environment, such as the locations of walls and points of intersection, allowing users to place virtual objects into a real context without needing to read an image. Such technology was implemented by Leap Motion, Microsoft HoloLens, Meta 2 Development Kit.

·Other creative input tracking techniques: Many tracking methods have been used in the AR applications(keyboard, conductive ink, invisible markers/IR invisible ink). These include mechanical, magnetic (measuring the intensity of in homogeneous magnetic fields with electromagnetic sensors), ultrasound (tracking time of arrival of an acoustic signal like dolphines do), inertial (use data from accelerometers and gyroscopes), vision-based as well as combining data from multiple tracking methods, using multiple sensors for more efficient tracking (Sensor Fusion).

2.7.4 Applications

If Oculous Rift's Virtual Reality reigns supreme as the most mind-melting of today's visual-based technologies, the surreal, immersive art of rearranging surfaces beyond recognition known as Projection Mapping must run a close second. Perhaps they're even neck and neck. But Projection Mapping remains limited. Much like film, it requires lots of projected light and a sufficiently dark surrounding atmosphere, to say nothing of the cost of its operating systems, especially high resolution projectors. In this subsection we will discuss numerous applications and systems, built to ease the process of Projection Mapping, as well as include interactions, using various tracking techniques.

■ **Development Kits**

☐ HeavyM : Paris-based Digital Essence is hoping to bring the trippy tech to the masses with HeavyM Software, a ready-to-use Projection Mapping tool[113]. HeavyM is an intuitive Projection Mapping software, it is currently running in beta version. It allows the creation and projection of visual animations with great facility, in order to be projected on real surfaces with a projector. HeavyM's interactivity should set it apart from other Projection Mapping software. In initial version, HeavyM users could only use the synchronization effect with music, as it was the easiest feature to implement. However, for professional projects, users will be able to synchronize HeavyM with motion capture technology and live tweets, with more native interactive plugins coming in the future. Founded quite recently, by engineers Etienne Mathé, Arnaud Berthonneau, and Romain Da Costa, Digital Essence specializes in Interactive Projection Mapping for professional customers. After taking on several mapping performances for, as Berthonneau described it, corporate events and live concerts, the trio really wanted to take the technology out of the big event arena, and place it in the hands of average users."Our vision is to democratize the technology and put mapping in locations and places where it's normally not possible," Berthonneau said.

While all of this sounds pretty awesome, Berthonneau said that HeavyM has a few limitations. Currently, the software is limited in terms of video management and clip manipulation. Users may also encounter limitations if their project demands a very fine level cutting area. However, future HeavyM releases should take care of such issues. Beyond mere community, the real goal here is to create a marketplace where users and contributors can meet, post videos, loops, and modules, and make them available either for free or for sale. "We believe that a collaborative model can be very interesting," Berthonneau said. "If we provide a basic version that is powerful and interesting for people, we think they may become involved in its development and improvement."

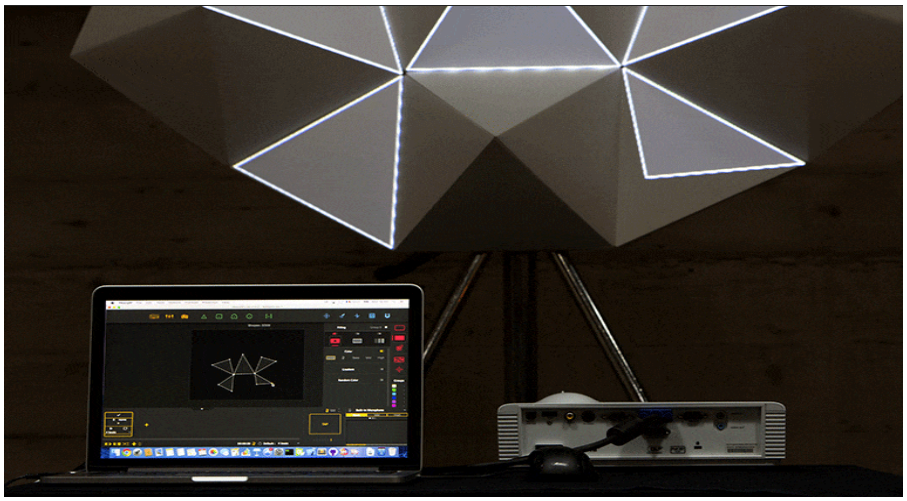


Figure 38: An example of using HeavyM, mapping image-to-geometry.

☐ DepthKit: Depthkit is the first volumetric filmmaking tool designed to empower creators of all experience levels to participate in the cutting edge of immersive storytelling. Depthkit is downloadable software that allows capture of full motion video and depth (also called RGBD or 2.5D Volumetric Video) for use in interactive 3D environments like Unity or WebGL.

Volumetric Filmmaking is a growing movement in immersive (AR, VR, MR) content characterized by interactive experiences, created using predominantly 3D scanned imagery. Techniques, such as volumetric video and photogrammetry, are used in conjunction with game engines to enable viewer

agency within reconstructed lifelike environments. A true hybrid of video games and cinema, volumetric filmmaking draws inspiration from related creative disciplines, such as documentary film, immersive theater, cinematic 360° virtual reality, interactive installations, choose-your-own-adventure books, walking simulator games, and generative art.

Depthkit can be considered like any video camera, but enhanced with the added dimension of 3D depth. It is designed for full motion capture. Unlike other scanning software that creates static 3D models of scenes or objects, Depthkit takes a stream of 30 independent 3D scans per second for cinematic motion in true 3D. This means that Depthkit isn't the best tool to capture large environments. Moreover, Depthkit works in conjunction with static capture solutions to create interactive 3D reconstructed environments, the raw ingredients of volumetric filmmaking. Depthkit Studio also provides the software and resources to set up a 360 Depthkit capture environment in a studio setting, capable of capturing full-body volumetric video. This includes multiple sensor capture as well as the ability to synchronize professional video equipment to the array.

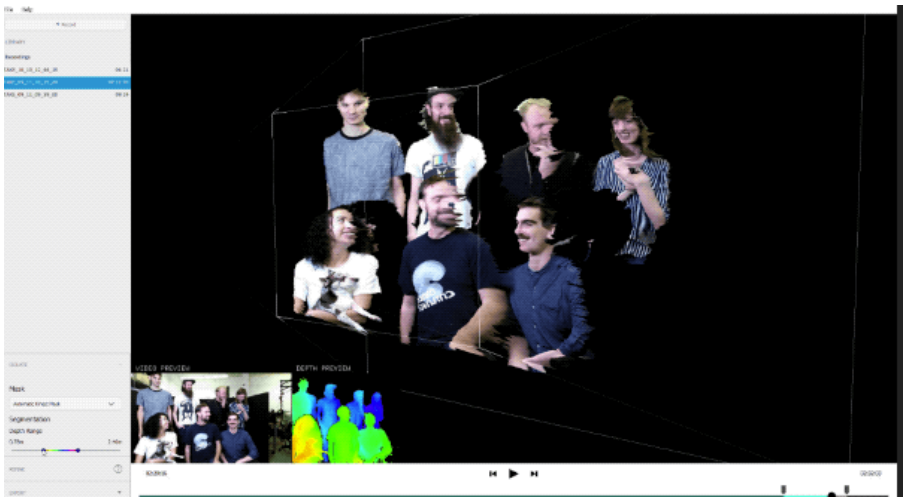
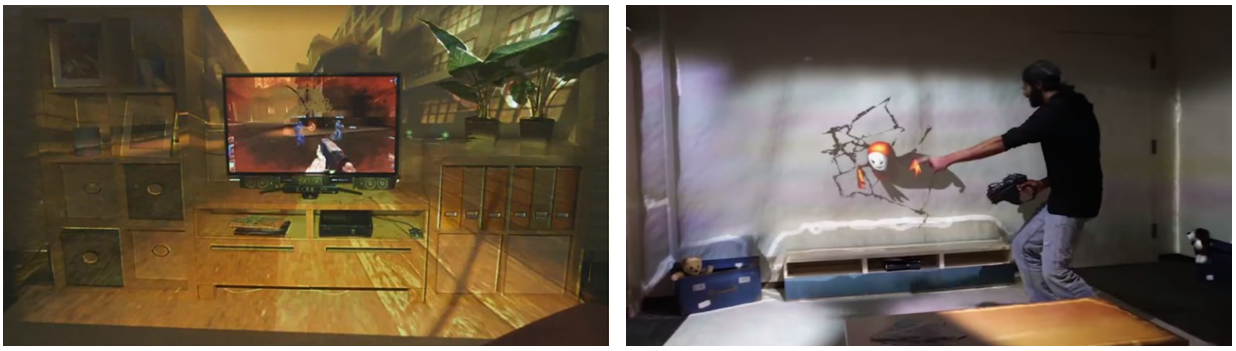


Figure 39: An example of using Depthkit, 3D-scanning humans.

- Kinect Projector Toolkit: Kinect Projector Toolkit is a library for Processing and OpenFrameworks which calibrates a projector to a Kinect depth camera, aligning a projection to the physical space it's lighting. Some of the best hacks are, when mashing together two pieces of commercial hardware that weren't designed with each other in mind. That's exactly what Gene Kogan did when he made the Kinect-Projector Toolkit for OpenFrameworks and Processing. This toolkit allows mixing the human-body tracking features of commercial depth sensor with imagery from a commercial projector. The toolkit also allows, to acquire the skeletal data from Kinect and accurately map it to any point in a virtual space. Then the virtual space is projected over the physical to complete the illusion. At the heart of this hack is the calibration process, similar to the one found in RGBDToolkit (DepthKit). This process allows the computer to work with the intrinsics and relative positioning of both the Kinect and projector lens. As long as the computer receives this data, it can accurately map skeletal joints to virtual objects, such that the interactions mirror what happens in the real world.

Figure 40: An example of an Interactive Installation, created using Kinect Projector Toolkit.

- ☐ IllumiRoom & RoomAlive: IllumiRoom is a Microsoft Reasearch project that augments a television screen with images projected onto the wall and surrounding objects[72]. The current proof-of-concept uses a Kinect sensor and video projectors. The Kinect sensor captures the geometry and colors of the area of the room that surrounds the television, and the projector displays video around the television that corresponds to a video source on the television, such as a video game or movie(See Figure:41(a)).



(a) IllumiRoom (b) RoomAlive

Figure 41: Microsoft's Spatial Augmented Reality Applications

After IllumiRoom the team continued the research demonstrating a new concept of RoomAlive[10]. According to Brett Jones, in IllumiRoom, a single projector was used to surround a traditional television with projected light to enhance gaming experiences. IllumiRoom explored focus plus context visualizations anchored by traditional screen based gaming experiences. RoomAlive, on the other hand, explores gaming completely untethered from a screen, in a unified, scalable multi-projector system that dynamically adapts gaming content to the room and explores additional methods for physical interaction in the environment.(See Figure:41(b).

- ☐ Open Pool: Digital Billiard DIY Kit: Open Pool is an open source, augmented billiard reality system, using Projection Mapping technology [114]. With a standard pool tables, balls, cues and some add-on hardware devices, Open Pool transforms the standard pool experience. The Kinect v2 attached on the ceiling watch the locations of balls and a projector shows cool visual effects around the moving balls. Also, the OpenPool Pocket Detector, an original sensor device put on each pocket, made of mbed, LEDs, IR sensors and a 3D-printed case, sends signals when a ball falls into a pocket. More importantly OpenPool is open source, so not only can users play, but they can also create and add on their own effects.

(a) (b)

Figure 42: OpenPool examples installations

■ Latest Custom Software & Hardware Installations

- ☐ Cunductive Ink + Projection Mapping = Magic Storytelling: As part of their presence at the inaugural Retail Design Expo, Dalziel and Pow created a space to stand out from the crowd, a conversation starter: experimental, innovative, playful, integrated, exploratory and, to many visitors, surprising.[115] The idea came from the work they created for Portuguese children's' wear brand Zippy, magnified as a prototype to test ideas and push the range of interactions and feedback. Screen-printed illustrations spring to life when touched, as conductive ink triggers a host of playful digital animations – a mix of informative and charming content, from product information to brand stories to chasing ducks.

  Pushing the boundaries of storytelling within a space, making it fun and engaging, putting ideas before hardware. A total of 48 interactions and over 100 unique animations make up the broad spectrum of content on the stand, from simple sound toys to more complex exchanges that layer Projection Mapping and movement around a physical space. All created in-house at Dalziel and Pow with multi-disciplinary teams along with Bare Conductive, K2Screen and Ototo for their help. Dalziel and Pow is a London-based agency with over 30 years' experience in brand and retail design. Their passion is creating great customer experiences. They develop brand environments and communications across all key touch-points, translating inspired concepts into retail experiences that are unique and engaging for the consumer, effective and profitable for our clients.(See Figure:43 below.)

Figure 43: Cunductive Ink + Projection Mapping Installation

☐ Projection Mapping on Apple Keyboard: Mash Studio created this fun vignette showcasing how a physical desktop can be brought to life with Projection Mapping. As the user types, the keys light up reacting to the user's input. The space between the display monitor and keyboard is used as contextual visual feedback(See Figure:44 below.).The keyboard also comes complete with a projection mapped extrusion effect, giving the keys that extra sense of depth[116].

Figure 44: Projection Mapping on Apple Keyboard

- Projection Mapping Multiplayer Game: EELS is a multiplayer game developed by B-Reel London and produced by Leo Seeley. Brought to life through Projection Mapping onto three-dimensional surfaces, the game brings the virtual worlds and the real worlds together. Using a smart phone, the user controls the left or right direction of an eel as it moves across the 3D surfaces. Not only is the real world space augmented with Projection Mapping technology, the game is fully interactive(See Figure:45 below.). The EELS game has taken some important steps to bringing interactive Projection Mapping to the masses[117].

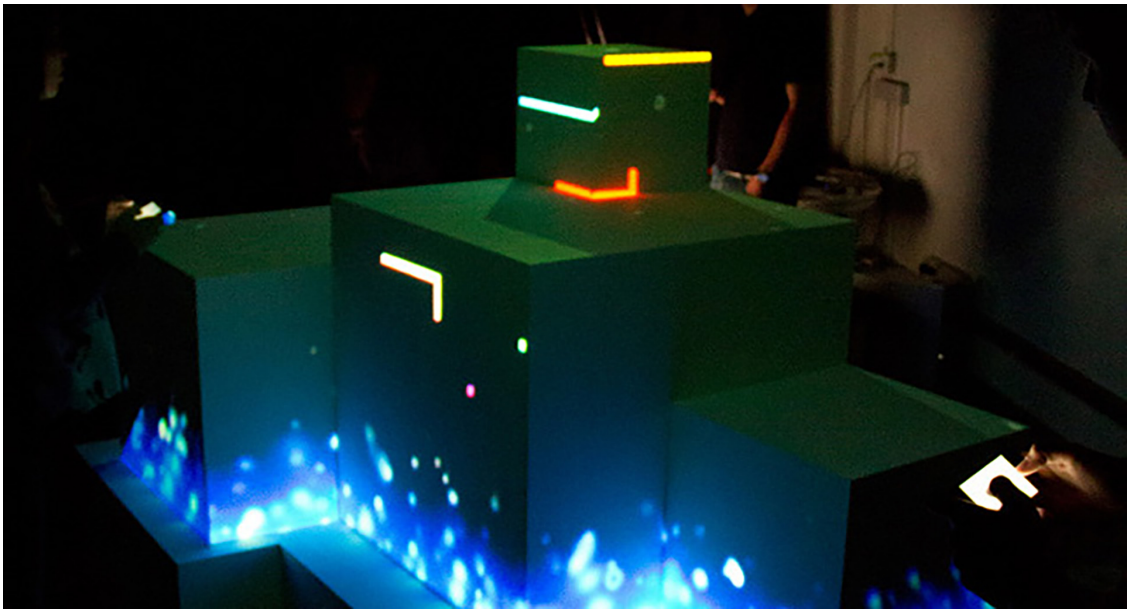


Figure 45: Projection Mapping Multiplayer Game

- Push Play: Brent Watanabe combined a custom computer application, acrylic paint, and video projection to create this video game installation for "Push Play," an event at Seattle University that explores social, cultural, and philosophical issues around games.[118].(See Figure:46 below.)

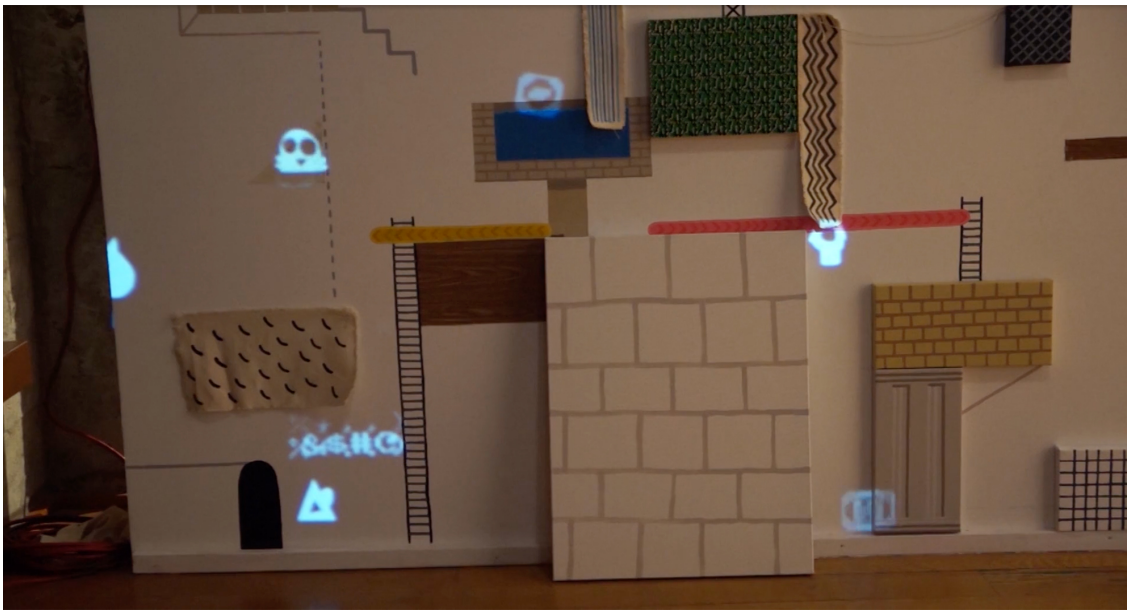


Figure 46: Interactive Video Game Installation

- Umbra: Umbra is an artistic dance performance that uses some neat interactive Projection Mapping[119]. With only a webcam to track the movement of the dancer, dynamic and real time effects are mapped around the performer for a haunting a surreal effect. The piece was done by Craig Downes, an audio/visual freelance artist from Belfast, and we were fortunate enough to get details straight from the source. Here's more about Umbra from Craig in his own words.

  It was made in Max/Msp, a visual programming language that helps you build complex, interactive programs without any prior experience writing code. It's been used to create many Projection Mapping tools/experiences, including VPT. It uses a Logitech C920 webcam, which hooked up to Max, where the magic begins. Code is then written so that the webcam tracks the dancers movements. Once this was working the ghost-like shadows where created using Jitter (the visual programming part of Max). This allowed for the feed to put through a small video reverb to create these ghost like shadows. RGB values and greyscale were coded and effected using different elements of jitter. It was sent to MadMapper using Syphon. Syphon is an application that allows users to send graphics content from one application to another. Once in MadMapper it was important to ensure the frame rates were being maintained at the right level whilst being transferred as they were going to mapped be to a large scale. Also, once in MadMapper a few placements of the shadow were tested before we see the final ones in the video. The magic all happens in Max; MadMapper was used large scale giving me the ability to move the images freely where ever needed. The first instance of Umbra was shown at the University Of Ulster Magee.(See Figure:47 below.)

Figure 47: Umbra: Interactive Projection Mapping Dance Performance

- ☐ Awake: Interactive Ink + Projection Mapping: Sofia Aronov's "Awake" is an interactive painting that combines paint, projected light, and capacitive sensors to create an engaging, "choose-your-own-adventure" viewing experience. Viewers are guided to follow their curiosity and explore the coral reef painting. When viewers touch a piece of coral, the painting comes to life with projections of marine-life animations. The experience for each viewer is unique depending on which marine element the user decides to interact with and the order of their interactions. This piece is the first iteration of a series of interactive paintings that use sensor triggers and projected light. The marine illustrations were painted on white paper with Bare Conductive, a conductive ink. Each individual marine element is connected to an Arduino Unovia alligator clips on the back of the painting. The Arduino gathers proximity data and sends it to a Processing sketch, which triggers the projected animations[120].(See Figure:48 below.)

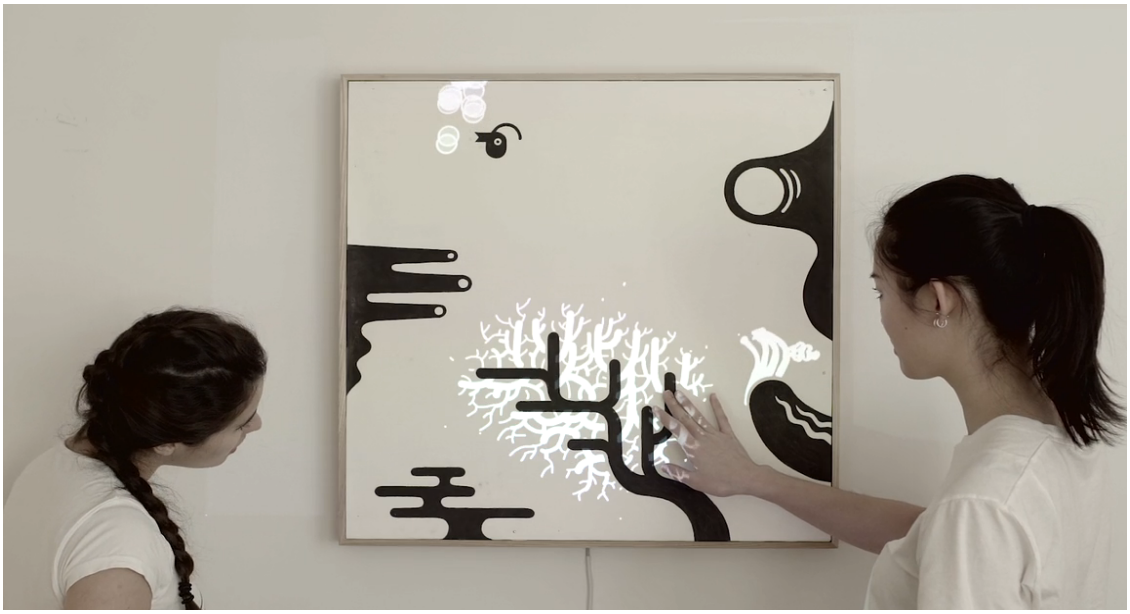


Figure 48: Awake: Interactive Ink + Projection Mapping

☐ Interactive Dining Experience: StoryLab worked together with Dig & Serve to produce a fully interactive dining experience featuring Projection Mapping, locally sourced food, and a narrative that took diners to outer space.[121].(See Figure:49 below.)

Figure 49: Interactive Dining Experience by StoryLab

☐ Cloud Pink: Cloud Pink is an interactive installation by Seoul-based creative Everyware, formed by Hyunwoo Bang and Yunsil Heo, that invites visitors to poke and prod a fabric screen, thereby causing a visual reaction in the projected simulation of wafting pink clouds.[122].(See Figure:50 below.)

Figure 50: Cloud Pink: Interactive Installation

☐ Bordeless by TeamLab: Japanese art collective TeamLab used Projection Mapping technologies to craft a magical dream world in the Digital Art Museum in Odaiba, Tokyo[123].(See Figure:51 below.)

Figure 51: Borderless by TeamLab

☐ The Enlightened One: Three interactive experiments by Jesse James Allen using HeavyM, Spout, Beautiful Chaos and Unity, a Leap Motion Controller, and original sound design[124].(See Fig-

ure:52 below.)

Figure 52: The Enlightened One: Interactive Installation by Jesse James

2.7.5 Summary

In this subsection we discussed about the impact of Projection Mapping, as well as the potential repercussions of adding interactivity into Projection Mapping Performances. Below, we summarize how Projection Mapping takes light to a whole new level:
Immersion:Projection can map an entire venue, enveloping the audience in an image and color-rich environment that transforms the ordinary into the extraordinary.
Impact: Story, flowing images and animation can create a greater emotional impact than static slide shows or music alone.
Motivation: Greater emotional impact leads to motivation—whether it's to get up and dance, consider a product purchase, or join the crowd in cheering on a sports team.
Setting tone: Light influences mood. Projection Mapping can extend the mood-altering properties of light into a more focused and shared experience.

As a result, Projection Mapping can help producers or owners, who want to motivate and energize their customers and audiences. In the sequel of Interactive Projection Mapping we also presented the latest applications and software conducted, in order to bring Interactive Projection Mapping closer to the crowds.

2.8 Video Mapping Tools

2.8.1 State-of-the-art VideoMapping Tools

Artists and advertisers use video mapping to add extra dimensions, notions of movement, or even optical illusions to objects that are or were previously static, to create an audio-visual kind of narrative. In this section we will briefly inspect the most popular Video Mapping tools, as a state-of-the-art of Projection Mapping and conclude to one, according to our needs for Interactive Projection Mapping.

Artists/designers perform Projection Mapping in two phases: manufacturing, where the scenario, modeling, motion graphics, video editing and other processes are created, and then the production phase that adds elements like software, cabling and installation among others, as we will discuss in detail in the Requirements Analysis section. Below we will mention some of the most-popular VideoMapping Software as Standalone Software, Interactive Installations, VideoJocking and Apps/Rasbery Pi's after an online survey and personal experimentation:

•Standalone Software:

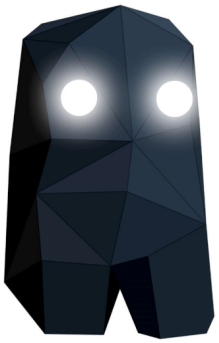
•MadMapper:

This is one of the best software for video mapping that is simple and easy to use, powerful and also very versatile – it is the ultimate video mapping tool. With MadMapper, unlimited number of videos can be mapped on any real life surface with as many projectors the graphic card can handle. LED arrays can also be controlled when light mapping, in real-time, these can be fed with video content or generative materials, as well as control moving lights with pixel content. Madmapper is supported by an enormous community, which results in constant updating of the software. New features, such as a Dark User Interface, a live editor to code materials and an online library to share materials have been imported. It also features the miniMAD hardware companion which controls the light, with automatic network synchronization for multi-projection, and DMX output to control lighting equipment. Other salient features include an ultra-fast video player, multiple inputs, masking, video effects, Mesh warping, 3D Calibration and lighting, soft edge, spatial scanner, LED mapping and much more. Madmapper software can also be used for architectural video mapping, art installation, stage design and live shows.
Additionally, MadMapper is more of a mapping software and pixel manipulation software than it is a visual mixing or VJ software, as it will take video inputs and warp pixels, convert to ArtNet (for video to control lighting instruments) or video map the surface. It was designed to "shape video" between applications and that's exactly what it does best; a great accompaniment with any of these other pieces of software. Madmapper is hard to beat. And with the release of version 3, Madmapper now works on Windows PC's. However, it is a fantastic tool that will set you back $435. Out of most of the desktop Projection Mapping software, Madmapper is hands down one of the easiest to use out of the gate. Their GUI is logical, all sliders and buttons are MIDI/OSC/Keyboard programmable.

Cons: There are afew con's with Madmapper. One cannot feather masks or make bezier curves with them (Bezier can be added to mapped layers, but masks are great for removing parts of a scene that are not wanted). MM has "presets" for switching between settings, but it does not offer cross-fades between presets or a timeline to arrange for show playback. It is a dedicated mapping software that can handle almost any project when combined with other performance software such as VDMX.

Price & License: $400 (2 licenses)

•Millumin:



Millumin is a rapidly growing tool for warping video. It offers non-linear video editing tools, has many tutorials and interfaces well with After Effects. 30 day trial versions of the software are available. Millumin has seen some great changes over the years and has been quickly encroaching on the theatrical stronghold of Qlab for performances. Simply put, Millumin allows the creation of a timeline with transitions and sequences for audiovisual shows. Millumin is suggested for Mac users. Below we list some of thefeatures that V2 includes:

- •ISF shaders support
- •LED mapping
- •Arduino, Kinect, Kinect 2, LeapMotion (support)
- •Data track (to send signals such as MIDI, DMX, OSC, Arduino)
- •Multiple canvases and vertical soft-edge
- •INetSyphon and NDI support

A special edition of Millumin, called "V2+" includes additional features:

-Timecode support (MTC and LTC)

-Advanced audio (routing channel per channel, audio effects for spatialization)

-Preview grid (to check camera, next columns)

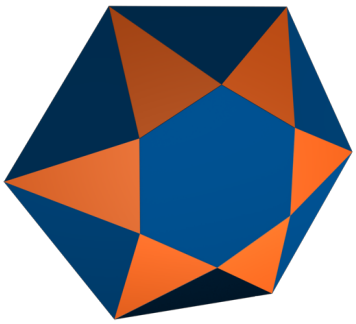
-Plugin for Cinema4D working with 3D structures

-Prototype a scenography in 3D (then visualize it with VR headset)

Cons: Currently, there is no version of Millumin for Windows or Linux. The GUI takes a few tutorials to get used to and its full license price point is higher than others.

Price & License: ($49-$79-$799 Student-Rental-Full)

•Mappio:



Visution Mapio is the best software for video mapping for projection shows, as it not only allows the use of a standard screen, but also any inclines like cylindrical and spherical shapes, among others. Its key features include, two app modes as a plugin or stand-alone app, solid color or gradient for slice, capture broadcasting video streams, snapshots, importing 3D file format support, editable vectors and image masks, tools for splitting masks and slices, edge blending, advanced warping Bezier tool, and much more. An important feature is using the "undo" function unlimited times, copy and paste slices or masks, and auto-save and recover the project after a crash. It was one of the earlier mapping software available for Windows. Since 2015, Mapio has undergone significant improvements in its hardware acceleration and memory consumption. Some of Mapio 2 features include:

•Network distributed video playback with frame-to-frame synchronization providing remote control of all machines from one master server. It also allows adjusting mapping configurations on-fly and supports remote start (WakeupOnLAN)/restart of the machine and remote updates of Mapio as well.

•Automation of autonomous work with MapioObserver, automatic start, and control of the main app, automatic restart.

•Support of audio files apart from video files allowing to play them synchronously with video files. This feature is needed for professional use, sometimes the sound is delivered apart from the video content.

•Warp is a brand-new tool that allows any warps anywhere across the slice within the defined area.

•Native support for NDI, Spout, and Kinect 1 or 2.

•QML generator and QML Scripts for interactive scenarios

Cons: The latest and most powerful versions of Mapio 2 are only available for Windows.

Price & License: (€100- €200- Lite-Pro Version)

• Heavy M(as aforementioned in the state-of-the-art)

HeavyM is one of the best video mapping tools to use for live events such as live concerts. It allows to literally transform any space according to one's needs. The best part is that HeavyM is compatible with all video projectors. This software solution will work flawlessly, irrespective of the projector model. Additionally, there is a massive animations and effects library, where are over 1000 combinations of effects that can be used to impress the audience. HeavyM has made a lot of waves after its lengthy beta period and successful Kickstarter. Opting for a subscription model or lifetime plan, HeavyM has found a way to keep the software price point low. The software is straightforward to use and continues to improve as time passes.

Cons: HeavyM feels a little late to the game compared to some of the other software that have been released and improved over the years. Their Free version is nice, and their "Start" version is at a good price point, but both versions are lacking some key features that only their "Live" version has to offer. And while the "Live" version starts at $69, that is only for 3 months. A lifetime version of Live is $359 and at that price point, you can get a copy of Madmapper that has a wider feature set.

Price & License: ($0-$449 Free-Lifetime)

•**Interactive & Installation:**

•Isadora:

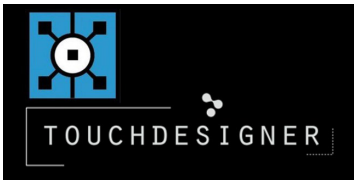
Isadora works well with artists, designers, and performers, who want to add video and interactive media to their projects, as it combines a media server, visual programming environment and powerful video/audio processing engine for an interactive media playback. Stunning effects can be created in real-time, in a user-friendly and easy to learn and use platform, which features 8 HD video playback channels, unlimited video layers, 6 independent video projector outputs, integrated video mapping tool, 4 live camera feeds, custom effects via Open GLSL, built-in video tracking, low latency response to real-time output, and so much more. One of the great things about it is, that it allows fast generation and customization of sequences. Isadora empowers video designers and interactive artists to create responsive media experiences. As a media server or Projection Mapping tool, Isadora is flexible and highly customizable. Below are some key features:

- •8 Channels of HD Video Playback on an SSD Equipped i7 Laptop.
- •Unlimited Video Layers.
- •Integrated Projection Mapping Tool.
- •6 Independent Video Projector Outputs.
- •4 Live Camera Feeds.
- •Fast, GPU Based Video Effects including FreeFrameGL.
- •Custom Effects via OpenGL Shader Language (GLSL).
- •Syphon & Spout Integration.
- •Professional Video Codec Support including HAP.
- •Professional Video Codec Support including HAP.
- •Input and Output via OSC, MIDI, Serial, TCP/IP and More
- •Low Latency Response to Real-Time Input
- •Built-In Video Tracking Support.
- •Arduino I/O via Built-In Serial Input/Output.
- •Javascript Language Support.
- •Detailed tutorials for Xbox 360, TouchOSC and many others.

Cons: Some things that hold it back are its learning curve and playback performance, which can reduce framerates if your media and project are not optimized.

Price & License: (Mac/PC)($450)

•TouchDesigner:



This is a visual programming environment for PC that is free for use, but not for commercial use. TouchDesigner gives the tools needed to create stunning, real-time video mapping projects with rich user experiences, whether it's architectural projections, rapid-prototyping, or interactive media systems. TouchDesigner's suit of features and open customization makes each project possible, with multiple solutions to tackle every job, whether small or large in ways that fit each scenario and budget. Features include tight integration with real-time 3D engine for complete pre-visualization, support for different projection formats for mapping, dome, VR, stitching and environment lights. The stoner tool is useful for keystoning, grid Mesh warping and masking, while the Kantan Mapper helps with 2D mapping and masking which includes Bezier and freeform shapes.3D is usually the most challenging bit, so TouchDesigner's CamSchnappr tool for 3D model based projection alignment and calibration comes in handy. It also supports third party apps like Vioso and Scalable Display's auto-mapping calibration tools. Additionally, it is great for large multi-screen installations all the way down to small interactive pieces. Derivative offers support and they have an ever-growing community of users.

Cons: TouchDesigner has its own learning curve and the professional license can be expensive. Unlike Isadora, TD does not offer scene based programming. That being said, Touch Designer is great for achieving specific tasks that we have time to plan, build, and troubleshoot.

Price & License: ($300-$600-$2,200 Student-Commercial-Pro)

•**Interactive & Installation:**

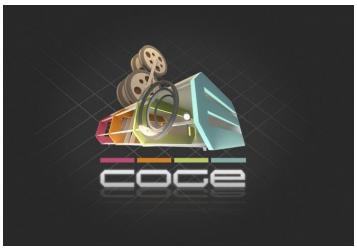
•Resolume Arena:

Resolume Arena is a performance suite that caters for video artists, VJs and designers because it has video playback, live editing, VJing and multiple sources which support video mapping as well. Despite being pricey than other best video mapping software listed here, it is a robust platform with layer based mixing software, and other great features such as faster deck switching, a flexible interface with crisp high-resolution screens, plus any color can be picked and mapped to follow light changes. With the Arena tool, video can be projected on any type of surface such as complex geometrical structures or whole buildings. Its edge blending feature allows to project a widescreen image with two or more projectors or wrap around for a 360-degree experience. Anything can literally be mapped, from buildings to pumpkins, DJ booths to cars, and LED mapping on giant stages. Arena also allows sending out colors, which will be in sync with the visuals, with real time rendering, meaning everything happens on the spot. This software can also be integrated with other apps like Spout on Windows, but also provides available connection with a webcam. Arena combines all of the functionality and features of their "Avenue" VJ software along with Projection Mapping, projector blending, 3rd party plugins, and SMPTE sync. They are now on Version 6 with new enhancements to their Projection Mapping toolset as well as DMX / Lumiverse / Artnet support.

Cons: Resolume Arena is one of the more expensive applications. Its feature set is robust but also functions as a layer based mixing software. It favors it's own codec DXV which can add complications down the road when sharing VJ bins with other programs. They also favored DXV codec over the HAP codec(video codec-compress/decompress for fast decompression on modern graphics hardware). HAP has been rising in popularity across the VJ-sphere and is compatible with more applications. All in all, It can be a turnkey solution if it fits the budget, but even the training tutorials can cost up to 129€).

Price & License: (MAC-PC $500-$954 Student-Full)

•VDMX – Module8 - CogeVJ:



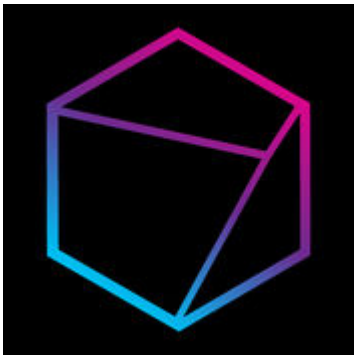
VDMX, Modul8, and COGE VJ do not offer Projection Mapping like the other applications , but they can easily be used in conjunction with all of the other Projection Mapping software and in some cases, they can outperform the all-in-one VJ/Projection Mapping software. For instance Imimot software; the makers of Coge VJ, make two other products, Mitti & Vezer that give timeline based playback options to any of the mac VJ or Projection Mapping applications. Module8 is owned and operated by Garagecube, which also owns MadMapper. They offer discounts on MadMapper to Modul8 owners (and vice versa). VDMX, offers a modular based setup which can be as complex or minimalistic. Using Syphon, it is easy to send visuals out of VDMX and into other mapping applications.

Cons: The downside to all of these is that they are Mac only and over the years, Apple has pivoted away from the pro market.

Price & License: (MAC only - VDMX($350)-Module8($400)-CogeVJ($129):

•**Apps & Rasberry Pi's:**

•Optoma Projection Mapper:



What was once known as Dynamapper has transformed into an iOS, Android, Amazon app that acts as a standalone Projection Mapping tool.It works on Android and iOS. Custom content can be side-loaded, plus a variety of visual packs is provided in their store. It works by mirroring the device screen, allowing the user to make adjustments and then run it full screen through the projector. It also has a recording functionality, that allows the user to record the mapping and share it with other devices, such as a computer or alternative playback device.

Cons: This app packs a punch, but can be cumbersome to use on smaller screens. The number of layers and playback that it can support will depend on the device the user is using. For example, an android app on an LG G5 can playback on 10 simultaneous layers. iOS may limit the user due to Apple's dongle monopoly factor. Cheaper HDMI or VGA adapters might give out after a while, and if for example an iPad for $399 is combined with a $5 app, and a $49 adapter, a better more versatile mapping software or 13 Raspberry Pis can be bought.

Price & License:(iOS, Android, Amazon - $4.99)

•PocketVJ:

The PocketVJ is a standalone open-source projection-mapping, presentation-playing, video presenter tool. Created by Marc-André Gasser in Berne, Switzerland. The PocketVJ can be bought or built from the Mag Design Github repository: https://github.com/magdesign/PocketVJ-CP-v3. The player pack offers Sync, Projection Mapping, Seamless Video Playback, Syphon, Audio streaming, Screensharing, HTTP browser for file loading and the player controlling. A presenter mode, Pi Wall for linking and syncing multiple displays, and is customizable to the user's needs. Furthermore, it's Open-source.

Cons: The PocketVJ has a wide range of features and for that reason, the web browser based control panel that can feel cluttered fr an inexperienced user. Each icon in the control panel executes a bit of code on the raspberry pi. Sometimes it can take a moment before the player updates; Outside of that, once the user gets to know how to use this device, a lot can be achieved.

Price & License: ($116.267)

•miniMAD:

The miniMAD could handle some heavy Projection Mapping when it was first released and has been tested on multiple installs. Nowadays, it supports DMX playback for control of LED devices. The MadMapper team continues to update their products and take in user feedback. At $180, this box is a solid pick for almost any installation. It auto syncs with other miniMADs, offers quick controls to adjust mapping, playback, and cropping, and when paired with the MadMapper app, it optimizes video export for best performance.

Cons: The miniMAD requires a full mad mapper license to handle the optimized export, so on top of the $180, another 435 must be spent for MadMapper.

Price & License: ($180 *)

2.8.2 Video Mapping Tool for SceneWizard

In the sequel of presenting Video Mapping tools, we will indulge into using one of them for our project's demonstration performance. In this section, therefore, we will briefly present the tool we inferred to, after considering the requirements of our project, discussed in section 3.3.

TouchDesigner [125] is a node based, visual programming language for real-time interactive multimedia content, developed by the Toronto-based company Derivative. It's been used by artists, programmers, creative coders, software designers, and performers to create performances, installations, and fixed media works. TouchDesigner's open and highly-visual procedural architecture encourages discovery, creativity and productivity providing an exploratory and responsive way to work. TouchDesigner is the perfect choice for any modern media curriculum.

In addition to this, TouchDesigner can be found in post-secondary education - either as program courseware or in university research facilities. In Derivative's main page there is a list of links of upcoming and recently conducted workshops that aim to provide educators and students with some tools to facilitate bringing TouchDesigner to the classroom.

History: Looking for a solution for live rendering, live effect generation, and for a fast prototyping environment Greg Hermanovic, Rob Bairos, and Jarrett Smith founded Derivative. In 2000 Hermanovic used the Houdini 4.1 code base as the initial scaffolding for TouchDesigner. This started as a passion project dedicated to the creation of a real-time 2/3D interactive environment for authoring media systems and installations. From 2002 to 2007 TouchDesigner's release title adopted the trailing 007 to 017 digits to indicate it's versioning. Finally in 2008 Derivative released a beta version of the platform as TouchDesigner 077, a rewrite of its previous incarnations that incorporates a fully procedural OpenGL compositing and effects pipeline. Currently, the software is in version 099, which is also the version we used to create the visuals.

Features: TouchDesigner covers several major areas of 2/3D production, including but not limited to:

★ Rendering and Compositing.

★ Workflow and Scalable Architecture.

★ Video and Audio In / Out.

★ Multi-Display Support.

★ Animation and Control Channels.

★ Custom Control Panels and Application Building.

★ 3D Engine and Tools.

★ Device and Software Interoperability.

★ Scripting and Programming.

Operators: Operators are the building blocks in a TouchDesigner project. These objects are represented as Nodes in the user interface and are connected in order to create procedural effects and animation. Each operator is customized with a unique set of parameters and flags that control its operation and processing. Operators, often referred to as ops, come in six varieties:

⋆ COMP (Components) – Object Components that represent 3D objects, panel components that represent 2D UI Gadgets, and miscellaneous components. These components can house entire networks of other operators.

⋆ TOP - Texture Operators handle all 2D image operations.

⋆ CHOP - Channel Operators are used for motion, audio, animation, and control signals.

⋆ SOP - Surface Operators are the native 3D objects of TouchDesigner responsible for 3D points, polygons, and other 3D "primitives".

⋆ MAT - Materials are used for applying materials and shaders to the 3D rendering pipeline.

⋆ DAT - Data Operators are for ASCII text as plain text, scripts, XML, and tables.

COMP: Component operators differ from other operators in the TouchDesigner family as they are capable of holding networks of other operators. These components encompass both 3D objects and interactive panel elements used when designing interfaces in TouchDesigner. Components also support the use of in and out connections, allowing them to act as modular components across projects.

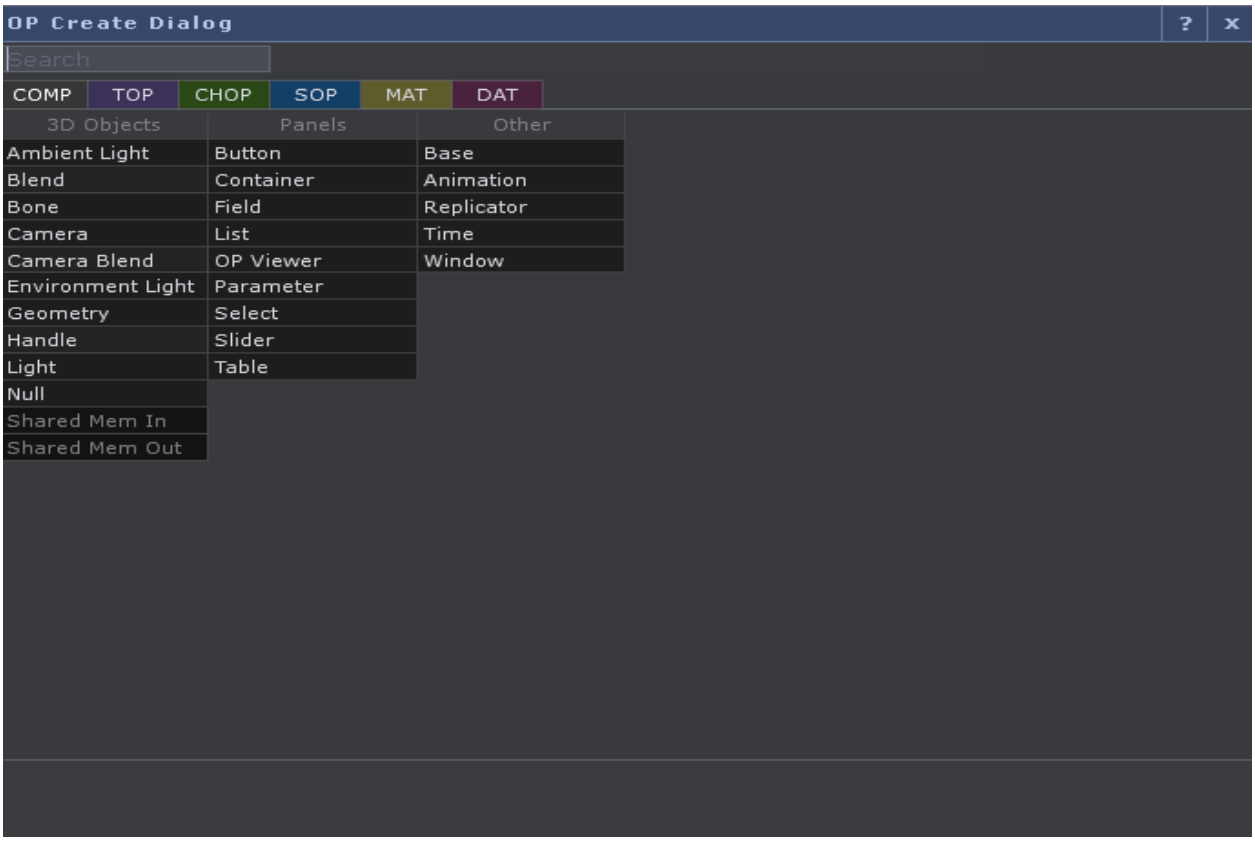


Figure 53: TD: COMP Operators

TOP: Texture operators are image-based operations that are GPU accelerated. Data in TOPs can be scaled to any resolution, limited only by the amount of RAM available on a system's graphics card.

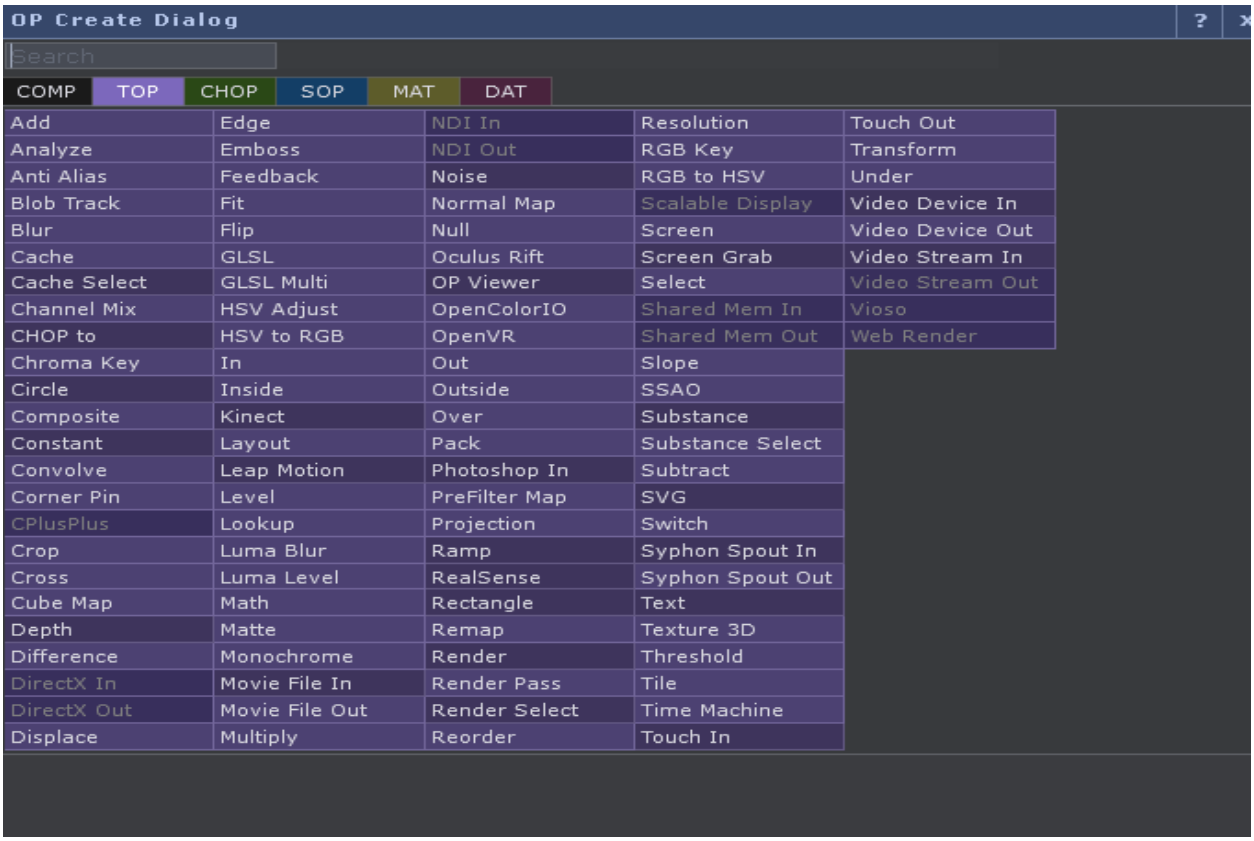


Figure 54: TD: TOP Operators

CHOP: Channel operators are the backbone of the control system in Touch Designer. Used for processing motion data, audio, on-screen controls, MIDI data, and other input devices, these operators organize data as a series of channels. According to the derivative wiki entry about CHOPs, they "were designed to reduce the tedium of motion editing and to help build and manage more complex motion."

Figure 55: TD: CHOP Operators

SOP: Surface operators are objects responsible for 3D operations and modeling in TouchDesigner. These objects are used to generate, import, modify, and combine 3D surfaces.Supported surface types are polygons, curves, NURBS surfaces, metaballs, and particles. This is perhaps the oldest part of TouchDesigner and has its roots directly in the Houdini 4.1 code base.

Figure 56: TD: SOP Operators

MAT: Materials are used as a part of the 3D rendering pipeline in TouchDesigner. Several standard material types exist, as well as materials that support importing custom vertex and pixel shaders.

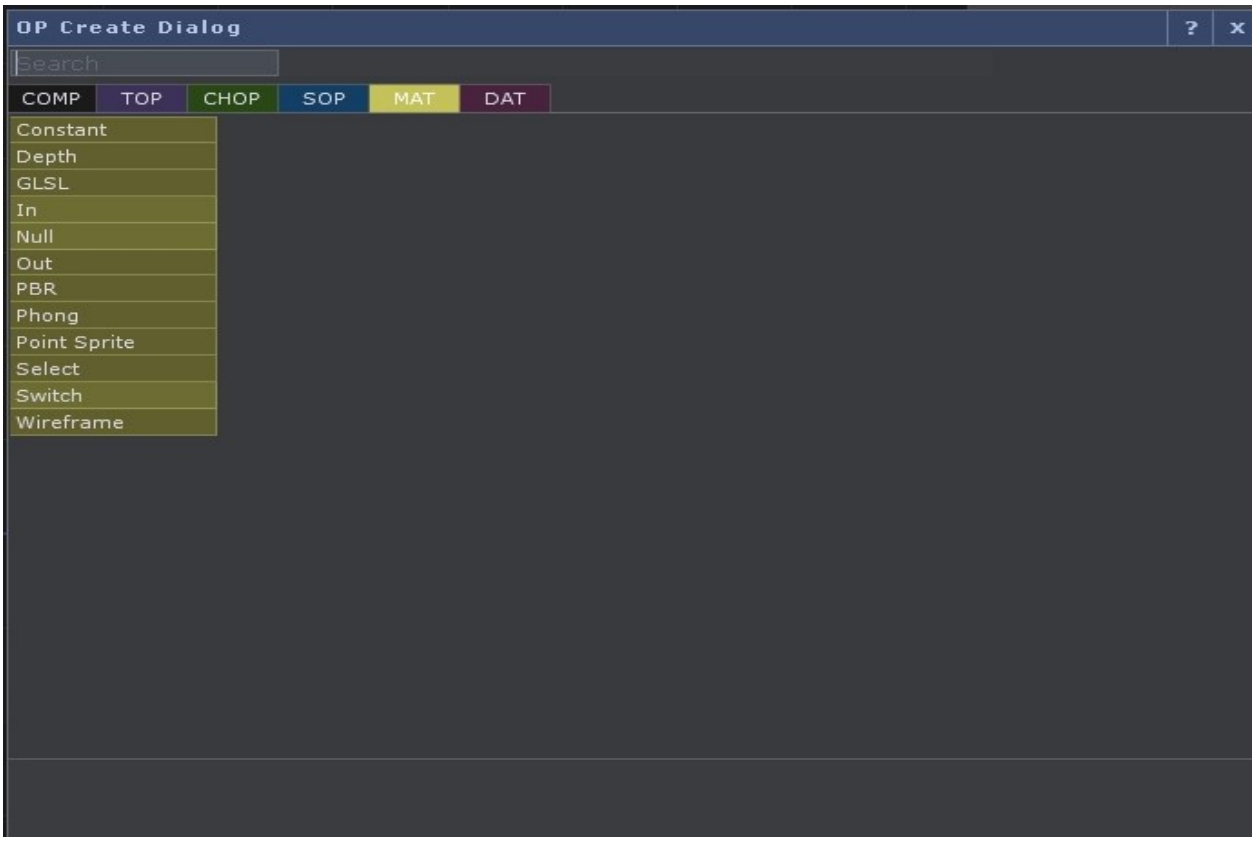


Figure 57: TD: MAT: Operators

DAT: Data operators are used to hold text, tables, text-encoded data (XML, JSON), and scripts. These operators are also sometimes used to store readme documents and other code comments in a given network.

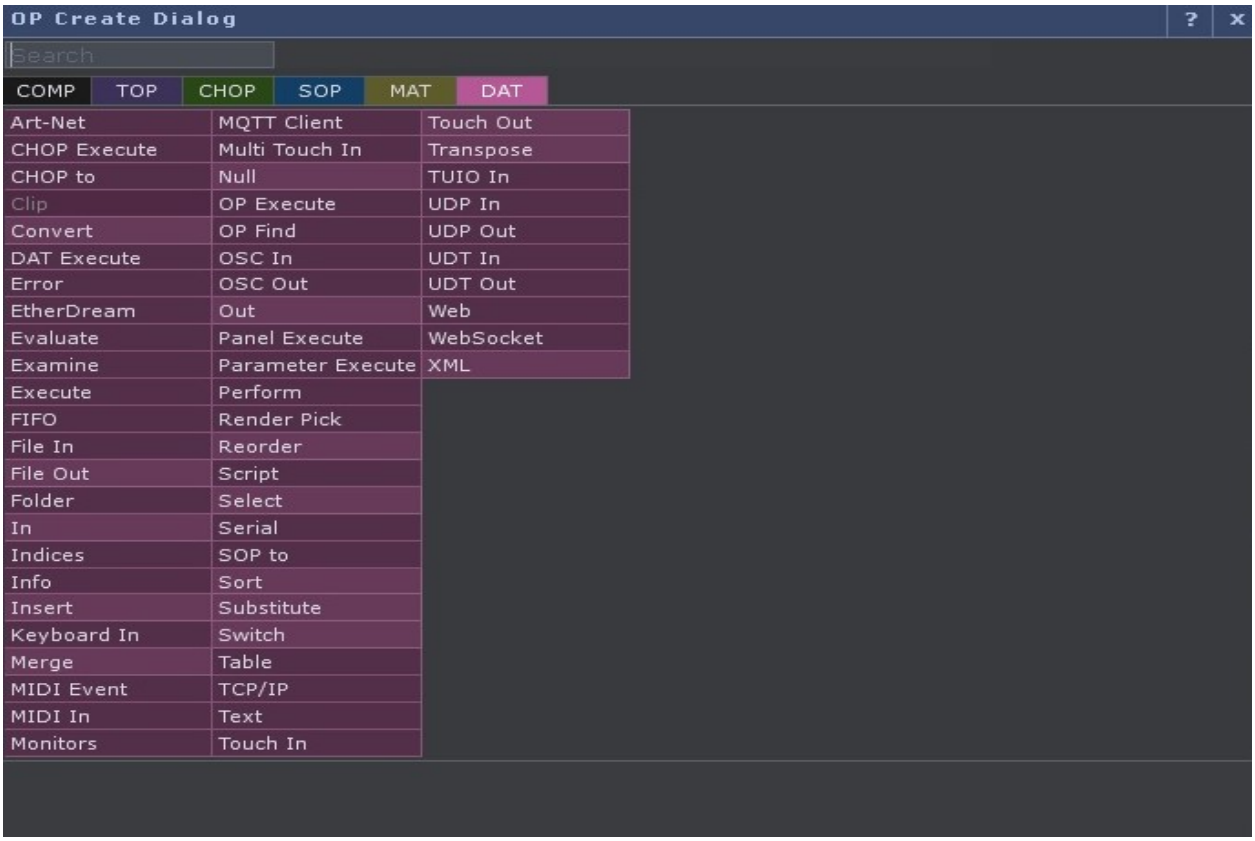


Figure 58: TD: DAT Operators

Within each operator family, "generator" operators have 0 inputs and create data, and "filter" operators have 1 or more input and filter data. Each operator family is a unique color. Only operators of the same family (color) can be “wired” together. Many operators have parameters that are references to operators in other families: such as “Links”. Also "Exporting” flows numeric data from CHOPs to all operators.

Figure 59: TD: Nodes

Creating Operators: To add new operators to a network, the OP Create Dialog must be opened. The OP Create Dialog can be opened by pressing the <tab> key, double-clicking on the network background, clicking the "+" button in the Pane Bar, selecting “Add Operator” from the right-click menu in any network, or by right-clicking on the input or output of another operator.

Adding Multiple Operators: Using the <ctrl> key allows multiple operators to be added to the network. First, we open the OP Create Dialog. Then press and hold the <ctrl> key while making our selection of operators. As operators are clicked on they will be added to the network.

Wiring Multiple Operators together: Using the <shift> key allows multiple operators to be wired together and added to the network. First, we open the OP Create Dialog. Press and hold the <shift> key and as we select operators from the menu they will be wired together and added to the network. When selecting an OP of a different family type, a new branch of connected operators will be started.

Converting Data between OP Families: It is possible to convert data between different Operator families using the following conversion operators. For example, geometry can be converted into a DAT list of point positions using the "SOP to DAT "operator, or convert a TOP image's pixel values into red, green, and blue channels in CHOP using the "TOP to CHOP" operator.

TouchDesigner is an environment with extreme depth, and many potential pitfalls, as many tasks will require due diligence in regards to time and effort. In spite this, TouchDesigner is node-based. Consequently, instead of opening a text document and typing line after line of code, TouchDesigner's graphical interface is used to make applications out of nodes. Each node, or Operator in TouchDesigner, performs a specific, small, and granular action. For more complex tasks, a network of operators will work ensemble. Even though there are many node-based programming languages in existence, such as Cycling 74's Max/MSP, what sets TouchDesigner apart is it's visual nature. Everything in TouchDesigner has a visual counterpart and items like text, control data, audio, videos, and more, is visualized through each and every operation that is performed. This is unlike any traditional programming, and even node-based, language, but it is what makes TouchDesigner an innovative environment to work with. Visualizing the steps involved in performing complex tasks, undoubtedly simplifies the process of learning and producing. "As researchers here we can't just talk about our work, we have to physically build and present functional demos to get the kind of critical feedback from mentors we need to really move things forward. I can't think of a better tool to quickly prototype ideas than TouchDesigner 088. I've used it here for everything from running an immersive, real-time interactive visual environment with 10 HD streams, to orchestrating multiple low-DOF robots and animating shape changing interfaces. I use TouchDesigner 088 because it combines a forward-thinking product architecture and vision I've always believed in, extensibility for those custom jobs and a friendly and knowledgeable user community." - David Robert, MIT Media Lab, Cambridge MA

3 Requirement Analysis

3.1 Introduction

Scene Wizard is a Software Development Kit designed for Unity3D Game Engine. Currently, there are custom made state-of-the art software that utilize Depth cameras in order to include human interactions with visuals, such as HeavyM, which we saw in the previous section. However, in order to access the program's features one needs to suffer a collateral damage of 44euros. Simultaneously, software like Kinect Projector Toolkit, does calibrate the scene allowing an automated Projection Mapping accurately aligned, but does not provide the possibility of physically interacting with visuals. On the other hand, RoomAlive is a development kit mainly targeting game designers. RoomAlive makes use of Unity3D's modular plugin framework and scripting interface, game art assets can be easily imported from external content and behaviors can then be applied to a game object using a C# scripting interface, e.g., how to react to gun fire. Thus, it requires quite high programming skills and game development knowledge. Our main goal with SceneWizard is to automate the process of adding interactivity effortlessly and pricelessly. Using Microsoft's Kinect SDK for Unity we will provide a database of gestures that are ready-to-use.

Additionally our development kit should be a system that hosts Spatial Augmented Reality visual effects that are being generated by state-of-the-art video mapping software, such as MadMapper, or AfterEffects, as well as provide an interface that links them. The interface should also include editor tools that ease the image registration problem of Projection Mapping.

As a result, we created a Tool, that can be utilized in a Unity Platform Environment, which allows users to compose, produce and perform Interactive Projection Mapping, as form of controlling the Visuals' appearance on the Time-line of a Real-Time Performance. The user can easily and conveniently recreate a virtual replica of the real scene in Unity's virtual 3D Space, by using our Tool's UI, with respect to the geometric relationship between the Projection's Components. Furthermore, the user can handily align the virtual surface object matching the real surface's shape, despite the surface being in an abstract shape, as a result of an oblique projection, which is usually the case. This can take place, after the set-up of all the

surfaces and desired scene-versions, by entering PlayMode with a Button from SceneWizard's UI and before the actual Performance. Also, the user can potentially create multiple versions of a specific scene, in order to display more than one set of visuals and synthesize a visual story. Thus, during the performance, the user can (possibly with the help of an Assistant, if the creator is also the performer) control the spawning of each version of the scene with an Input from the keyaboard.

Additionally, the user can spontaneously associate each surface he/she desires with Initial-Following Visuals and Gestures, which will be rendered and projected on that specific surface before and after the corresponding gesture's occurrence. Moreover, the resetting of the scene's initial state can be controlled via an Input from the keyboard, so that the gestures can be performed again. The Gestures will be provided by our custom Gesture Database, we have pre-trained and composed for the Kinect Depth Sensor, which is ready-to-use. The Visuals can be provided via the Spout Protocol we integrated into Unity, as means of communication between the significant Software and share video content amongst them. We will thoroughly explain, this process of defining the Visuals' names in the Implementation Chapter 4.

A very robust tool, provided from SceneWizard is the possibility of saving each project's settings (Scenes-Surfaces-and the Correlation between them) as stand-alone Asset Files, which can be transfered between various projects in Unity, in order to maintain Source Control or just simply use pre-created files in a different project.

In this section we will outline what requirements we set for completing our project and then we will outline the hardware and software we used in our development. Additionally, we will discuss about the Use Cases framework of our tool, further in this chapter.

3.2 System Requirements

Initially, we want Scene Wizard to be a complete project any designer or artist can adjust to his needs. As such Scene Wizard must:

- •Be a fully importable UnityPackage
- •Provide possibility to create and save project - performance
- •Provide possibility of editing and saving scene.
- •Providing a collection of gestures.
- •Have simple Demos to demonstrate the use of the tool.

□ As a Gesture Detector the tool must also contain:
- •Scripts that act like Kinect Manager, to control the motion sensor and poll the data streams.
- •Scripts that act like Gesture Manager, to initialize the gesture database and monitor the gesture events.

☐ In addition to assist the SceneWizard tool optimization we need:
•An editing tool to handle the image registration problem of Projection Mapping.

•Scripts that act like Gesture Manager, to initialize the gesture database and monitor the gesture events.

•Scripts that synchronize Unity's Edit and Play Mode, in order to maintain the settings for the real-time performance.

•Scripts that enable the user to control his scenes.

☐ Finally we need fluid support realtime videosharing framework. Thus we also need:
•Scripts that enable videosharing in between Unity and a VideoMapping Software.

•Scripts that extend the videosharing abilities, in correlation with the gesture events.

3.3 Platform Information

Scene Wizard was fully developed in Unity. Unity is a cross-platform game engine developed by Unity technologies, first announced and released in June 2005 at Apple Inc's Worldwide Developers Conference as an OS-X exclusive game engine. Unity provides intuitive tools for designing awesome 3D content, as well as cross-platform publishing, millions of ready-made assets in the Asset Store and a strong online community. We chose Unity, as one of the greatest advantages is its flexibility to deploy projects on multi-platforms like Windows, iOS and Android. As of 2018 the engine has been extended to support 27 platforms. In addition to that, Unity works effectively as the base for linking Kinect and a VideoMapping Software withing the gesture recognition concept. Furthermore, nowadays, more and more artists and designers choose Unity, to incorporate live visuals into their performances. Unity's real time rendering engine enables amazing visual fidelity. From luminous day, to the gaudy glow of neon signs at night; from sunshafts, to dimly lit midnight streets and shadowy tunnels–create an evocative atmosphere that enthralls players. In addition to that, Unity's real-time renderer and automation tools allow, complex scene set-ups, using multiple cameras – multidisplay output, multiple scenes, as well as creating an app for a specific performance. As a result, Unity is an on-growing tool for VideoJockeys, as it is great for creating generative art, simulations, interactive experiences and games. Hence, Unity sufficiently gratifies our needs. The version we used was Unity 2018.1. This version of Unity was selected due to compatibility issues with our version of Spout Protocol and earlier or newer unity releases.

For the videosharing settings, Spout was imported in Unity, as well as Spout for Windows was installed on our Operation System in version 2.2006, which is the latest version published in the official site. Alongside, for accessing Kinect's monitor settings we imported Kinectv2 Unity Plugin in version 2.0.1410. This included the following plugins:

☐ Kinect.2.0.1410.19000 That's our plugin, along with all the scripts needed to build a Kinect-enabled Unity application.

☐ Kinect.VisualGestureBuilder.2.0.1410.19000 That's our plugin, needed to implement the Machine Learning algorithms that determine gesture recognition events.

Note that, the plugins require UnityPro, but the APIs worked fine also on the free version of Unity. For future releases of Unity, it is recommended to check, whether the imported packages have been updated, in order to maintain accessibility and functionality.

Inter alia, in order to perform our Tool's Demonstration, we will need a generative art application, preferably a Video Mapping tool, as these tools are in vogue for Projection Mapping amongst artists and designers, as well as developers. By virtue of presenting our tool, we need to generate visuals to implement on our project. While performing an Interactive Projection Mapping with SceneWizard, the user will simultaneously run both Unity and the Video-Mapping Software and share video data between them during Run-time. After inspecting the software for Projection Mapping, we concluded on using TouchDesigner, as it fits best for our needs. Most importantly, it is free to use for streaming up to 1280x1280 Resolution and does a great job in creating real-time video effects.

To develop Scene Wizard an MSI GP62M 7RDX Series was used, with Windows 10 Home OS(64-bit). This laptop has a 4-core IntelCore i7 CPU @ 2.80GHz, 8GB RAM, DirectX12 SDK, GeForce GT 1050.

Testing and Demo was done with BenQ W1080ST Projector. Additionally a scene was built with cardboard boxes, which had to be covered with white sheet paper, in order to overlay the deeper color of the boxes and reflect the projector's light for efficiently.

3.4 Use Cases

Scene Wizard was primarily designed for Video Jockey designers or artists that are keen on Projection Mapping. A video jockey (abbreviated VJ or sometimes veejay) is an announcer who introduces music videos and live performances on commercial music television stations such as VH1, MTV, Channel V and Much Music. The term "video jockey" comes from the term "disc jockey", "DJ" ("deejay") as used in radio. Music Television Network (MTV) popularized the term in the 1980s. However, anyone with a developing background (or even without), that is interested in manipulating visuals via human interactions, is able to use, experiment with and expand this tool. Finally, we should consider the End-users' use cases. As an End-user we define the person who will have the chance to experience the interaction and visualize its results. Below we will first present the designers'/artists' use cases inside Unity's environment, followed by the developer's use cases in the development environment and finally the End-users' use cases in the real environment.

3.4.1 Designer's/Artist's Use Cases

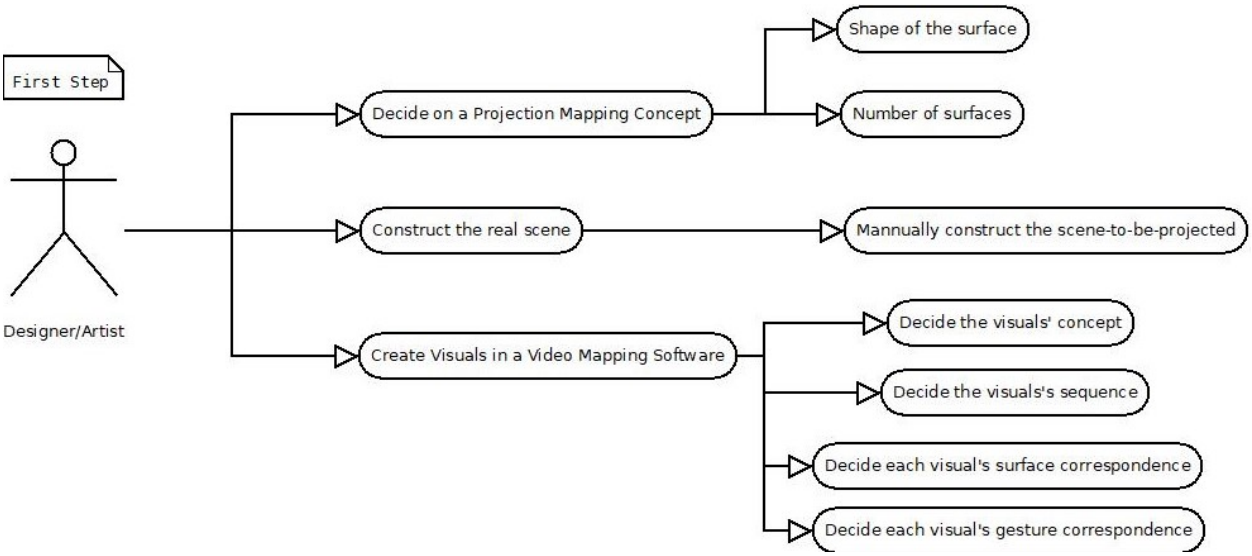


Figure 60: Designer's/Artist's Use Cases: Step 1

Before creating the virtual scene of the performance in Unity, the first step of the designer is to determine the intrinsics and extrinsics of the procams system, as well as concluding on a Projection Mapping concept with a specific scene, visuals and story. There are cases where, for example, an artist is invited to project visuals on a ready set-up scene, other cases include deciding on a scene according to the quality of the projected content, and finally there are cases where the artist must decide on both. The size and complexity of detail of the images that will be projected will have a direct correlation to the cost. To maximize cost, creativity and collaboration, however, the artists should plan the Projection Mapping as far in advance as possible. In addition to that, creative uses of projection that are small or simple in design should be planned at least six weeks before the event. For those that are more complex, the user should consult with his/her team. After creating the visuals, each visual should have a correspondence Spout Sender Name broadcasted from the Video Mapping Software, that will determine the correct matching of the visuals' sharing content via Spout Protocol. The construction of the real-scene needs to take place in the beginning, so that the designer already has the geometric concept to work on.(See Figure:60 above.)

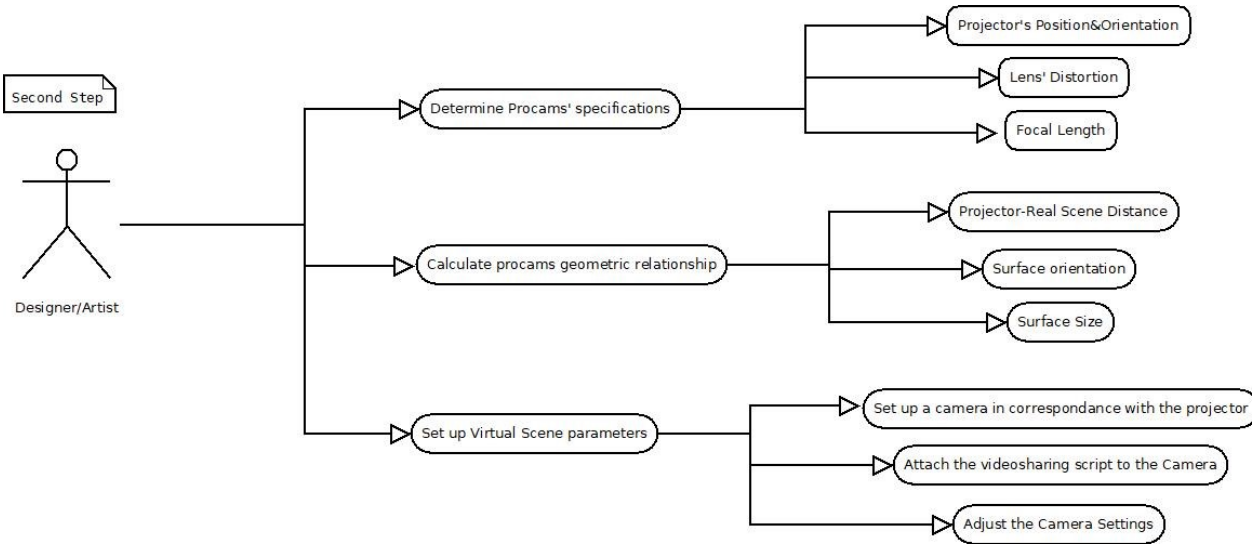


Figure 61: Designer's/Artist's Use Cases: Step 2

The task also includes defining the projector's specifications, such as the resolution of the projected image, the throw distance and throw ratio. The last two parameters are needed, in order to specify the horizontal field-of-view of the virtual camera that "watches" the virtual representation of the scene: Throw Ratio(of the projector) = Distance(between projector and surface)/Width(of the Image projected), for each foot of image the projector needs to be throwRatio feet away. Afterwards, the designer needs to calculate the surfaces' orientation and size. These parameters will specify the setting up of the virtual scene that follows after the setting up of the real scene. Scale and unit of measurement play a very important role in a believable scene. In many "real world" setups, it is recommended to assume 1 Unity unit = 1 meter (100cm) as many physics systems assume this unit size. To maintain consistencies between Digital Content Creation 3D software and Unity it's always a good idea to validate imported object scale and size. DCC 3D software such as 3dsMax/ Maya/ Blender/ Houdini, etc have units settings and have scale settings in the FBX export configuration (consult each software manual to configure). In general setting these tools to work in cm and export FBX at automatic scale will match the scale in Unity import properly. However it's always reassuring to confirm it matches whenever starting a project. Finally, according to these parameters the designer can set up the virtual scene defining the surfaces' positions and orientations in correspondence with the camera. The camera must contain the imported video-sharing component script (Spout Sender). This component will allow the camera to share its content to the video-mapping software, in order to avoid Unity's Game View task-lines. Furthermore , the camera's settings need to be adjusted, in order to produce an aligned image.(See Figure:61 above.)

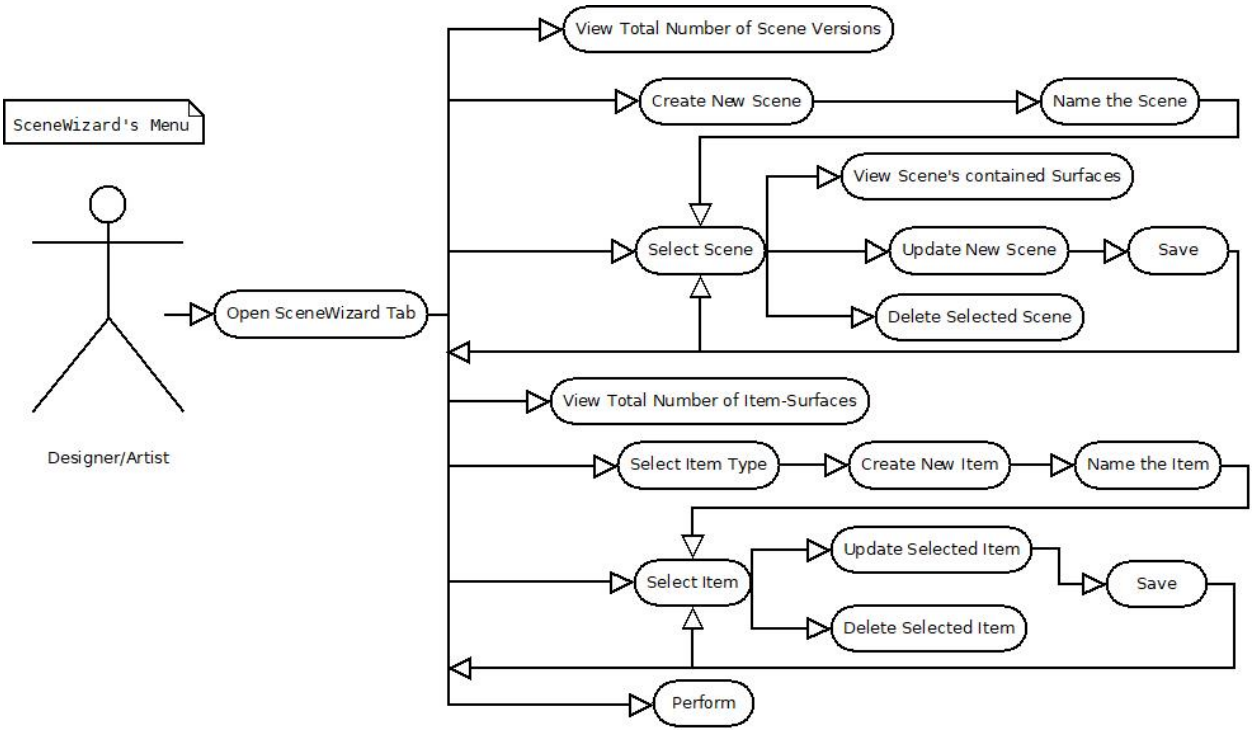


Figure 62: SceneWizard's Main Menu Use Cases

After determining the concept and setup of the project, as well as adjusting the aforementioned scene's parameters, when launching the SceneWizard's Main Menu(See Figure:62 above.) , the user is introduced to SceneWizard's Editor Window Interface, from where he can perform all the necessary actions.

Pressing SceneWizard Tab
If the SceneWizard Tab is pressed, the Main Menu of Scene Wizard is launched with the name "All my Assets".

Pressing the "Create New Scene" Button
If the "Create New Scene" Button is pressed, a new .asset file is created in the "SceneItemsLists" folder ready to name. And the project view of Unity shows the file in its folder. This file represents the total surfaces of a specific version of the scene.

Pressing the "Update Selected Scene" Button
If the "Update Selected Scene" Button is pressed, a new editor window pops next to SceneWizard's Main Menu editor window (InventoryScene Editor). This window allows user to specify and select the Scene's surfaces.

Pressing the "Save" (in InventoryScene Editor Window) Button
After Updating the Scene, if the "Save" Button is pressed, the InventoryScene Editor Tab is closed and the user returns to Main Menu (All my Assets Editor Tab). This button also saves any modifications performed on the selected Scene within the InventoryScene Menu.

Pressing the "Delete Selected Scene" Button

If the "Delete Selected Scene" Button is pressed, an extra editor window pops up with the title "Deleting..." and two Buttons: "Yes" and "No", to reassure the deletion of the selected Scene. If the user presses the "Yes" Button, the corresponding .asset file of the selected scene will get erased from Unity's Assets appropriate folder (SceneItemsLists folder) and the popped "Deleting..." Window closes. If he presses "No" Button, the popped "Deleting..." Window closes. The user needs to select a scene before pressing the button, by double-clicking on the preferred Scene's name field. Otherwise, nothing will get deleted.

Pressing the "Select Type of Item before Creating" Button
If the "Select Type of Item before Creating" Button is pressed, a dropdown menu appears with options of the available types of surfaces the user can create. These include a Cube, Sphere, Plane, and also a CustomType surface. The latter would be chosen if the user has a custom model of a surface, created outside of Unity in any DCC 3Dsoftware like Blender. Note that CustomType needs to be able to access the MeshFilter Component of the imported model. Thus the form of the imported model must be specific.

Pressing the "Create New Item" Button
If the "Create New Item" Button is pressed, a new .asset file is created in the "Items" folder ready to name. And the project view of Unity shows the file in its folder. This file can be included in any of the Created Scenes.

Pressing the "Update Selected item" Button
If the "Update Selected Item" Button is pressed, a new editor window pops next to SceneWizard's Main Menu editor window (ItemInspector Editor). This window allows user to specify the surfaces parameters, in order to customize the item with an initial visual, a gesture, a triggered visual, as well as the scene setup of this specific Item-Surface (position, rotation, scale).

Pressing the "Save&CloseTab" (in ItemInspector Editor Window) Button

After Updating the Item, if the "Save&CloseTab" Button is pressed, the ItemInspector Editor Tab is closed and the user returns to Main Menu (All my Assets Editor Tab). This button also saves any modifications performed on the selected item within the ItemInspector Menu.

Pressing the "Delete Selected Item" Button
If the "Delete Selected Item" Button is pressed, an extra editor window pops up with the title "Deleting..." and two Buttons: "Yes" and "No", to reassure the deletion of the selected Item. If the user presses the "Yes" Button, the corresponding .asset file of the selected scene will get erased from Unity's Assets appropriate folder (Items folder) and the popped "Deleting..." Window closes.. If he presses "No" Button, the popped "Deleting..." Window closes. The user needs to select an item before pressing the "Delete Selected Item" Button, by double-clicking on the preferred Item's name field. Otherwise, nothing will get deleted.

Pressing the "Additional Instructions" Button
If the "Additional Instructions" Button is pressed, a new editor window pops, with all the necessary information, to guide the user during PlayMode.

Pressing the "Perform" Button
If the "Perform!" Button is pressed, Unity exits Edit Mode and enters Play Mode.

Pressing the "Contains Items" field If the "Contains Items" field is pressed, a drop-down list of the contained surfaces of the selected scene pops, which informs the user which objects are contained in each

version of the scene.

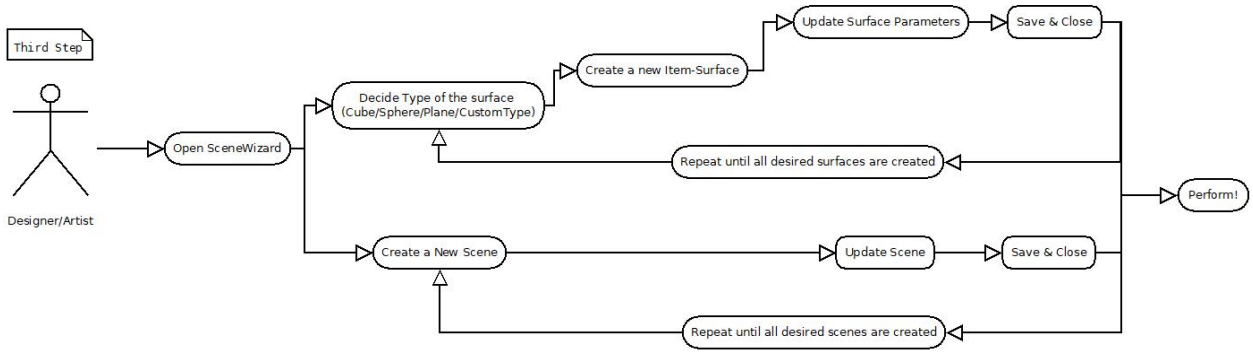


Figure 63: Designer's/Artist's Use Cases: Step 3

The next step (Step3) is, using SceneWizard, to create a number of surface-items that will represent the desirable "masking" of the different elements of the projection scene, in order to register the image for each different surface. The user has the possibility to decide between creating a cube, sphere, plane, or custom type of a 3D surface. After creating and naming the Item-Surface, the user needs to update its settings, in order to correspond to the desirable scene and effects. To successfully update the item, the user must save it. Often, the scene-to-be-projected contains a various number of surfaces, presumably of different shape and scale, thus the process needs to be repeated until all the surfaces have been covered. Finally, the user should create a scene, which will then need to populate with the preferred items that have been created before. The case will most likely require multiple versions of one scene, as the user has the ability to iterate through the scenes and control which scene to perform. Thus, the latter task will be repeated until the desirable number of scene versions has been created. The last step, is to actually initialize the real-time performance.(See Figure:63 above.)

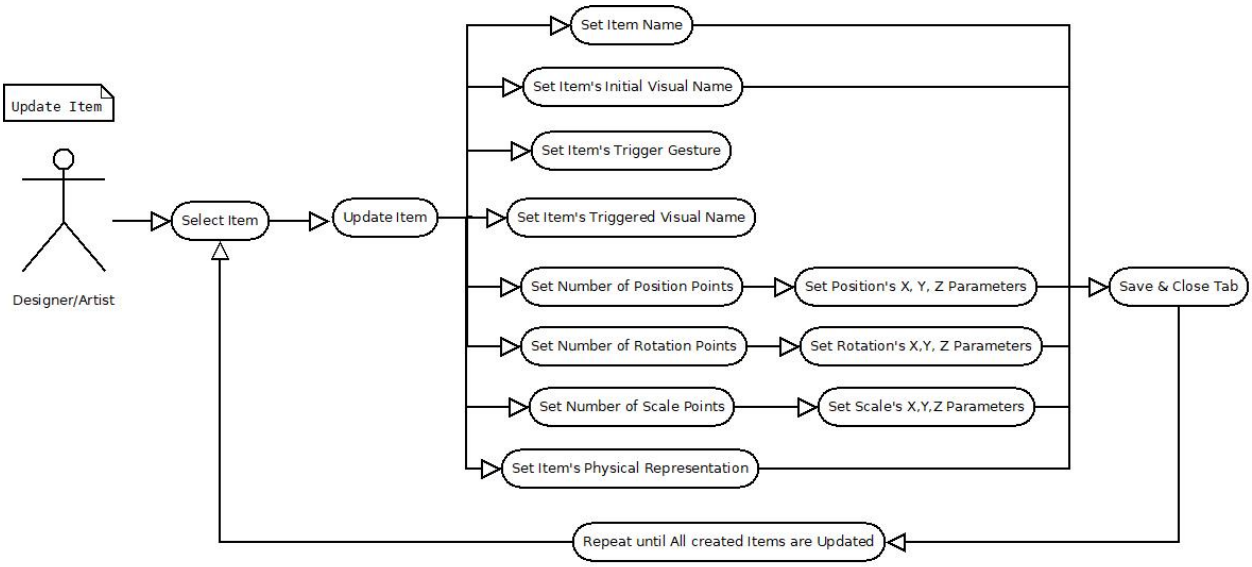


Figure 64: Designer's/Artist's Use Cases: Updating an Item

After selecting the item, to update, the user needs to specify its name, preferably use the one initially given on creation. Additionally, the user needs to type the name of the visual that will primarily "mask" the Item-Surface. To continue with, the user will need to choose a gesture from the provided gesture database, as well as type the name of the visual, which will get trigerred on a gesture event and switch the item's surface "mask" to the one given on this field. The number of position points map the number of surfaces that need to be displayed with the same visual masking, each position point corresponds to each rotation and scale point. Consequently, the number of position, rotation and scale points must to match. This process will be repeated, until all the items that have been created, contain the appropriate data for representing the scene.(See Figure:79 above.)

ItemName -Text field
The user types the desirable name for the Item-Surface.

Initial Visual Name - Text field
The user types the matching name of the visual, that will initially "mask" this specific Item-Surface, when entering Play Mode. The name needs to be the corresponding name of the specific visual, in the Video Mapping software the users is using to create Visuals.

Select Gesture from Gesture Database - Drop-down Button and Text field
The user selects a gesture from the Drop-down Button. If the Button is pressed a list with the available gestures, contained in the provided database, drops.

Triggerred Visual Name - Text field
The user types the matching name of the visual, that will "mask" this specific Item-Surface, right after the latter gesture is performed. The name needs to be the corresponding name of the specific visual, in the Video Mapping software the users is using to create Visuals.

ItemType - Label field
This label field signifies what is the type of the item, the user is currently updating.

Position Points - Size Integer field and Vector3 float field
Position points include a Size Integer field, where the user types an integer that signifies the number of items-surfaces that will appear in the scene with the specific Initial Visual Name, Gesture, and Triggerred Visual Name. After typing an Integer, the user must hit enter, to apply(If, for example he types the number 2, after pressing "Enter", two Vector3 fields will appear under the Size Integer field). On Vector3 field, the user can type the corresponding X, Y, Z float points that will determine each item's position.

Rotation Points - Size Integer field and Vector3 float field
Rotation points include a Size Integer field, where the user types an integer that signifies the number of items-surfaces that will appear in the scene with this specific Rotation. The Rotation points correspond with the Position points, thus their sizes must match, since the given rotations will be applied on each item, given in the Position points. On Vector3 field, the user can type the corresponding X, Y, Z float points that will determine each item's rotation.

Scale Points - Size Integer field and Vector3 float field
Scale points include a Size Integer field, where the user types an integer that signifies the number of items-surfaces that will appear in the scene with this specific Scaling. The Scale points correspond with the Position and Rotation points, thus their sizes must match, since the given scales will be applied on each item, given

in the Position points. On Vector3 field, the user can type the corresponding X, Y, Z float points that will determine each item's size.

PhysicalItem – Object field
PhysicalItem is an Object field, from which the user can select a potential custom physic model for this Item-Surface. If this field is pressed (click on the circle, on the right side of the field), a "SelectGameObject" window Tab, from where the user can select a prefab model, from the ones contained in his project folders.

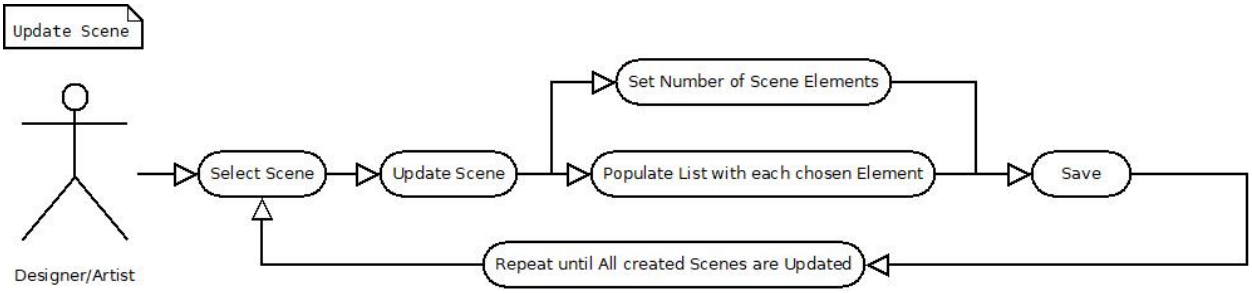


Figure 65: Designer's/Artist's Use Cases: Updating a Scene Version

Subsequently, to update the scene, first the user needs to select it and afterwards set the number of surfaces that will represent the current scene. Each element of the scene corresponds to the previously created items. Finally, the scene must be saved and the process should be repeated until all the created scene versions have been updated.(See Figure:65 above.)

When Updating a Scene, after the User presses the appropriate Button, the InventoryScene Tab opens, where:

InventoryList - Size Integer field - Elements
InventoryScene Tab includes a Size Integer field, where the user types an integer that signifies the number of items-surfaces that will appear in the scene with this specific SceneVersion. After typing an Integer, the user must hit enter, to apply(If, for example he types the number 3, after pressing "Enter", three Elements will appear under the Size Integer field, each corresponding with an Object Field).

InventoryList - Object field
Each element of the Scene (InventoryList) is represented with an Object field, from which the user can select an Item-Surface, to populate the Scene. If this field is pressed (click on the circle, on the right side of the field), a "SelectGameObject" window Tab, from where the user can select an Item-Surface (asset file), from the ones contained in his project folders.

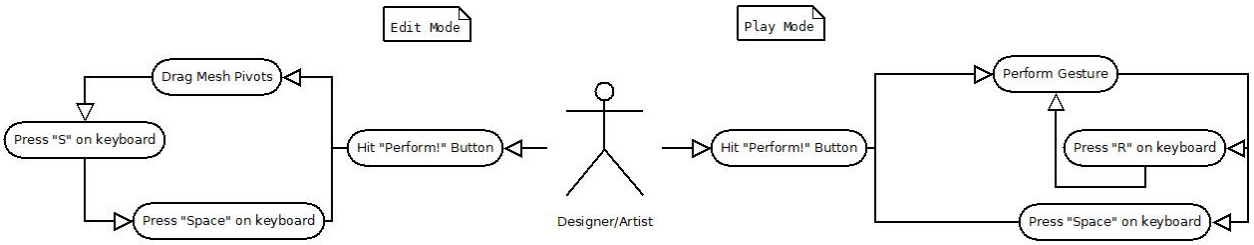


Figure 66: Designer's/Artist's Use Cases: Edit & Play Mode

Before, actually performing the show the user should solve the image registration issue. After pressing the "Perform!" Button, the surfaces appear in the scene, but it is quite unlikely that the surfaces will be registered perfectly and aligned, as there is always the proximity error. Thus, the user can adjust the image-Item-Surface, by dragging and dropping with the mouse cursor the blue pivot points displayed on the item. After adjusting all the surfaces the user, focused on the Game View, needs to hit the "S" key, on the keyboard, in order to save the modified "Masks" (Meshes) of the surfaces. The "Masks" are also saved as, .asset files in a folder called "Meshes". This process, needs to take place for all the Scene Versions that the user wants to perform. So as to succeed this, the user needs to iterate through all the Scene Versions, before performing on real audience, via pressing the "Space" key on the keyboard. In addition to that, when the user is on Performance, he can hit the "R" key willingly, in order to reset a Version of the Scene, to the initial state, before the Gesture Event happened, so that the Gesture can be performed again and for as many times he wants.(See Figure:66 above.)

Pressing the "Perform!" Button
If the Button "Perform!" is pressed, the project exits Edit Mode and enters Play Mode. The first Version of the Scene, as seen in the SceneWizard's List of All Scenes, appears.

Pressing the "S" key, on keyboard
If the "S" key is pressed from keyboard, while the project is on Play Mode, all the "Masks" of the current appearing Scene Version, will be saved in the Project's folder "Meshes".

Pressing the "Space" key, on keyboard
If the "Space" key, is pressed from keyboard, while the project is on Play Mode, the next Version of the Scene, as seen in the SceneWizards List of All Scenes, will appear.

Pressing the "R" key, on keyboard
If the "R" key is pressed from keyboard, while the project is on Play Mode, if the end-user has already performed the gesture and the visual has switched, then the Item-Surface will reset, returning to its initial state-visual.

Dragging pivots with Mouse
When the user sees the blue pivot points representing each items triangles' edges, he has the ability to click with the Left Mouse Button, on each blue point and drag the point, to adjust the covering area of the Item-Surface, as long as the mouse is clicked. When the user releases the mouse, the deformation of the surface stops at the point the mouse was released, displaying a manipulated surface.

3.4.2 End-User Use Cases

The End-User's use cases are slightly different from the designer's/artist's. We consider these use cases from the moment the primer user (performer-designer-artist) performs the Projection Mapping interactive show on an audience. In this case there are two assumptions:

- ⋆ First, that the artist performs an interactive show, on a participating audience. This signifies that an artist has previously used Scene Wizard to create an interactive Projection Mapping performance and controls the performance, allowing the audience (single or multiple End-users) to participate and get involved.
- ⋆ Second, that the artist performs the show, participating himself, to present the interactions and captivate the audience.

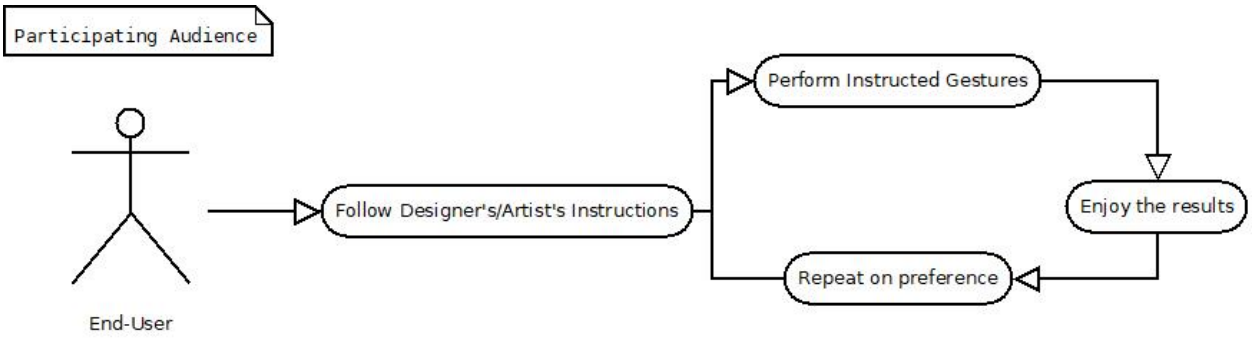


Figure 67: End-User Use Cases: Participating Audience

As an audience, the End-user must listen and follow the instructions given from the artist that performs the project created on Unity's SceneWizard. The artist will give the End-user the appropriate instructions, on what gestures to make and how. Presumably, the artist will not give detailed instructions of the gestures, rather than let the user explore the performance by realizing himself how he can interact with the visuals, providing a more fascinating and stimulating experience. The End-user can then perform the gestures repeatedly, in consultation with the artist, in order to entertain or educate himself.(See Figure:67 above.)

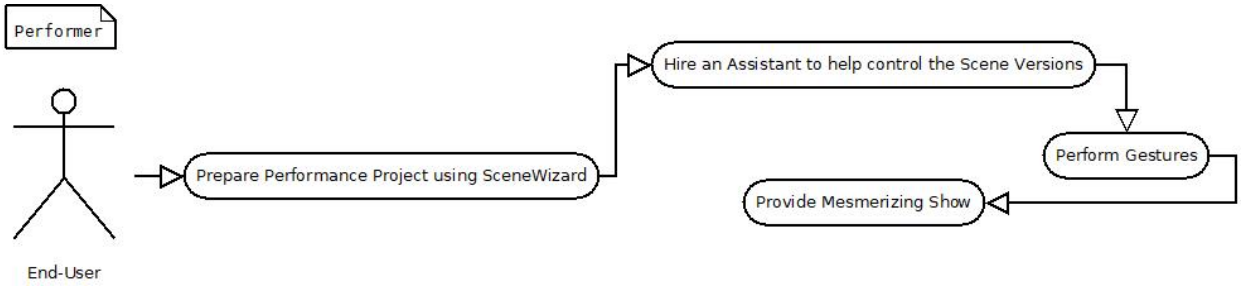


Figure 68: End-User Use Cases: Performer

As a performer, the End-user must prepare his performance utilizing our tool in Unity. When creating multiple versions of the scene the performer, will also need to hire an assistant, who will support the performance and control the iterations through the different versions of the scene. Finally, the performer

will actually perform the Interactive Projection Mapping Show, using his body movements to manipulate the projected visuals in an astonishing magical way.(See Figure:68 above.)

3.4.3 Developer's Use Cases

Even though SceneWizard is primarily a tool for designers and artists of Projection Mapping, it is also designed as tool for Interactive Applications, using Kinect. As a result, we not only have in mind use cases for the designers and artists that use SceneWizard, but also use cases for Developers-Users, that can expand the tool according to their needs, following our architecture.

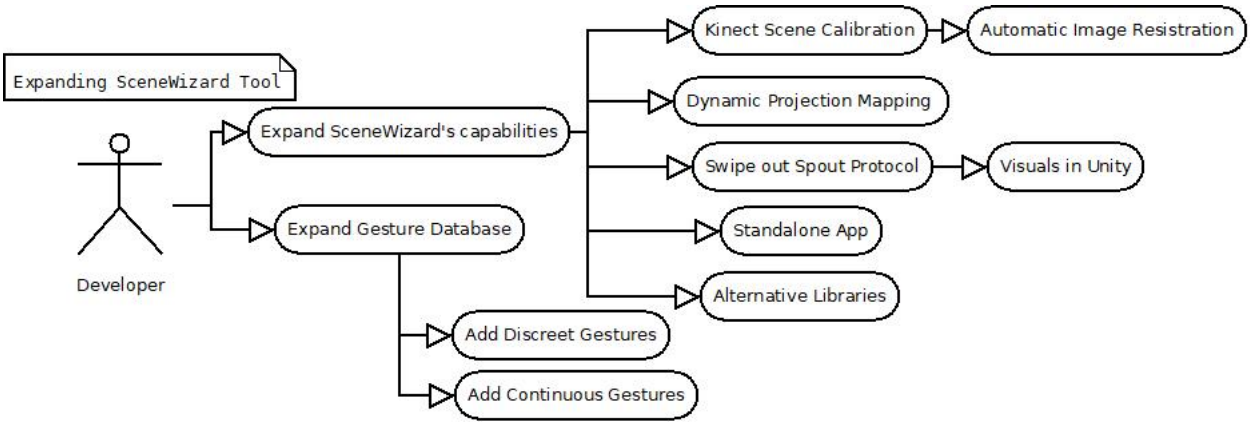


Figure 69: Developer's Use Cases

There are numerous ways, in which this tool can be broadened, on what concerns using the Kinect Depth Sensor. We have seen in the state-of-the-art applications that use the sensor to calibrate the scene, expunging the Image Registration problems automatically. Kinect can also be used to track any moving components, such as objects of the scene, besides the basic User-Tracking. Furthermore, Graphic's designers and developers can entirely use Unity to create visuals, swiping out the need of Spout Protocol, when developing the visuals in Unity. Additionally, one can further expand the provided database, enriching it with additive discreet or continuous gestures.(See Figure:69 above.) We will thoroughly discuss the possibilities of Future work on our project in Chapter 6.

4 SceneWizard - User Interface

4.1 Introduction

In this section we will analyze the aforementioned Use Cases in conjunction with presenting the UI elements, that can be utilized, in order to achieve each Use Case. SceneWizard's User Interface is everything that the user can see and interact with, in order to utilize our tool. The Interface we built for SceneWizard includes pre-built UnityEditor UI components, such as structured layout editor windows and display areas, as well as Editor UI controls, like buttons and notes, that guide the user into SceneWizard's functionality.

4.2 Main Window

The Main Window of SceneWizard can be opened by clicking on the homonymous custom tab that is being generated along with Unity's classic Tool bar, after importing our UnityPackage, as we can see in the figure below:

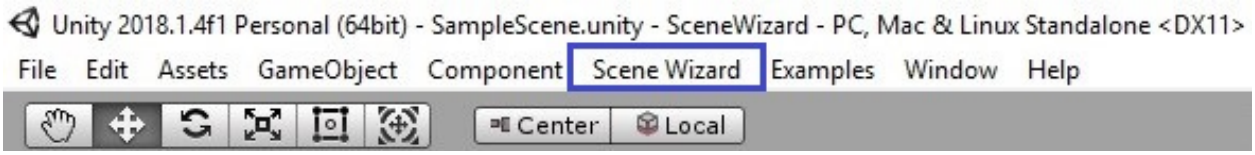


Figure 70: SceneWizard option in Toolbar of Unity

After clicking this Tab Option the main window of SceneWizard appears, where the collections of our data assets are demonstrated and can be further manipulated. As we have already mentioned in the Use Cases Chapter, the user can create new Items-Surfaces, by selecting a Type first, as well as create new Scene

Versions, via the Main Window. The Items and Scenes can be further customized, by clicking the appropriate Button, or completely deleted from the project. Furthermore, the user can recur the "Instructions" Button, to inform him/herself on the possible actions. Finally, the user can enter PlayMode, by clicking the "Perform!" Button. So all the functionality of our tool is summarized on our main editor window. An example snapshot(Fig:71) below demonstrates the main window of SceneWizard, where we can see with the same row:

•The total number of our Scene's Versions, demonstrated with an Integer.

•A white-colored sub-note, which guides the user on how to select a scene version for further manipulation-customization.

•A list with all our Scene Versions currently contained in our project's folder "SceneItemsLists".

•A list corresponding with the Scene Versions list, which demonstrates which Items-Surfaces each Scene Version contains at the moment.

•Three Buttons in a row, significant for the SceneVersion manipulation (Create New, Update Selected, Delete Selected).

•The total number of our Items-Surfaces, demonstrated with an Integer.

•A list with all our Items-Surfaces currently contained in our project's folder "Items".

•A second white-colored sub-note, which guides the user on how to select an Item-Surface for further manipulation-customization.

•A list with all our Items-Surfaces currently contained in our project's folder "Items".

•A Drop-down Button to select the Type of the Surface desired before creation.

•Three Buttons in a row, significant for the Item's-Surface's manipulation (Create New, Update Selected, Delete Selected).

•A Button to view additional instructions concerning SceneWizard's functionality.

•An informative Box to guide the user on hitting the "Perform!" Button, in order to begin the Performance.

•A Button to enter the "PlayMode" of Unity and begin Performing Interactive projection Mapping.

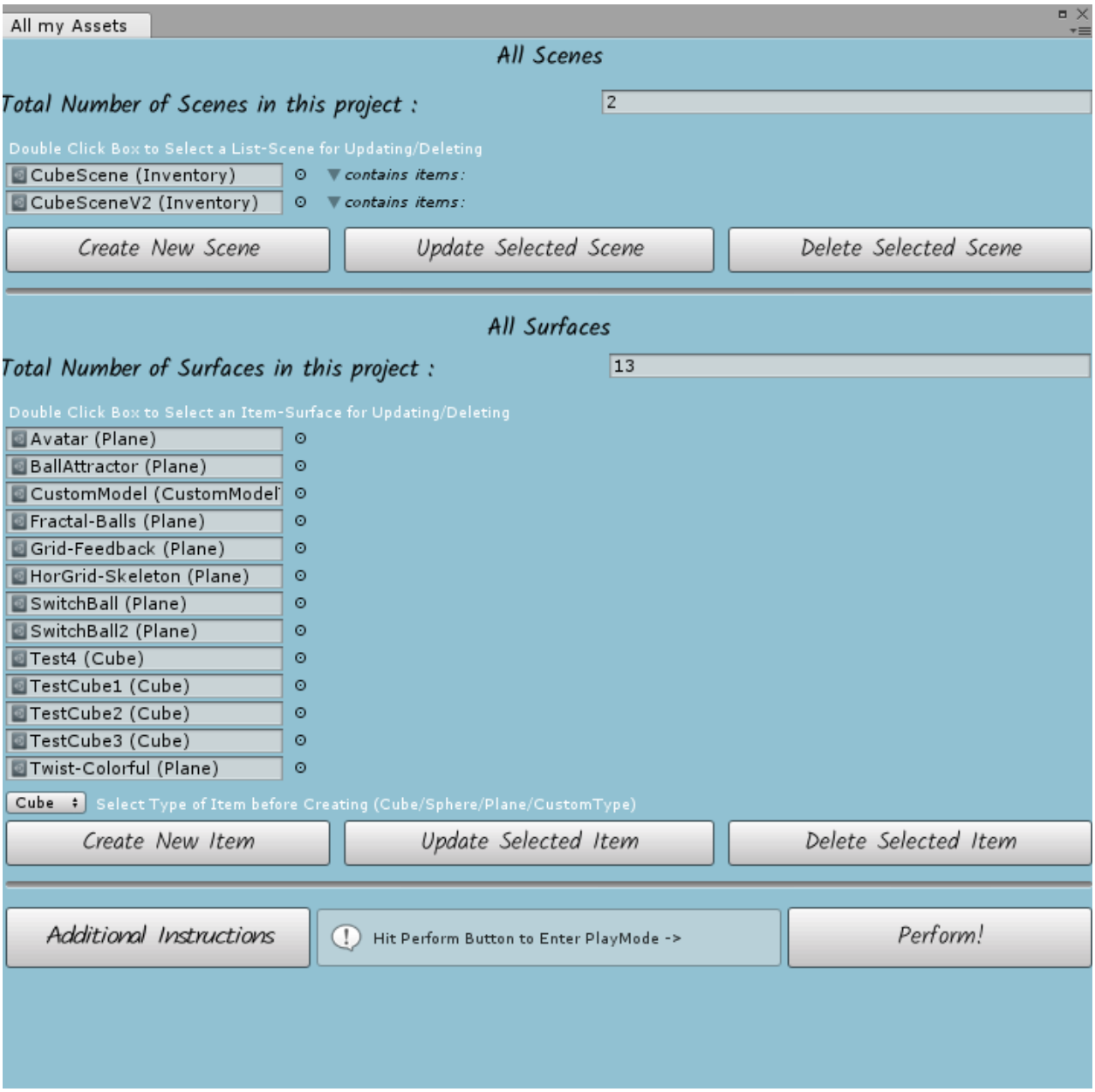


Figure 71: SceneWizard Main Window in Unity

Below we summarize the procedure and flow of the basic steps required to set-up a scene before the Performance, in a Figure(Fig:72) using the Main Window of SceneWizard, as well as the custom Inspectors, that assist in customizing the Items-Surfaces and Scene-Versions.

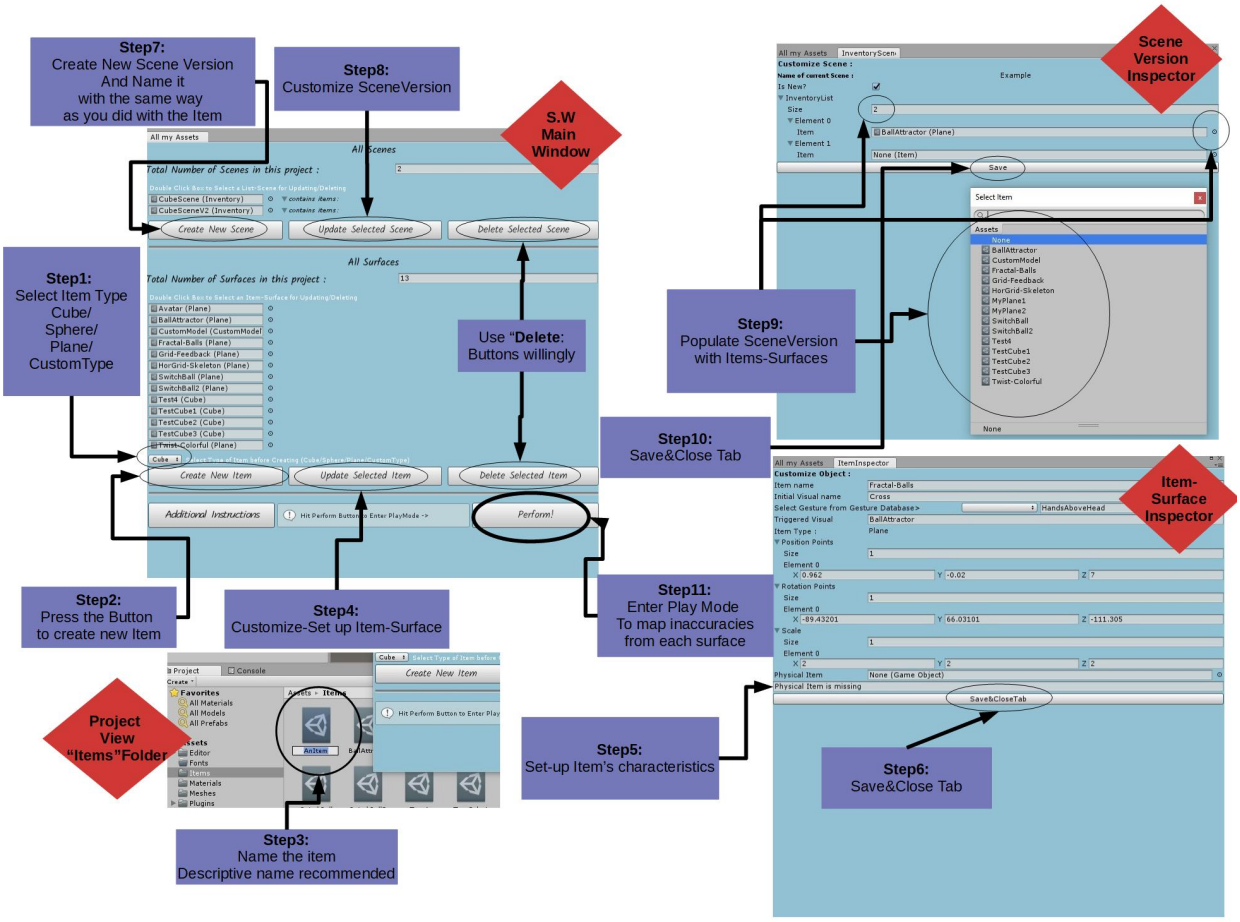


Figure 72: Basic Steps of using SceneWizard Tool

Before creating a Surface the user must select its Type from a Dropdown Button existing right above the "Create New Item" Button, as we can see in the Figure:73 below. The provided types cover the primitive 3D shapes: Cube, Sphere and Plane, but SceneWizard can also grant the creation of a CustomType Surface, which can be linked to a custom 3D Model created from any external Software, as long as its "Mesh Component" can be accessed.

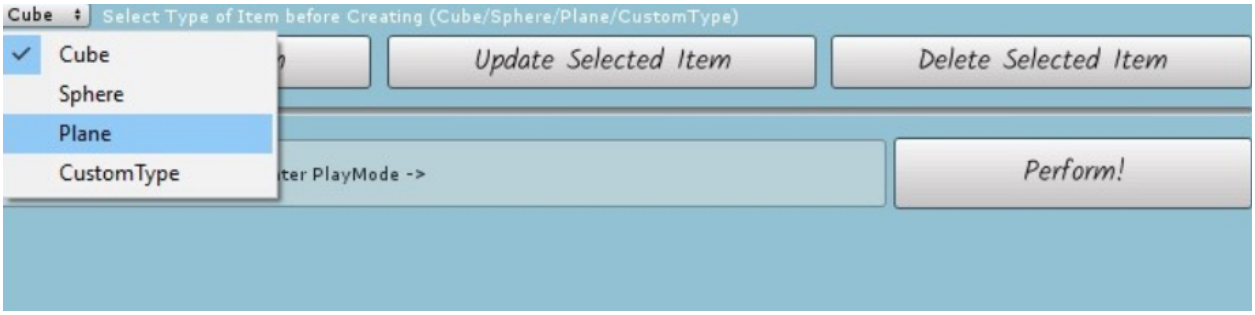


Figure 73: SceneWizard: Item Type Drop-Down Button Selection Example

Upon the creation of the Item-Surface, after the user has clicked the "Create New Item" Button, a new asset file gets created in the "Items" Folder of our Project View. The new Item gets focused and awaits its renaming process, as we can see in the Figure below:

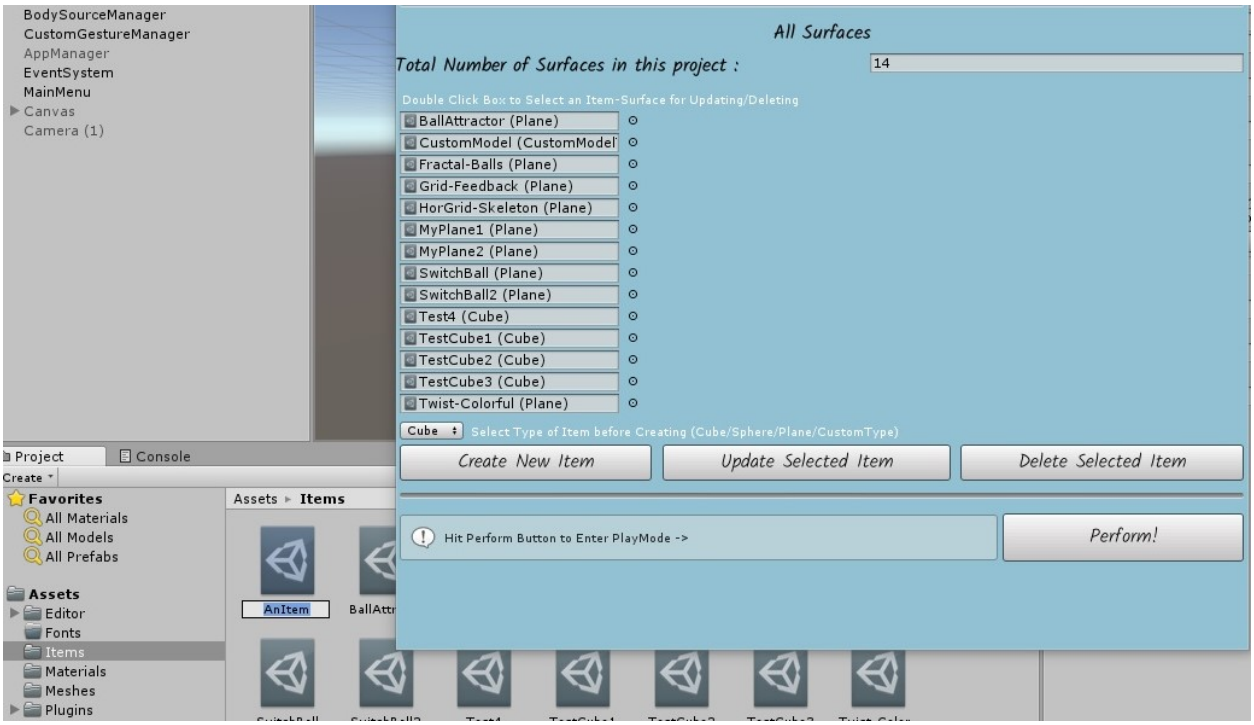


Figure 74: SceneWizard: Create New Item Example

When the user has created all the desired surfaces, the next step is to create one or multiple versions of his/her scene and populate each version with the covetable surfaces. Upon the creation of the Scene-Version, after the user has clicked the "Create New Scene" Button, a new asset file gets created in the "SceneItemsLists" Folder of our Project View and the new Scene file gets focused, awaiting its renaming process, as we can see in the Figure below:

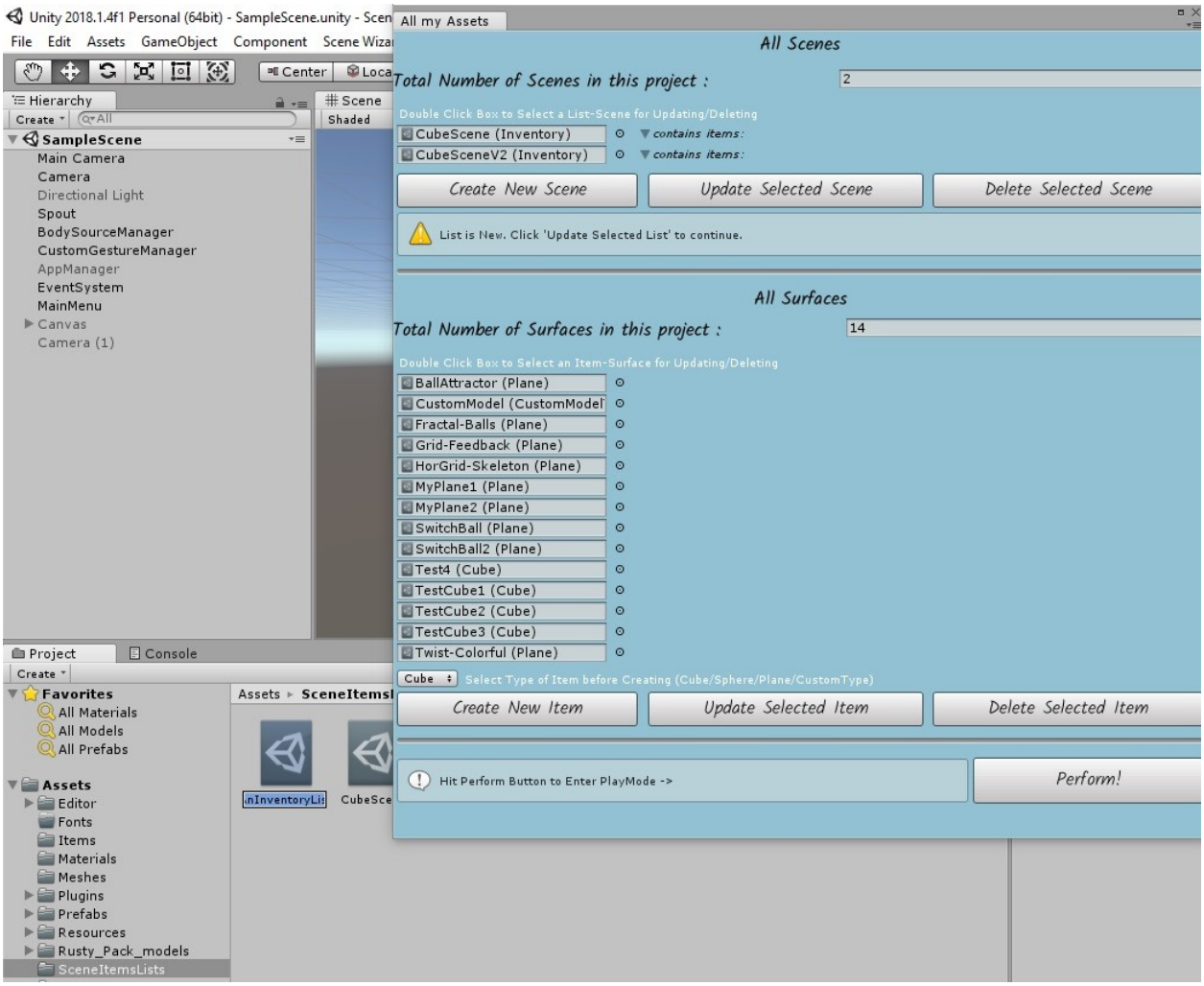


Figure 75: SceneWizard: Create New Scene Example

After the new Scene's Creation, a warning message is being displayed on the Main window, to let the user know the specific scene is brand new and requires customization (See Figure: 76 below.)

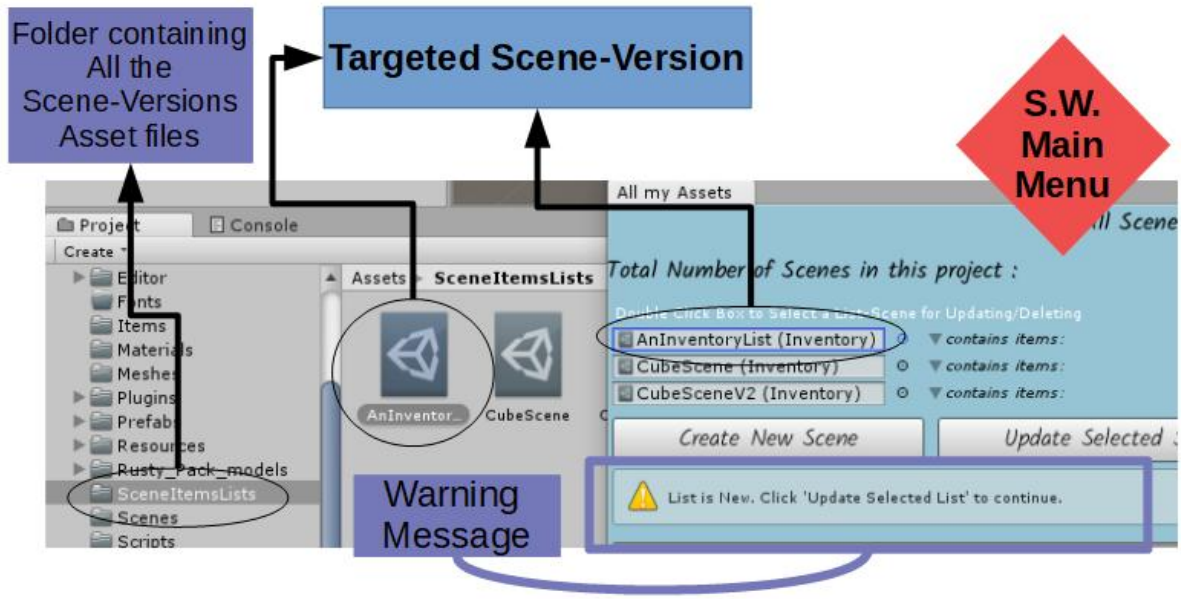


Figure 76: An example of creating a New Scene-Version and the Warning Message that is being displayed.

Moreover, the user needs to customize the Scene-Versions, by populating them with the previously created Items-Surfaces, according to his/her wish and concept. For example, our Demo contains two different versions of one specific scene. The scene is described by a Cube, but we separate the Cube into two different Planes, in order to render different visuals on each side of the Cube. As a result, each Scene-Version contains two different Items-Surfaces, representing the two Plane-sides of the Cube, as we can see in the example Figure:77 below, where the second Scene-Version contains two planes, which names indicate the visuals being displayed on that Surface before and after the gesture.

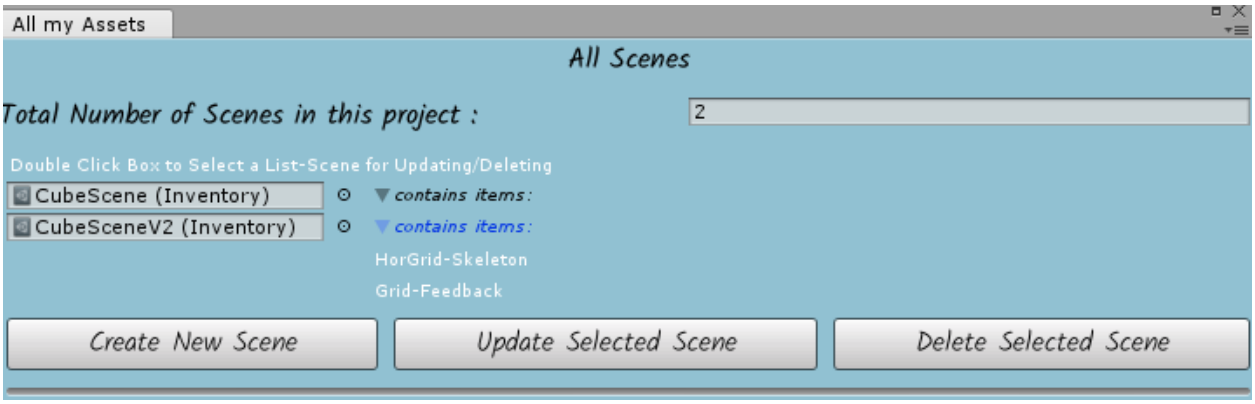


Figure 77: An example of a Scene-Version's containing elements.

Furthermore, if the Scene-Version has been created and is not new anymore, but there are no containing elements in the Scene, warning messages are being displayed to the user via the main window, in order to avoid performance conflicts, errors or blank displaying Surfaces, as we can see in the Figure:78 below.

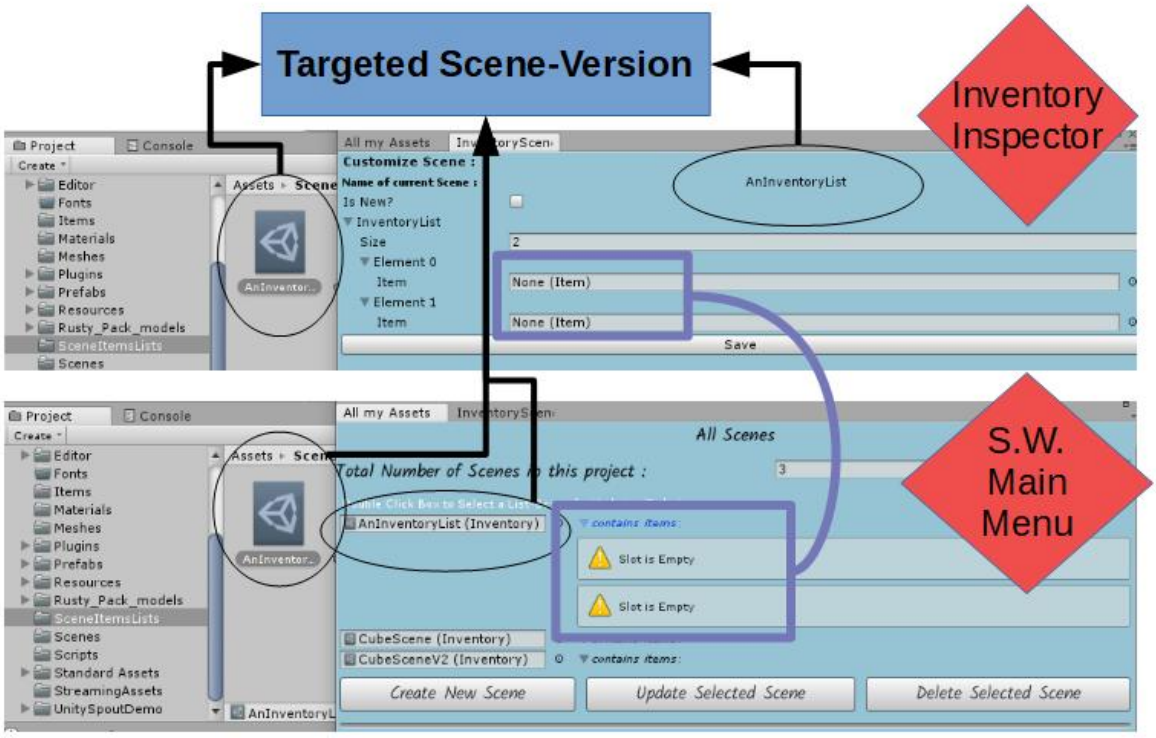


Figure 78: An example of an empty Scene-Version and the Warning Message that is being displayed.

4.3 Peripheral Windows

In order, to control the customization of each and every asset existing in our project, the user will come across additive editor windows, such as the Inspector Editor for the Scene-Version, the corresponding Inspector for Items-Surfaces, the Deletion Validation Windows for deleting Scenes or Surfaces and finally the Instructions guide Editor.

SceneWizard Surface Inspector

After having created the desired number of Items-Surfaces and Scene-Versions, the user needs to customize them as he desires according to his Interactive Projection Mapping concept. The Surface Inspector (Item Inspector) pops up, as a tab, next to our main window's tab, when the user clicks on the "Update Selected Item" Button, in order to allow the user to customize the selected Item-Surface and fill up its parameters. This way the surface can be "decorated" by the desired visuals and corresponding gestures. Moreover,

this Inspector allows the user to define the pose characteristics of each Item-Surface, which must match the intrinsics and extrinsics of real scene's procams. On the contrary, a user can experiment with these parameters, in order to succeed the maximum resolution and image quality for his/her projection. To set-up the position, rotation and size of the surface on the {x,y,z} axis of the 3D virtual space the user can fill up the corresponding Vectors with numbers (1 Unity unit represents 1m in the real world space). Given the fact that the real environment's projector is a corresponding virtual camera, with respect to the intrinsics of the projector, in pose {0,0,0}, the user can create a virtual replica of the real scene, by setting up the Surfaces pose with respect to the geometric extrinsics of the components. The Figure:79 below demonstrates the Steps to update an Item-Surface.

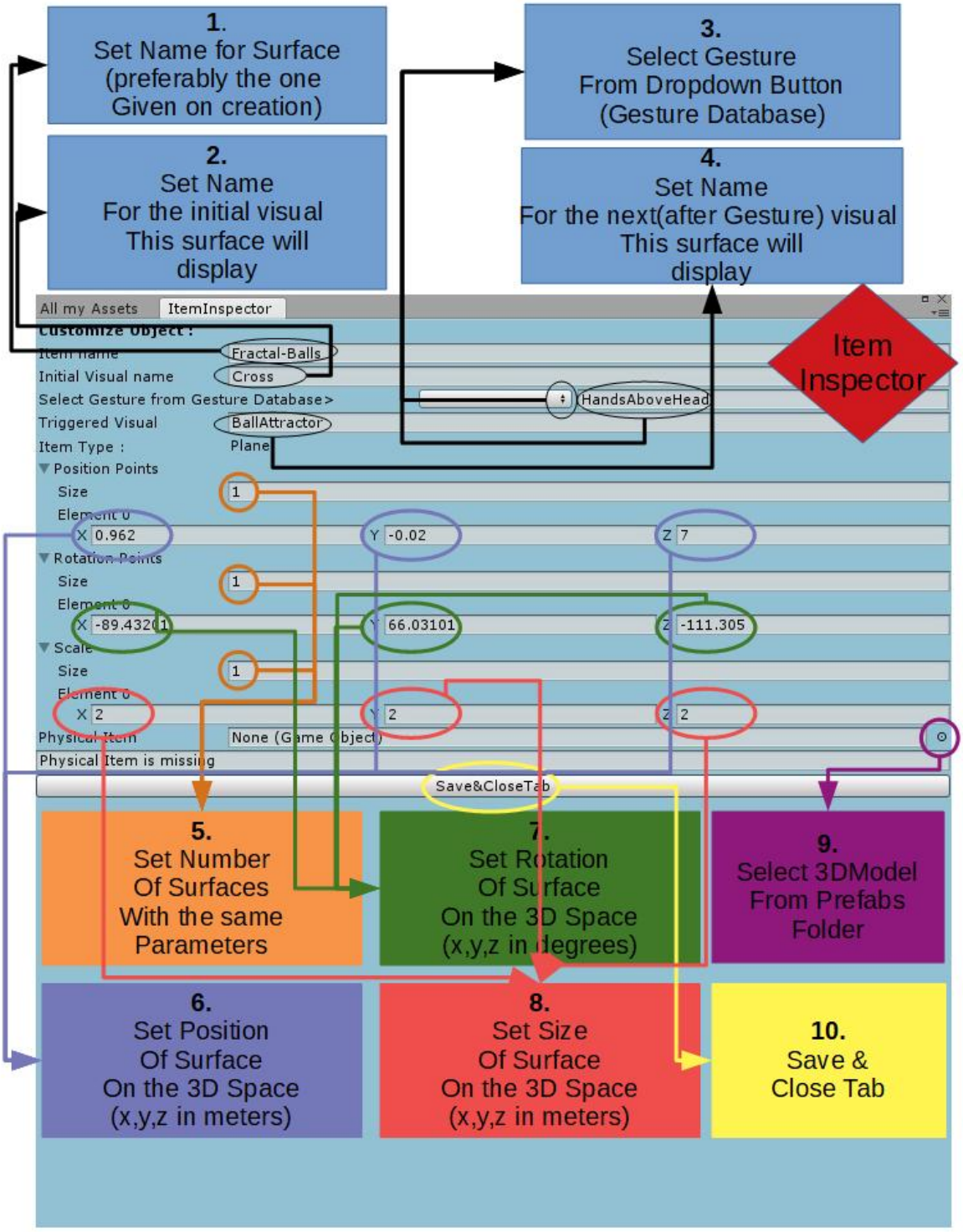


Figure 79: SceneWizard: Update Selected Item Example

Additionally, more than one Surfaces with the same characteristics can be created willingly. The number of same Surfaces can be defined by adjusting the size of the position, rotation, and scale Vectors List, then each element can be posed desirably. However, this lists' size much match numbers for specific characteristics. For example, one cannot have 3 Surfaces with specific gesture and same visuals in 3 different positions, but only set up two rotation or scaling elements. The Vector3 Lists correspond to each other and a warning message is being displayed, in case the user mis-sets the size numbers, as we can see in the Figures: 80 and 81 below.

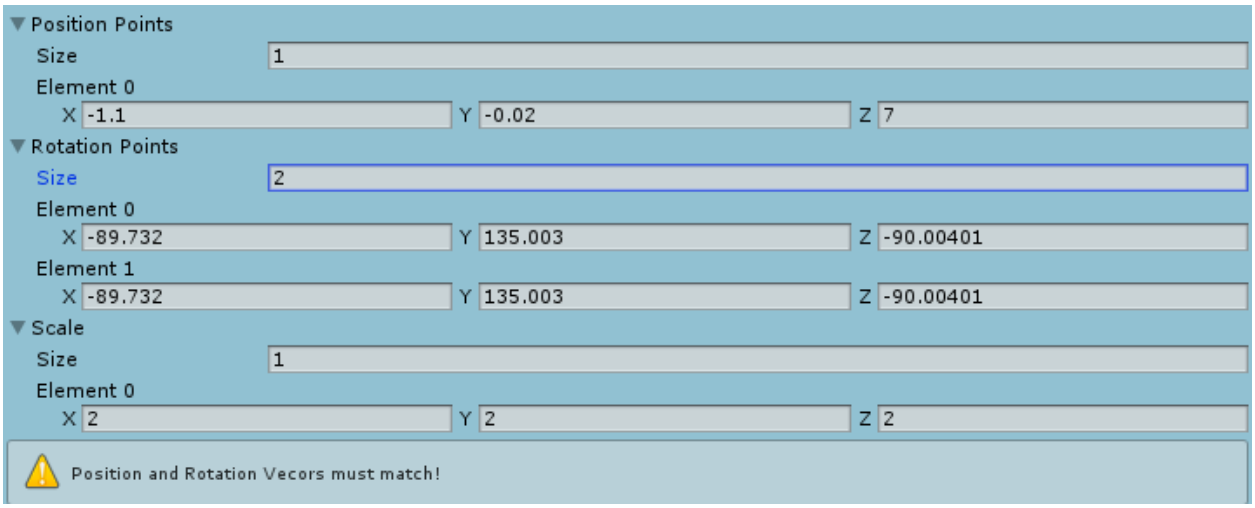


Figure 80: Non-Matching Sizes Position-Rotation Vectors Example

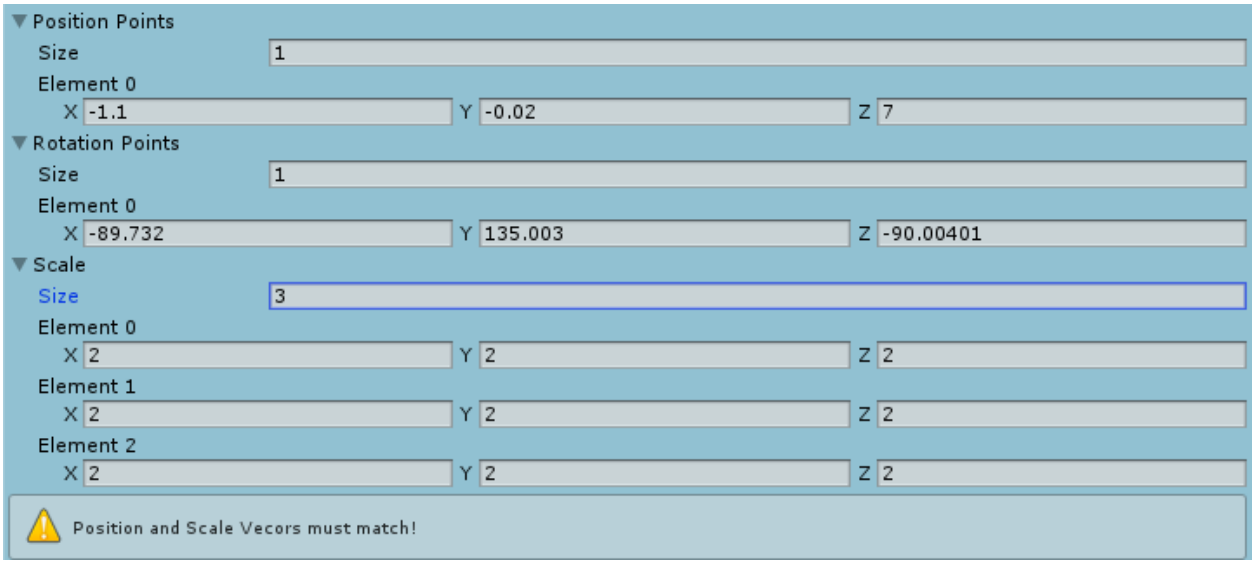


Figure 81: Non-Matching Sizes Position-Scale Vectors Example

The Figure:82 below demonstrates an example of how we set an item's parameters in the Editor, next to

the corresponding visuals.

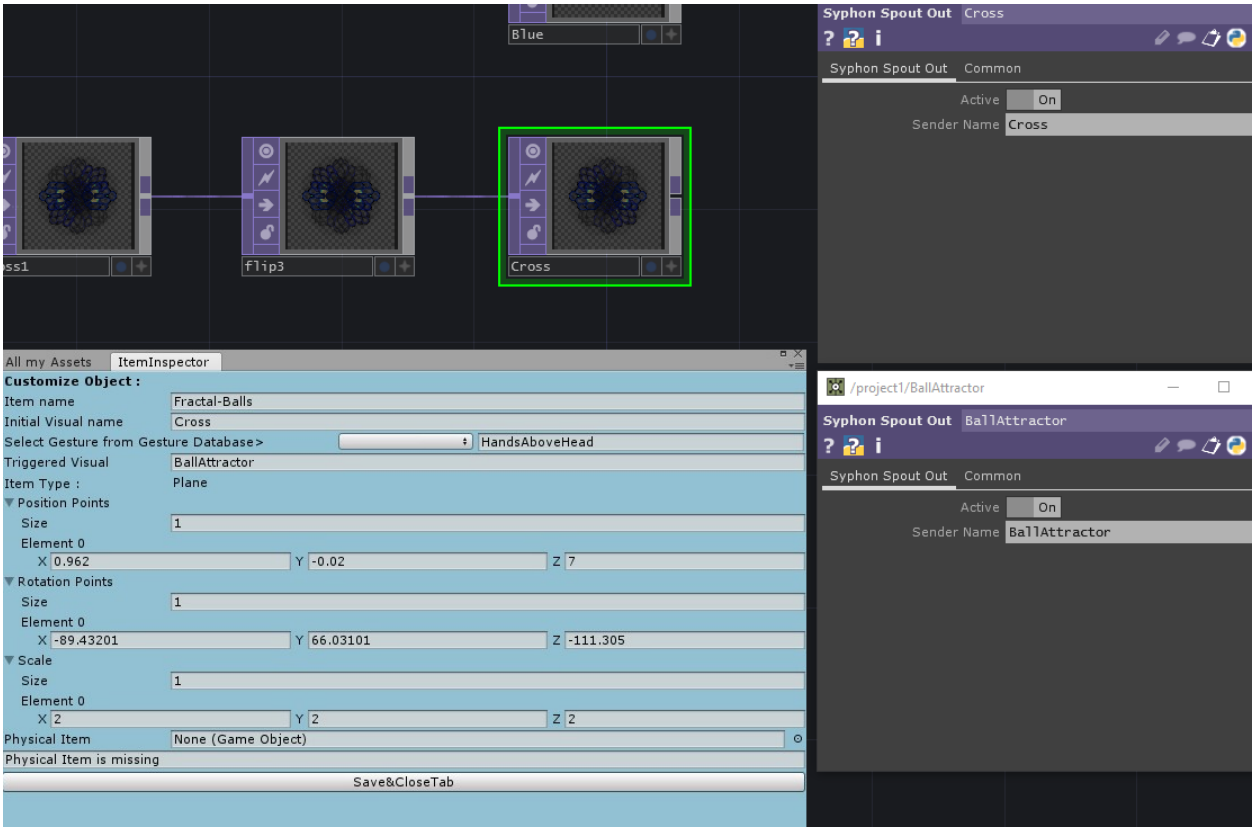


Figure 82: An example of Items-Surfaces Corresponding Parameters

SceneWizard Item Deletion Inspector

Finally, if the user selects an Item-Surface and wishes to delete it from the Project's Folders, he can click on the "Delete Selected Item" Button. A validation window will pop up, re-assure the deletion of the selected Item, as we can see in the Figure below:

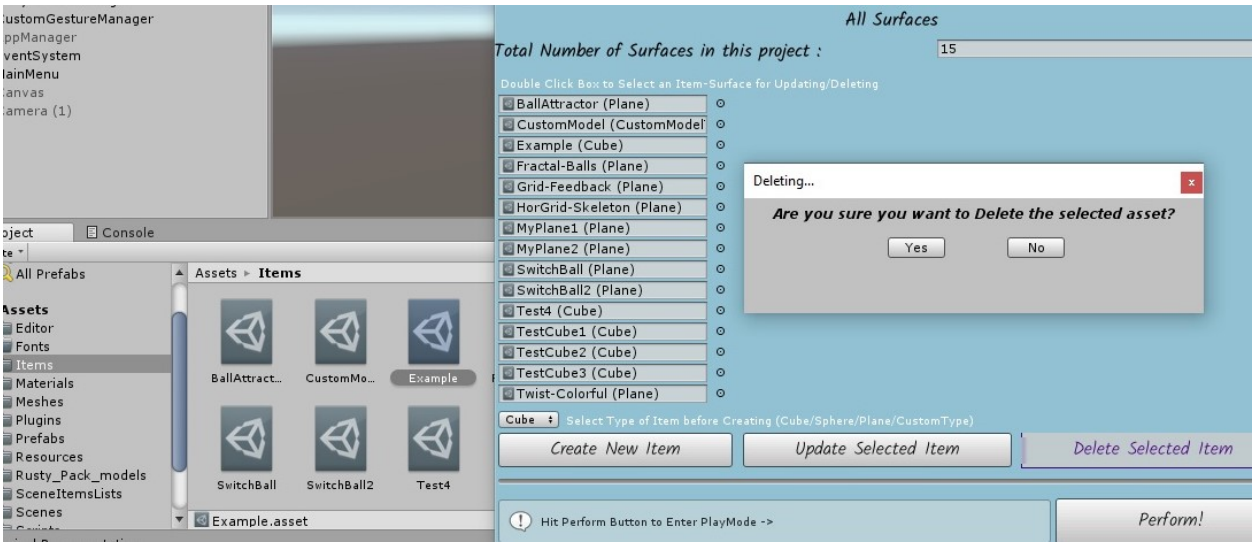


Figure 83: SceneWizard: Delete Selected Item Example

SceneWizard Scene Inspector To continue with the Assets' Inspection. The following step for the user is to colonize the Scene Versions with the corresponding number of Items-Surfaces, as we can see in the figure below:

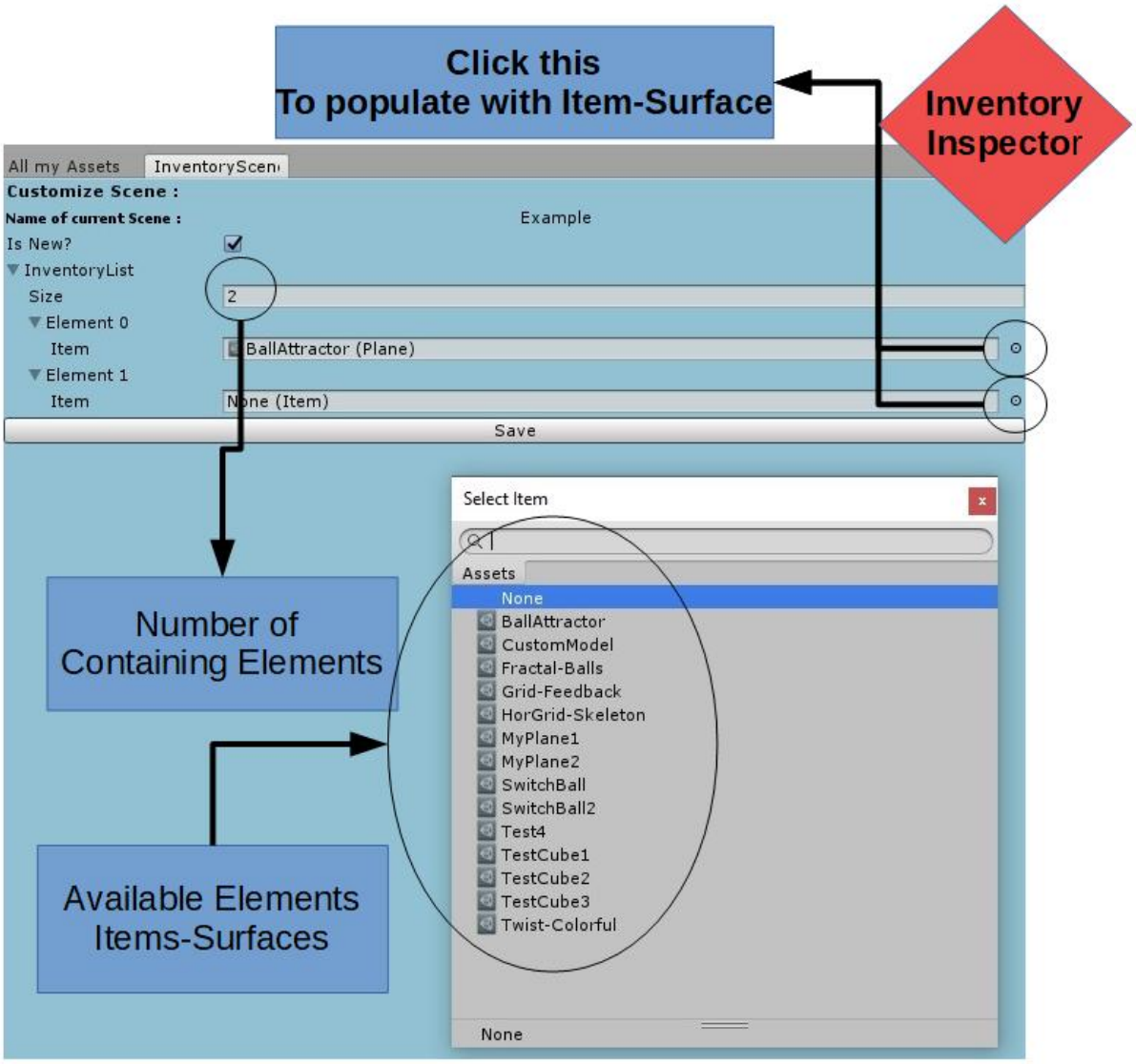


Figure 84: SceneWizard: Populate Selected Scene Example

Finally, a similar Button for the Deletion of a Selected Scene Version, exists in the main window and upon a click, pops a validation window, similarly to the deletion of an Item-Surface.

SceneWizard Instructions Inspector

When all the Scene-Versions have been desirably populated, the user can get informed about the following actions, by pressing the "Instructions" Button. When this Button is pressed another Editor Window pops, to guide the user about the Keyboard's functions he/she has the possibility to make use of, as we can see in the Figure:85 below:

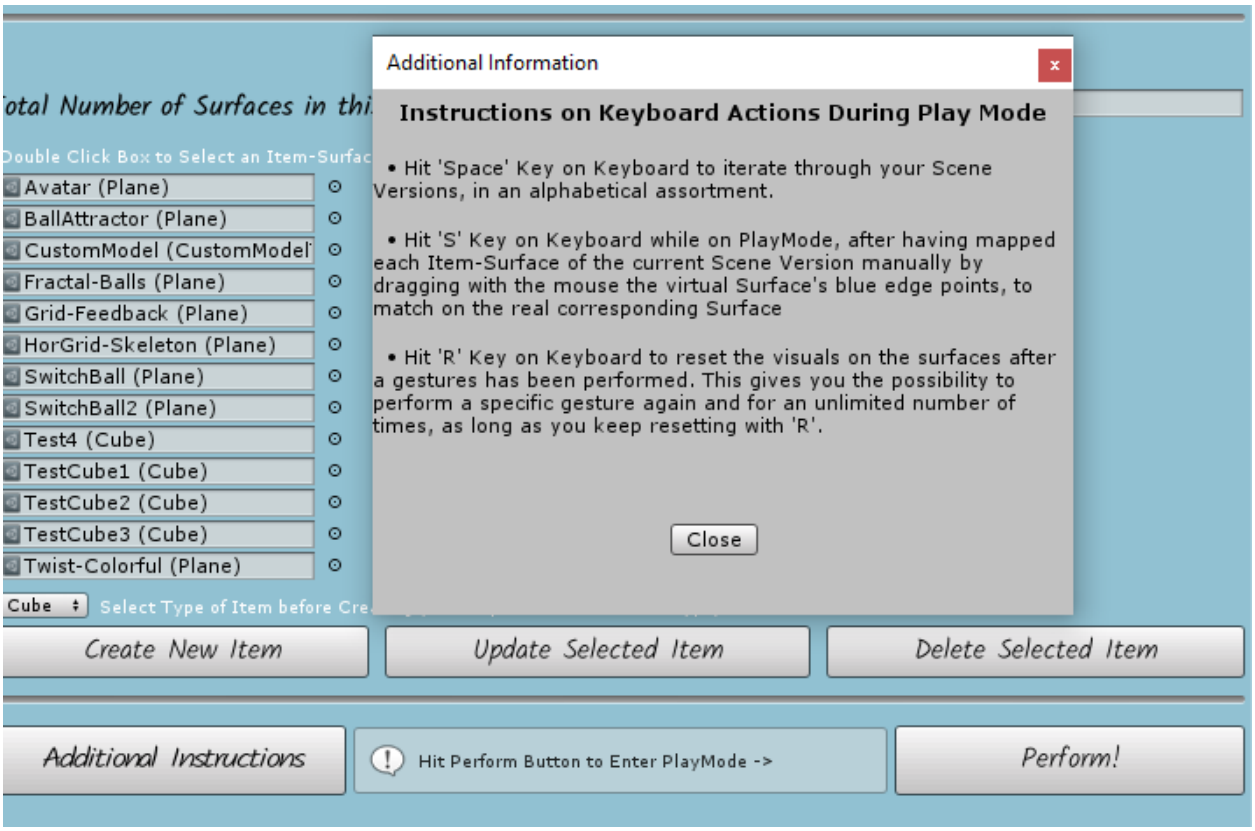


Figure 85: Instructions for SceneWizard during PlayMode.

SceneWizard during Play Mode

We have already discussed, how unrealistic and doubtfull it is to achieve a 100% match between the virtual and the real Surfaces' alignment. However, SceneWizard provides simple tools to eliminate the image-to-geometry registration issue of Projection Mapping. In the Figure below, we can see an example Scene with two different Planes, where one of them (right Plane), is selected and ready to be edited:

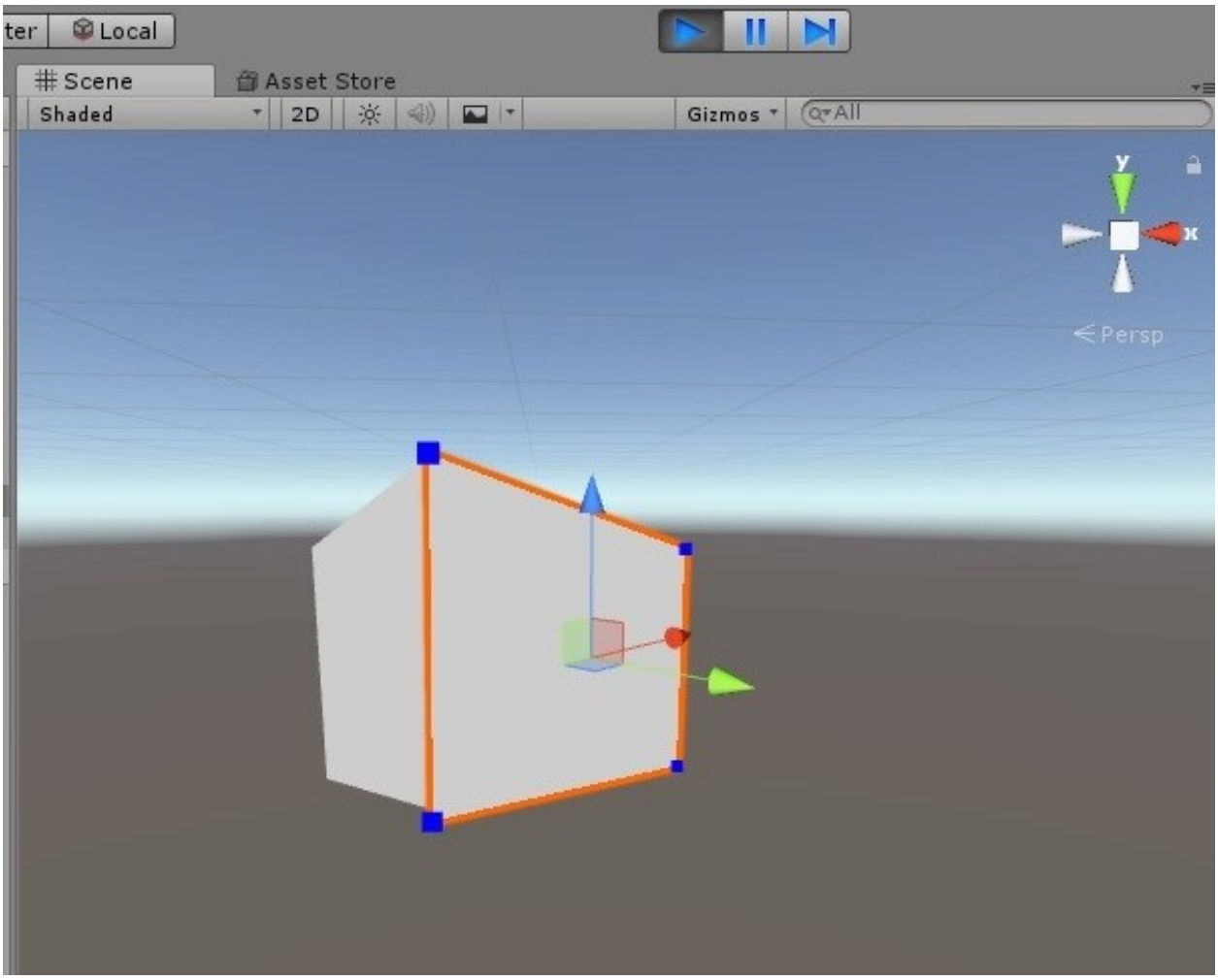


Figure 86: Selected Surface in Play Mode, Blue points: Surface's edges.

The user can drag and drop these blue dots, deforming the object's Mesh to match the real-world's surface. The Figure:87 below shows the same Plane as the Figure:86 above, but after expanding the right side's edges towards right:

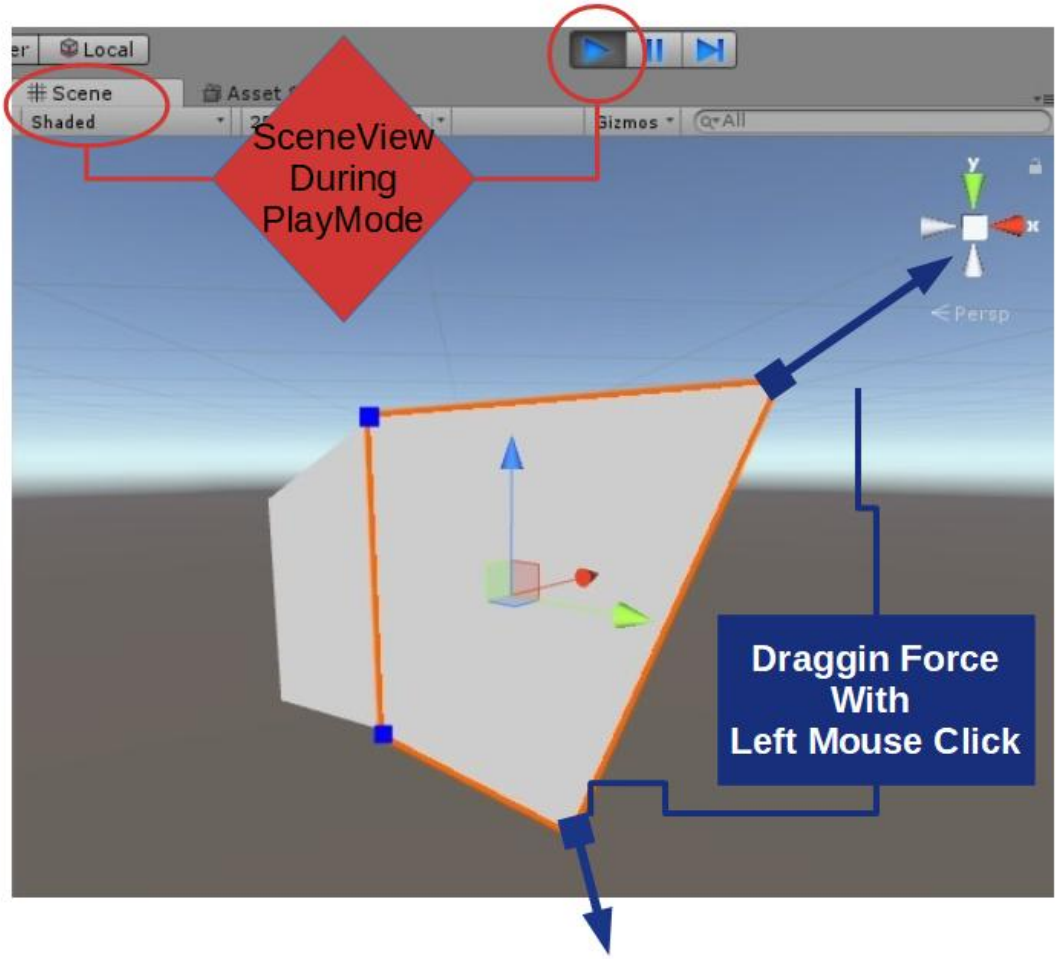


Figure 87: Selected Deformed Surface by dragging blue edges in Play Mode.

When all this is done, the user is ready to perform an Interactive Projection Mapping Performance, according to his set-up and story concept, just by standing in front of a Kinect camera, or more! The results can be interesting, exciting and quite mesmerizing to the viewers.

5 Implementation

5.1 SceneWizard Tool

5.1.1 Introduction

In this section we will discuss the main functionality and architecture of our tool's system and how/why everything was implemented. Besides Kinect's API and Spout Protocol, which we will discuss in detail further in this section, SceneWizard also provides us with tools of its own. All these tools were developed from scratch in Unity and can be not only used, but also customized willingly. Below we will analyze them one by one.

5.1.2 System's Architecture

Our main goal was to introduce gesture recognition into Projection Mapping performances, in order to manipulate visuals by moving our body, producing likewise a mesmerizing result. Working in Unity we needed a concept that will efficiently implement this idea into Unity's system and work-flow. Unity consists of three fundamental building blocks:

•GameObjects → Every kind of content in Unity begins with a GameObject, any Object in Unity's Scene is a GameObject. However, a GameObject needs properties, in order to represent something tangible, the Components.

•Components → They define and control the GameObjects' behavior.

•Variables → Components have any number of editable properties that can be tweaked via the Inspector window in the editor, and/or via script.

When working in Unity, any item used in our project can be represented by an asset. An asset may come from a file created outside of Unity, such as a 3D model, an audio file, an image, or any of the other types of file that Unity supports. There are also some asset types that can be created within Unity. In the classic way, while creating a virtual world with Unity, the user commits to one or multiple scenes, which contain the environment. The environment is usually compiled by GameObjects, which in their turn are driven by components and variables and so on. It would be reasonable, as to represent the Surfaces of a real-environment scene with GameObject primitives, or custom 3D models. Their behavior, would then be defined and controlled by MonoBehaviour Class Scripts, which is the base Class from which every Unity script derives from. However, if the user-artist would be obligated to use Unity's workflow, he or she would be subjected into enriching their programming skills, in order to create an interactive performance using GameObjects and Monobehaviors.

Our tool is based on the use of ScriptableObjects, instead of MonoBehaviours. ScriptableObject is a serializable Unity class that allows us to store large quantities of shared data independent from script instances. Using ScriptableObjects makes it easier to manage changes and debugging. In addition to this, one of the main use cases for ScriptableObjects is to reduce our Project's memory usage by avoiding copies of values. Furthermore, using SciptableObjects, we can store and manage data during Editor Mode, as well as save data as Assets to use at Runtime. The main concept, is using Unity's ScriptableObjects, to create the possibility of a data-driven Projection Mapping performance, within Unity's Play Mode.

When using these data containers, we basically get several advantages in controlling the set-up and performance of our project. To begin with, our scene must consist of surface Items, which represent each surface we need with a different visual effect, while some Surfaces may share a state, as if for example two Surfaces may display the same visual. ScriptableObjects encourage a strong separation of shared versus non shared states of Objects, thus working as a pattern. Secondly, working with ScriptableObjects, allows us to create multiple versions of one scene, which we can create in Edit Mode and perform in Play Mode. This way, any changes that will be produced during Play Mode, will not affect our scene versions, therefore our assets will remain intact and able to perform again. Additionally, ScriptableObjects give us total control over file granularity. In Unity, MonoBehaviours have sub-file granularity, which indicates large amount of potential data information that form dependencies and does not allow sharing files between different projects due to merge conflicts. ScriptableObjects provide access to all the engine's structures and behaviors while maintaining a lightweight base. Unlike MonoBehaviours classes, ScriptableObjects don't need to be attached to GameObjects. They're smoothly tied into Unity; we can instance a ScriptableObject as an asset from an easy click(see Figure:88 below) in our project directory or from the UI we created for our tool, and that object can also be inspected with that same UI – they're fully understood objects within the Unity ecosystem. Code-based instantiation does of course still exist, as a means of the performance. In the section below we will discuss in detail, how we implemented ScriptableObjects in our project. Note that, our project, consists of two types of ScriptableObjects:

- Asset Surfaces,
- Asset Scenes.

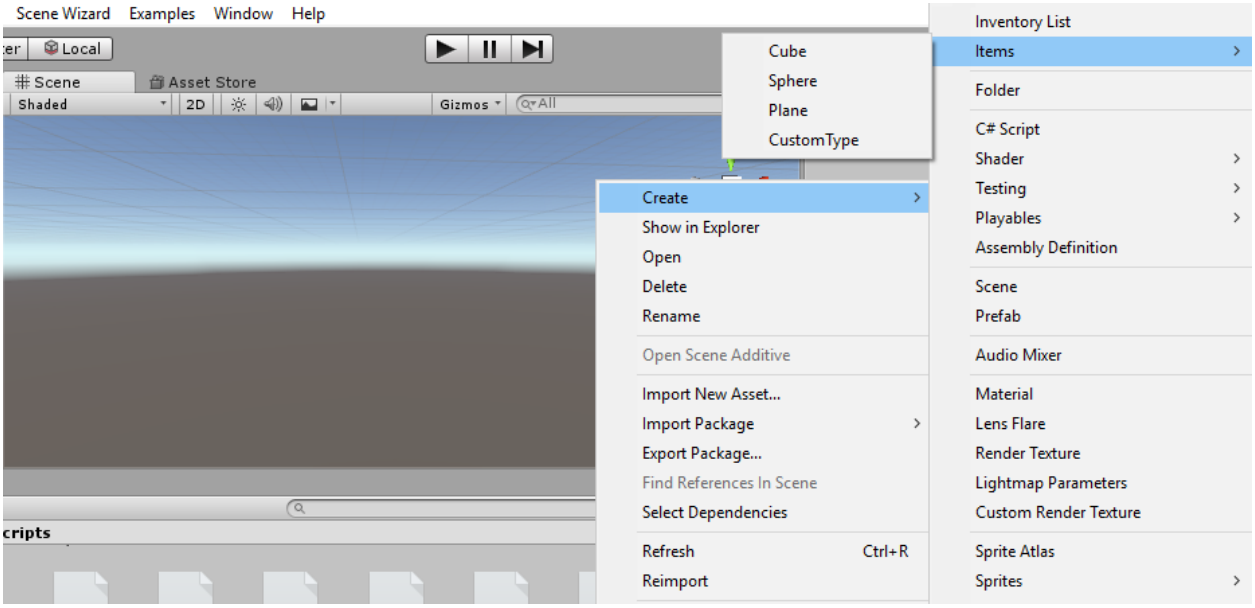


Figure 88: Unity Project Settings: Right Click Option

5.1.3 Assets Surfaces

The Surfaces assets, represent any surface from the real-environment scene. These are ScriptableObjects which contain several parameters data. These data characterize each surface. A surface is defined by:

i its name

ii its initial visual name

iii its gesture name (provided from gesture database)

iv its triggered visual name

v its type

vi its position vector points

vii its rotation vector points

viii its scale vector points

ix its physical representation

To generate this type of data containers, we created a script in our project called "Item", which is deriving from ScriptableObject Class of Unity, instead of MonoBehaviour. This action informs Unity that we still want to use Unity features and methods, like a typical MonoBehaviour, but we will no longer need to attach this Script onto a GameObject. Instead, it will be treated like any other common asset that can be created, similar to creating a prefab, scene or material. Then, we fill this Script's declaring variables with some serialized fields that will contain all the data, corresponding to the information displayed on the Item-Surface. These data define the Item-Surface, as we mentioned above, and for these we declare:

i A string to hold the name of the Item-Surface: "itemName" variable.

ii A string to hold its initial visual name: "visual" variable.

iii A string to hold its corresponding gesture name (provided from gesture database): "gesture" variable.

iv A string to hold its triggered visual name: "newVisual" variable

v An enumeration[1] to declare the various types of surfaces "ItemType" enumeration variable.

vi An enumeration type to hold its type: "itemType" variable.

vii A 3-array Vector (Vector3) to hold its position on the 3D space (x, y, z): "spawnPoints" variable.

viii A 3-array Vector (Vector3) to hold its rotation on the 3D space (x, y, z): "spawnRotationPoints" variable.

ix A 3-array Vector (Vector3) to hold its scale on the 3D space (x, y, z): "spawnSize" variable.

x A GameObject to reference its physical representation (if needed for imported 3D models).

The code snippet below, shows the ScriptableClass of our Items-surfaces and the declarations of the variable we mentioned above, where each variable is hidden from the Inspector of Unity, so the users do not get confused in performing extra actions,

Listing 1: Items/Surfaces Types

```
[System.Serializable]
[ExecuteInEditMode]
public abstract class Item : ScriptableObject
{
    public enum ItemType
    {
        Cube, Sphere, Plane, CustomModelType
    }
    [SerializeField]
    [HideInInspector]
    public ItemType itemType;               -------> v.
    [SerializeField]
    [HideInInspector]
    public string itemName;                  -------> i.
    [SerializeField]
    [HideInInspector]
    public GameObject physicalRepresentation; -----> ix.
    [SerializeField]
    [HideInInspector]
    public string visual;                   -------> ii.
    [SerializeField]
    [HideInInspector]
    public string gesture;                  ------> iii.
    [SerializeField]
    [HideInInspector]
    public string newVisual;                -------> iv.
    [SerializeField]
    public Vector3[] spawnPoints;           -------> vi.
    [SerializeField]
    public Vector3[] spawnRotationPoints;   ------> vii.
    [SerializeField]
    public Vector3[] spawnSize;             -----> viii.
```

[1] an enum is a custom type of variable that we can create in our scripts

and the Figure:89 below demonstrates an example of how these parameters show in our custom Editor Window.

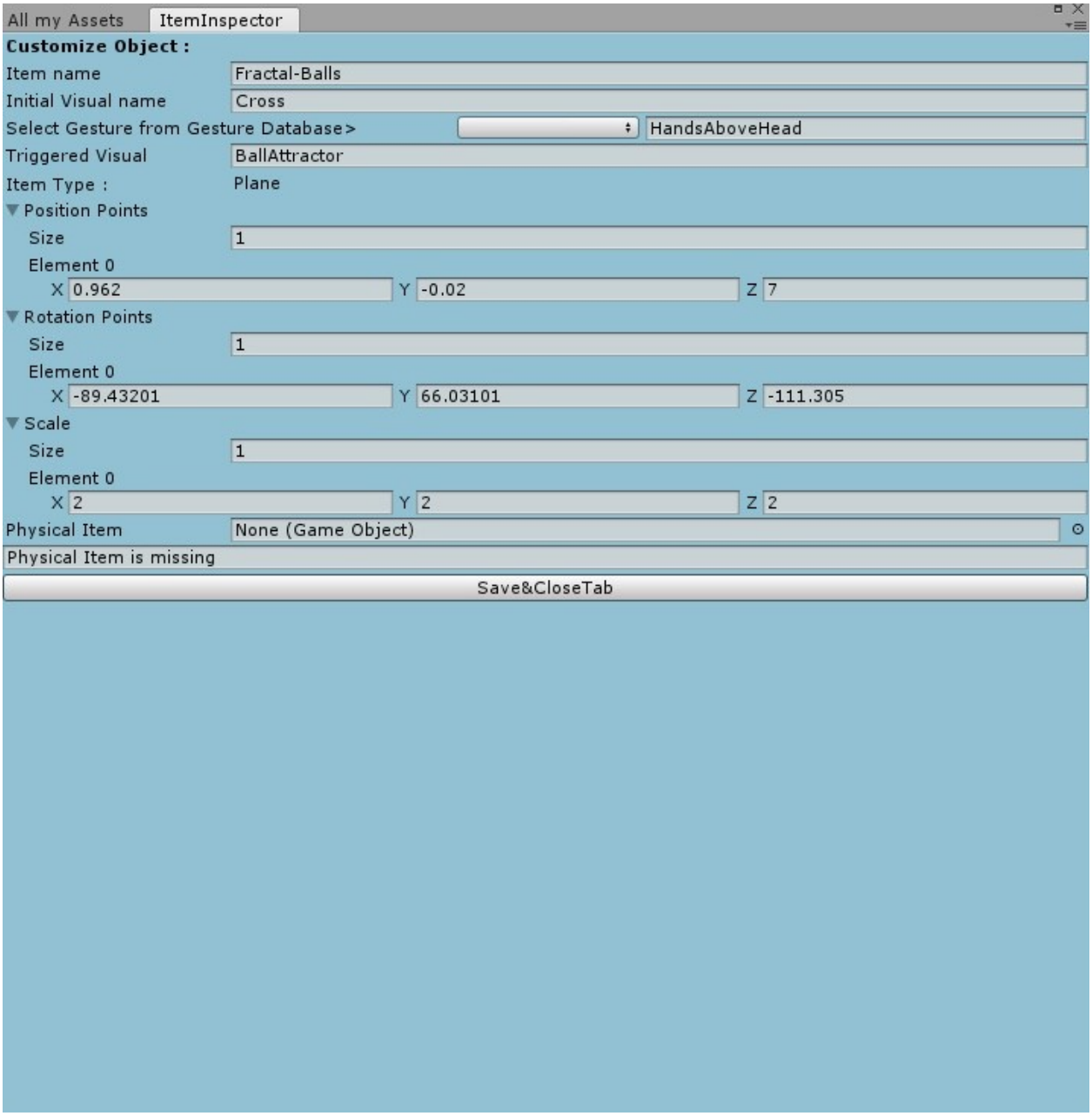


Figure 89: An example Items-Surfaces Data Parameters

For every surface, the user has the possibility to create different types of Items, such as Unity's primitive types (Cube, Sphere, Plane), as well as reference a custom type of item, in case the user desires to import a virtual 3D model of the scene (CustomType). Thus, each item will need it's own unique implementation

of the Item ScriptableObject. For this reason we declared the types in an enumeration variable and created Scripts for each type, which derive from the "Item" Class, such as the snippet code below, which creates a "Cube" type of an Item-Surface. In this script we implemented the aforementioned "right-click-create" option, to add our ScriptableObject to the Asset Menu, for the user's convenience, by giving our Cube's Class the attribute of being added in the Asset Menu, with a specific name, a initial name on creation and the order it will appear in the Menu.

Listing 2: Sub-Class of Item: Cube Item Type

```
[System.Serializable]
[ExecuteInEditMode]
[CreateAssetMenu(menuName = "Items/Cube", fileName = "CubeName.asset", order = 1)]
public class Cube : Item {
...
}
```

The parameters that characterize the items, may differ from item to item. In order to handle the items' mutability, we have created another assisting class that holds the unique fields of a specific item instance and allows us to have copies with mutable data. This class creates a template for each created item, storing all its specific data in an instance-specific field and returns the data when an item gets accessed. In the Script we declared all the significant variables of the an Item, such as:

i The Item itself: "item" variable.

ii Its Name: "theName" variable.

iii Its Initial Visual's Name: "myVisual" variable.

iv Its corresponding Gesture's Name: "myGesture" variable.

v Its New Visual's Name: "newVis" variable.

The "ItemInstance()" public function returns the significant data when a specific item gets accessed. The code snippet below demonstrates this class:

Listing 3: Sub-Class of Item: ItemInstance

```
[System.Serializable]
[ExecuteInEditMode]
public class ItemInstance
{
    public Item item;
    [HideInInspector]
    [SerializeField]
    public string theName;
    //public int quantity = 1;
    [HideInInspector]
    public Transform itemSize;
    [HideInInspector]
    public bool isNew;
    [HideInInspector]
    public string myVisual;
    [HideInInspector]
    public string myGesture;
    [HideInInspector]
    public string newVis;
    private string _item ;

```

```
22      public ItemInstance(string itemName, int quantity, Transform itemSize, bool isNew,
            string myVisual, string myGesture, string newVis)
23      {
24          //this.item = item;
25
26          this.theName = itemName;
27          Debug.Log(itemName);
28          // this.quantity = quantity;
29          this.itemSize = itemSize;
30          this.isNew = isNew;
31          this.myVisual = myVisual;
32          this.myGesture = myGesture;
33          this.newVis = newVis;
34          this.itemSize.tag = item.name;
35      }
36  }
```

5.1.4 Asset Scenes

The Scene assets represent, each version of a specific virtual scene, corresponding to a real scene. Basically, Scene assets are the last piece of back-end, that holds the references to all our surfaces-items. Each Scene asset, can contain a number of surface-items as a list, which the user can select and modify, via SceneWizard's Editor Window. First of all, we have a getter(which is a public function, that is called to get the value of a cpecific private variable), whose goal is simply to maintain reference to the active inventory (Scene – List of surface-items). This is achieved by searching for all loaded objects of type Inventory, by using the "Resources()" Class, which allows us to find and access Objects including assets, as well as "FindObjectsOfTypeAll<>()" Method, which returns a list of all objects of our significant type(Inventory-SceneVersions Assets). – i.e. if there is an instance of Inventory available in the project's folder, it'll be assigned to "_instance" and returned, for the most part this is just a singleton[2]. The code snippet below, demonstrates the creation of these lists.

Listing 4: Parent-Class of Item: Inventory

```
1  [CreateAssetMenu(menuName = "Inventory List", fileName = "Inventory.asset", order = 0)]
2  [System.Serializable]
3  [ExecuteInEditMode]
4  public class Inventory : ScriptableObject
5  {
6
7      [HideInInspector]
8      [SerializeField]
9      public string InvName;
10     public bool isNew = false;
11     [SerializeField]
12     private static Inventory _instance;
13     public static Inventory Instance
14     {
15         get
16         {
17             if (!_instance)
18             {
19                 Inventory[] tmp = Resources.FindObjectsOfTypeAll<Inventory>();
20                 if (tmp.Length > 0)
```

[2]Singleton is a basic Design Pattern and Classes implementing Singleton pattern will ensure that only one instance of an object ever exists at any one time. It is recommended to use Singletons for objects that do not need to be copied multiple times during a Game/App/Tool.This is great for controller classes that Manage the Game/App/Tool

```
                {
                    _instance = tmp[0];
                    Debug.Log("Found inventory as: " + _instance);

                }
                else
                {
                    Debug.Log("Did not find inventory, loading from file or template.");
                    SaveManager.LoadOrInitializeInventory();
                }
            }
            return _instance;
        }
        set
        {
            // _instance.name = _instance.InvName;
        }

    }
```

It also makes use of "SaveManager" Script, created to automatically load the inventory as necessary. This scripts contains a "LoadOrInitializeInventory()" Method, which loads the inventory(Scene-Version Asset) if it exists, or creates a new one if it doesn't. It also contains a "SaveInventory()" Method, which saves our Inventory into a JSON[3] Data Format, by allocating GC memory[4] only for the returned inventory.

In addition to this, after any operation that modifies the inventory, we call Save(), which will be discussed further in Window Layout Section. Furthermore, we implemented the "OnDestroy()" Method, which occurs when a Scene or game ends. Stopping the Play mode, when running from inside the Editor will end the application, too. As this end happens an "OnDestroy()" will be executed, in order to free-up the memory space, used to "hold" our Singleton instance. The code snippet below, demonstrates these function:

Listing 5: Save Option in InventoryScript

```
// Simply save.
public void Save()
{
    SaveManager.SaveInventory();
}
void OnDestroy()
{
    //Debug.Log("CustomGestureManager.OnDestroy");
    if (_instance != this) return;
    _instance = null;
}
```

5.1.5 Editor Scripting – Window Layout

As we have mentioned before, we developed a User Interface, so that the user has the possibility to manage and modify our workable backend Inventory System of Surface-Items and Scene Versions. To recap our backend involves:

[3](JavaScript Object Notation) is a lightweight data-interchange format.

[4]Garbage Collection is the name of the process that makes the memory, that has been set aside to store data but is no longer in use, available again for reuse. Garbage Collection is a common cause of performance problems, if it happens too often and by using memory efficiently can minimize the impact of Garbage Collection on our "game", thus optimize performance

•Item Classes (as ScriptableObjects) → Each Type of Item is represented as an asset in our project's folder: "Items".

•Inventory (as ScriptableObject) → The base of each Inventory created, represented as an asset in our project's folder: "SceneItemsLists".

•Save Manager → Simple helper to manage saving in a convenient way.

To manage these assets, we created a custom Editor Window, based on Unity's Editor Class, which is base class to derive custom Editors from, used to create our own custom inspectors, by changing their appearance and editors for handling and editing multiple assets.

Initially, the code that creates the window and populates the assets, filtering the user's selected asset. This is done, by using the "SelectionMode()" enumeration of Unity, which teaks the selected Object and filters its Type. If the selected Object-Asset is one of the significant types (Inventory or Item), we tie the Object to a parameter, so that we can pass the specification of the targeted Object to the other Editor Windows (DeleteCheckEditor, SceneInspector or ItemInspector) and perform actions on it (editing modifications, deleting). We declare the targeted Object as a "SerializedObject", which is a class suitable for editing serialized fields on Unity Objects in a completely generic way. This class automatically handles dirtying[5] individual serialized fields, so they will be processed by the Undo system and styled correctly for Prefab overrides when drawn in the Inspector. When double clicking on an asset file, this specific file is targeted and ready to be modified.

To create our custom Editor Window, we declare a public function, in our custom window type and make use of the "GetWindow()"Class of the Editor, which returns the first Editor Window of type "InventoryInspector"(our custom window type). If there is none, it creates and shows a new window and returns the instance of it. The window appears, when the "ShowWindow()" function gets called, which displays the window that has been created and in continuous calls the "Populate()" function, to populate the assets, as we can see in the code snippet below:

Listing 6: SceneWizard EditorWindow

```
1  void Populate()
2  {
3      UnityEngine.Object[] Invselection = Selection.GetFiltered(typeof(Inventory),
           SelectionMode.Assets);
4      UnityEngine.Object[] Itemselection = Selection.GetFiltered(typeof(Item),
           SelectionMode.Assets);
5      if (Invselection.Length > 0)
6      {
7          if (Invselection[0] == null)
8              return;
9
10
11         inv = (Inventory)Invselection[0];
12         // initialization of the serializedObj, that we are working on
13         targetObject = new SerializedObject(inv);
14         Debug.Log("target=");
15         Debug.Log(inv.name);
16         //Debug.Log(inv.inventory.Length);
17     }
```

[5]An object that is set to dirty keeps any changes made to it after the game(PlayMode in our case) stops playing and carries them over into editing mode

```
18      if (Itemselection.Length > 0)
19      {
20          if (Itemselection[0] == null)
21              return;
22
23          item = (Item)Itemselection[0];
24          // initialization of the serializedObj, that we are working on
25          targetObject = new SerializedObject(item);
26          Debug.Log("target=");
27          Debug.Log(item.name);
28      }
29  }
30  public static InventoryInspector InvInspector
31  {
32      get { return GetWindow<InventoryInspector>(); }
33  }
34
35  [MenuItem("Scene Wizard/Create Scene Items Wizard to customize your scene...", false, 1)
        ]//Declaring to Unity engine that this is meant as a menu item.
36
37  public static void ShowWindow()
38  {
39
40      invetoryInspector = GetWindow<InventoryInspector>(false, "InventoryInspector", true)
            ;
41      //invetoryWindow = GetWindowWithRect(typeof(InventoryInspector), new Rect(0, 0, 820,
            720), true);
42      GUIContent titlecontent = new GUIContent("All my Assets"); //Creating some title
            content.
43
44      invetoryInspector.titleContent = titlecontent;
45      invetoryInspector.name = "MainWindow";
46
47      invetoryInspector.Populate();
48      invetoryInspector.Show();
49  }
```

The main Window of SceneWizard, pops after clicking the corresponding Menu tab, utilizing the ShowWindow() function, as we can see on the Figure:88. Our main Window demonstrates, all the significant asset files that we need to create an Interactive Projection Mapping performance. The window includes:

•The total number of versions of a scene we currently have in our project, as an integer field, that cannot be edited, is only for displaying information for the user.

•A list of all the versions of a scene, as object fields, that can be selected by double clicking on them.

•A list, that corresponds to the scene versions list, as label fields, that demonstrate which Item-Surfaces each scene version contains, by clicking on the arrow.

•Three Buttons that guide the user on Creating, Updating or Deleting a scene version.

•The total number of items-surfaces we currently have in our project, as an integer field, that cannot be edited, is only for displaying information for the user.

•A list of all items-surfaces, as object fields, that can be selected by double clicking on them.

- A Dropdown Button for selecting a Type of an Item-Surface, before creating it.
- Three Buttons that guide the user on Creating, Updating or Deleting an Item-Surface.
- A Button, at the end of the window's layout, that the user clicks to enter Play Mode.
- Additional Information that guide the user, on how to use the window.

The Figure:90 below, shows an example of the main window of SceneWizard.

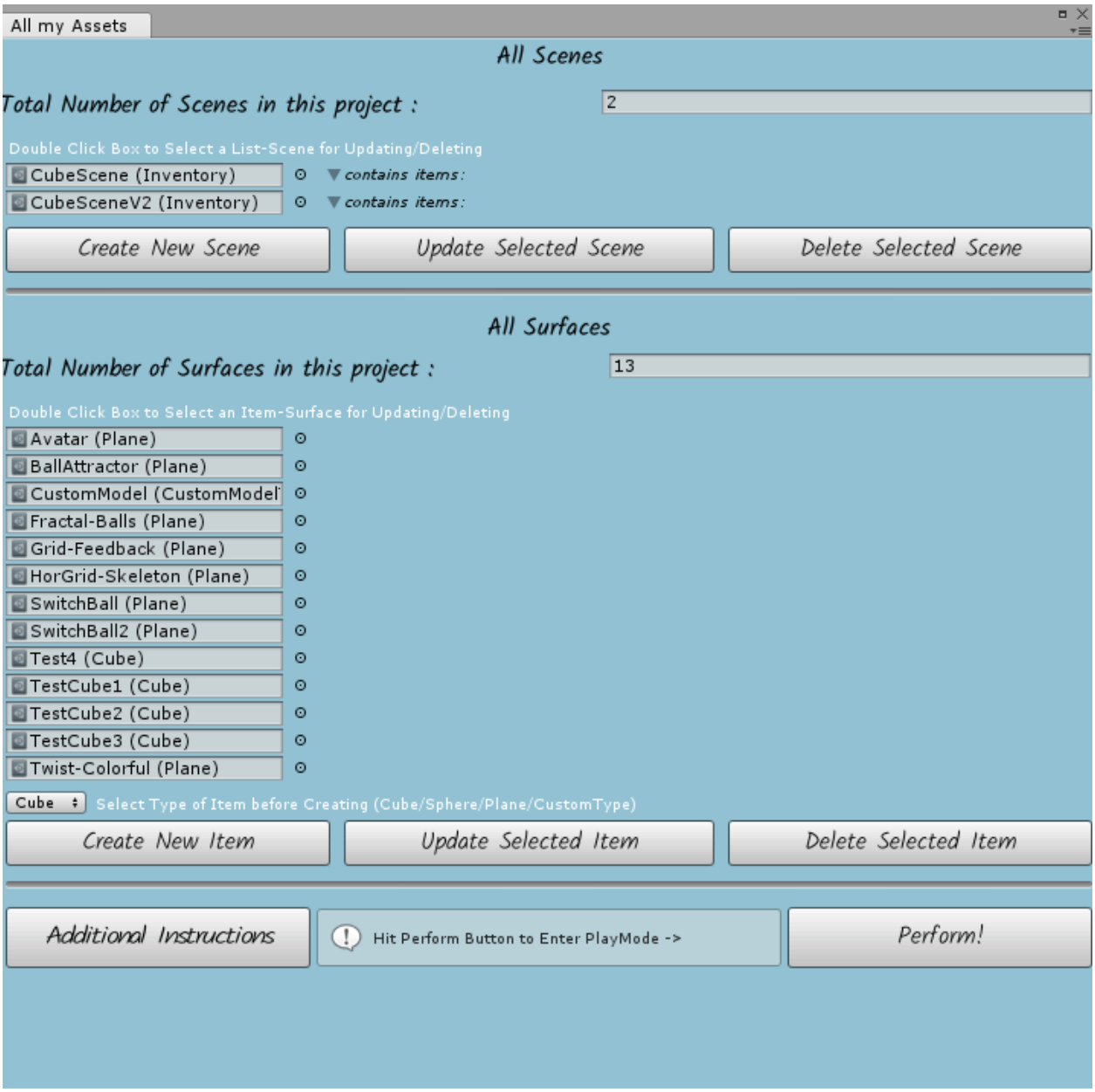


Figure 90: SceneWizard's Main Menu

All the fields and layouts of the window were developed from scratch, using Unity's "EditorGUI" and "EditorGUILayout" Classes. To draw all the controls in our Editor Window, we use the "OnGUI()" function. "OnGUI()" is called for rendering and handling GUI(Graphical User Interface) events, as a result "OnGUI()" implementation might be called several times per frame (one call per event). For each event that corresponds to an input from the user, such as key presses or mouse actions, our GUI gets repainted.

Additionally, we wanted to customize our window with color texture, to avoid Unity's tedious theme, as

well as creating a flexible environment for displaying our items, with a scroll-view. To create our custom color for our Editor Window, we needed to bypass Unity's License permissions, which forbid color options, unless one holds the Pro Licence. However, we could create our own 2DTexture[6] of a specific size (1,1) and apply a declared custom color on each pixel of the Texture, by "Texture2D.SetPixel()" Method. Afterwards, we draw this texture within the SceneWizard's window rectangle, with "GUI.DrawTexture()" Method. The code snippet below, shows how we succeeded this:

Listing 7: Editing our GUI

```
void OnGUI() //OnGUI gets called like an Update... very very often
{
    if (GUI.changed)
    {
        Repaint();
    }
    if (Event.current.type == EventType.MouseMove)
    {
        Repaint();
    }
    Event e = Event.current; // Event Validation for deleting Assets
   // DrawEmpty(new Rect(0, 0, 820, 720));
    //Get position of the window so i can reposition it when Update is selected so that
        InspectorGUI is visible
    Rect r = position;
    scrollPos = EditorGUILayout.BeginScrollView(scrollPos, GUILayout.ExpandHeight(true))
        ;
    Color color= new Color32(0, 191, 255, 65);

    // color = Color.black;
    //GUILayout.Label("Position: " + r.x + "x" + r.y);//To help me find the positions i
        want
    Texture2D tex = new Texture2D(1, 1);
    tex.alphaIsTransparency = false ;
   //TopRow
    tex.SetPixel(0, 0, color);

    tex.Apply();
    GUIStyle mySkin = new GUIStyle() ;

    mySkin.normal.background = tex;
    mySkin.font = myFont;
    GUI.DrawTexture(new Rect(0, 0, 1280, 720), tex);
```

In behalf of displaying our assets, we created our own GUI styles, as we can see in the example below. The example exhibits the styles for displaying the window's first two titles, as well as the assisting information below these titles. Firstly, we declare our new GUI Style and activate the Rich Text feature, in order to incorporate multiple font styles and sizes. Then, we apply our imported font style and adjust the Text's alignment to be in Middle-Centre. The text gets actually displayed by the "LabelField()" Method. Then, we adjust the space between the text, by adding space of 10 pixels, in order to display the next title and begin a horizontal area, so we can display two separate fields in the same line(title-text and integer of total Scene-Versions). Similarly, we display the sub-note for selecting the Scene-Versions.

Listing 8: Displaying all the Assets

```
//Display Title
GUIStyle styleTitle = new GUIStyle();
```

[6]Textures are image or movie files that lay over or wrap around GameObjects, to give them a visual effect.

```
styleTitle.richText = true;
styleTitle.font = myFont2;
// styleTitle.fontStyle = FontStyle.BoldAndItalic;
// styleTitle.fontSize = 12;
styleTitle.alignment = TextAnchor.MiddleCenter;
//GUILayout.Label("<b><size=13><color=black>All the InventoryLists of assets that
    represent your scene</color></size>", styleTitle);
EditorGUILayout.LabelField("All Scenes", styleTitle, GUILayout.ExpandWidth(true));
GUILayout.Space(10);
//Display additional information
EditorGUILayout.BeginHorizontal();
GUIStyle style = new GUIStyle
{
    richText = true
};
style.font = myFont2;
GUILayout.Label("Total Number of Scenes in this project :", style);
//GUILayout.Label("Total Number of Asset Lists", style, GUILayout.ExpandWidth(true)); //
    Making a label in our vertical view, declaring its contents, and adding editor flare
    .
EditorGUILayout.IntField(guids.Length);
EditorGUILayout.EndHorizontal();

GUILayout.Space(5); //The indent for our next line, in pixels.
EditorGUILayout.LabelField("Double Click Box to Select a List-Scene for Updating/
    Deleting", EditorStyles.whiteMiniLabel, GUILayout.ExpandWidth(true));
```

To display the list of all our versions of a scene, we followed a pattern, as well for displaying the list of the surface-items. To find all the significant asset files in our project we utilize the "AssetDatabase" Class of Unity, which is an interface for accessing assets and performing operations on assets. Specifically, we programmed the compiler to look for the assets in specific paths with destination the significant folders "SceneItemsLists" folder for the Scene-Versions assets and "Items" folder for the Items-Surfaces assets. The code snippet below shows how we display each version of a scene, by utilizing the "ObjectField()" Method, which receives any type of Unity Object plus few lines of code for debugging, if selecting is happening appropriately:

Listing 9: Displaying all the Assets as ObjectFields

```
EditorGUILayout.BeginVertical(); //Declaring our first part of our layout, and adding a
    bit of flare with EditorStyles.
int i = 0; // to be able to access selectedList elements
foreach (string guid in guids)
{
    string pathToAsset = AssetDatabase.GUIDToAssetPath(guid);
    if (!string.IsNullOrEmpty(pathToAsset))
    {
        Assets = AssetDatabase.LoadAssetAtPath<UnityEngine.Object>(pathToAsset);
        assetNames = AssetDatabase.LoadAssetAtPath<UnityEngine.Object>(pathToAsset).name
            ;
        assetNamesList.Add(AssetDatabase.LoadAssetAtPath<UnityEngine.Object>(pathToAsset
            ).name);
        assetsPaths.Add(pathToAsset);
        if (guid != null) //Preventing null errors if the objects are removed.
        {
            EditorGUILayout.BeginHorizontal(); //Adding a horizontal view to indent our
                next line so that the properties look like children, LOTS of things you
                can add like toggles and stuff to reduce this clutter.
            EditorGUILayout.ObjectField(Assets, typeof(Inventory), false, GUILayout.
                MaxWidth(200)); //Declaring our object as an object field, with the type
```

```
                    of object we want, and allowing scene object.

                    //selectedList[i] = EditorGUILayout.Toggle(selectedList[i] == true);//
                    Display toggle to make selectable
16              GUILayout.Space(5); //The indent for our next line, in pixels.
17              GUI.SetNextControlName(assetNames);
18              GUI.SetNextControlName("asset" + i);
19          }
```

To display the elements populating each version of a scene, we check if the scene is created brand new with the help of an assisting boolean variable "isNew", which is true when a new Scene-Version gets created and switches to false when this Scene-Version gets updated and saved. If the Scene-Version is brand new and the user still has not set not even the size of the containing elements-items, a warning message is displayed(see Figure:76 below, indicating that the current Inventory list is new. If the Scene is created and set in size, but has not been populated yet with any element, then a warning message gets displayed in the "contains items" field of this Scene-Version, indicating the corresponding slots are empty, thus need to be filled(see Figure:78. Below the figures, we can see the code snippet that performs this action:

Listing 10: Displaying all the Assets Contents

```
1  GUILayout.BeginVertical();
2  foldout[i] = EditorGUILayout.Foldout(foldout[i], contains, styleContains);
3  if (foldout[i] && (GUI.GetNameOfFocusedControl() == "asset" + i))
4  {
5      inv = (Inventory)Assets;
6
7      if ((inv.isNew) && (inv.inventory.Length == 0))
8      {
9          EditorGUILayout.HelpBox("List is Empty", MessageType.Warning);
10     }
11     //Debug.Log(inv.name);
12     for (int q = 0; q < inv.inventory.Length; q++)
13     {
14         if (inv.inventory[q].item == null)
15         {
16             EditorGUILayout.HelpBox("Slot is Empty", MessageType.Warning);
17         }
18         else
19         {
20             EditorGUILayout.LabelField(inv.inventory[q].item.ItemName, styleContains2);
21         }
22     }
23 }
24 GUILayout.EndVertical();
25 ...
26 if (inv != null)
27 {
28     if (inv.isNew)
29     {
30         EditorGUILayout.HelpBox("List is New. Click 'Update Selected List' to continue."
               , MessageType.Warning);
31     }
32 }
```

To continue with, the next thing that was developed, had been the Button controls. A specific style was created to customize our Buttons, as we can see in the example for “Update” Button below. We declared custom colors to create a 2DTexture, in order to customize our Buttons when they are highlighted with the

Mouse, by applying this custom Texture on the Button's background, when the Button is "onHover". The rest command define the fonts and alignment of the text in our Button.

Listing 11: Button Styles

```
//Display of bottom buttons
//GUIStyle for Bottom  Buttons
Color colorButton = new Color32(123, 104, 238, 20);
Color borderColor = new Color32(75, 0, 130, 255);
Color LettersColor = new Color32(75, 0, 130, 255);
Color ButtonLettersColor = new Color32(0, 191, 255, 255);
Texture2D texButton = new Texture2D(101, 101, TextureFormat.ARGB32, true);
Texture2D texButton2 = new Texture2D(10, 50, TextureFormat.ARGB32, true);
for (int w = 0; w < texButton.width; w++)
{
    for (int h = 0; h< texButton.height; h++)
    {
        if ((w == 0) || (h == 0) || (w == tex.width-1) || (h == tex.height-1) )
        {
            texButton.SetPixel(w, h, borderColor);
        }
        else
        {
           //texButton.SetPixel(w, h, colorButton);
        }
    }
}
// texButton2.SetPixel(texButton2.width, texButton2.height, ButtonColor);
texButton.Apply();
texButton2.Apply();
GUIStyle style1Button = new GUIStyle(GUI.skin.button);
style1Button.richText = true;
style1Button.font = myFont;
style1Button.normal.textColor = Color.black;
style1Button.hover.background = texButton;
style1Button.hover.textColor = LettersColor;
style1Button.alignment = TextAnchor.MiddleCenter;
//style1Button.margin = new RectOffset(0, 0, 0, 0); //removes the space between items (
    previously there was a small gap between GUI which made it harder to select a
    desired item)
// String to Display for Button1
string button1 = "Update Selected Scene";
```

All the rest buttons, were developed in a similar pattern. We will inspect the implementation for each button and its functionality through the code.

"Create New Scene" Button: When the user clicks this button, our main Window gets repositioned, so that the user can clearly see the created .asset file in the appropriate folder of our Project View "SceneItemsLists". The file needs to be named instantly, as well as updated. A warning message displays that the user will need to update this selected file and populate it with items-surfaces, as we mentioned before. The figure:75 demonstrates an example this action.

The folder "SceneItemsLists" contains all the .asset files that correspond to the versions of a scene. When the user tries to create a new file in this path, our code checks if the path is valid and creates a new file with a default name given from code.

With the function "GUILayout.Button()", which is a boolean Function returning true when the user

clicks the button, we create a single press Button that can be pressed and released as a normal Button. When this Button is released the function returns the expected true value. If the mouse is moved off the button it is not clicked. The functions parameters display the Button's text and style.

When the Button gets pressed, the main window get re-positioned in our 2D display few pixels on the right, so the user can re-name the asset appropriately.
The compiler checks the existence of a folder with the "AssetDatabase.IsValidFolder(givenPath)" Method, which, given a path to a folder(SceneItemsLists), returns true if it exists, false otherwise. If the folder does not exist, we create it and also create the asset file with the default name. However, we encountered an issue, while creating the asset file with the classic "AssetDatabase.CreateAsset()" Method. Although the method created the asset file, focused the ProjectView window and selected the significant file, in order to mimic the default Unity's Assets behavior and create a file that is automatically selected for renaming, after delving into the issue in Unity forums and blogs, we ended up utilizing "ProjectWindowUtil.CreateAsset()" Function, which meets our demands. In the sequel, we save the created Asset, by calling the aforementioned "Save()" Function, implemented in our "Inventory" Script. The code snippet below, demonstrates how we implemented this button:

Listing 12: Create New Scene Version Button Implementation

```
GUILayout.BeginHorizontal();
    //Create New List
    if (GUILayout.Button(button2, style1Button))
    {
        r.x = 339;
        r.y = 28;

        position = r;//Reposition the window so i can see the Inspector on the right
        string path2 = "Assets/SceneItemsLists/" + "AnInventoryList.asset";
        //if path exists creates asset file in the folder
        if (AssetDatabase.IsValidFolder(path))
        {
            Debug.Log(path);
            Inventory inventoryGO = new Inventory();
            inventoryGO = CreateInstance<Inventory>();
            //string path = "Assets/SceneItemsLists/";
            string[] duplicationHelp = AssetDatabase.FindAssets("AnInventoryList.asset")
                ;
            if (duplicationHelp.Length > 1)
            {
                Debug.Log("File Already Exists! Update its settings to customize it!");
                EditorGUILayout.HelpBox("File Already Exists! Update its settings to
                    customize it!", MessageType.Warning);
            }
            ProjectWindowUtil.CreateAsset(inventoryGO, path2);
            inventoryGO.isNew = true;
            inv = inventoryGO;
            EditorGUIUtility.PingObject(inventoryGO);
            Debug.Log(inventoryGO.InvName);
            //AssetDatabase.SaveAssets();
            inventoryGO.Save();
            //Directory.CreateDirectory(path);
        }
        //if path doesnot exist, creates folder and asset file in the folder
        else
        {
            Directory.CreateDirectory(path);
            Inventory inventoryGO = new Inventory();
```

```
37              inventoryGO = CreateInstance<Inventory>();
38              //string path = "Assets/SceneItemsLists/";
39              ProjectWindowUtil.CreateAsset(inventoryGO, path2);
40              inventoryGO.isNew = true;
41              //AssetDatabase.SaveAssets();
42              inventoryGO.Save();
43          }
44      }
45      GUILayout.Space(5); //The indent for our next line, in pixels.
```

"Update Selected Scene" Button: When the user clicks on the Update Button, a new editor window pops up as a new tab, next to our main window, as we can see in the Figure:91.

When the Button gets clicked, we firstly import any assets that have changed their content modification data or have been added-removed to the project folder, by refreshing our Asset Database. Then, we call a function called: "ParameterWindow2(inv.name)", which opens the InventoryInspector custom Editor Window, created from another Script in our Editor folder. Simultaneously, this function passes the targeted Scene-Version's name to the InventoryInspector, so all the modifications and updates happen to the correct asset file. The function pops the new tab by the "ShowUtility()" function, which shows the InventoryInspector Editor Window as a floating utility window. When the utility window loses focus it remains on top of the new active window and is always in front of normal Unity windows.

Furthermore, in the Click Button Field, we check if the targeted Object is anyhow "null"(i.e. uninitialized, undefined, empty, or meaningless value), we advise the user to select an asset before clicking the Update Button. If the target is not null, we call the "ApplyModifiedProperties()" Method, which flushes any changes made to the serialized properties accessed within this data stream. Moreover on this, in the Scripts that control the Editing of the Scene-Versions and Items-Surfaces, we call the "targetObject.Update()" Function at the beginning of the "OnGUI()" implementation, since we keep a reference to a "SerializedObject" instance for more than one frame. Thus, before we read any data from it, as one or more target objects may have been modified elsewhere, such as from a separate SerializedObject stream, we must manually call this method to refresh the modifications made. Respectively, these two different SerializedObject streams with the same target objects are independent from one another and we must manually synchronize them in this fashion if one or more of them is maintained over the course of several frames.

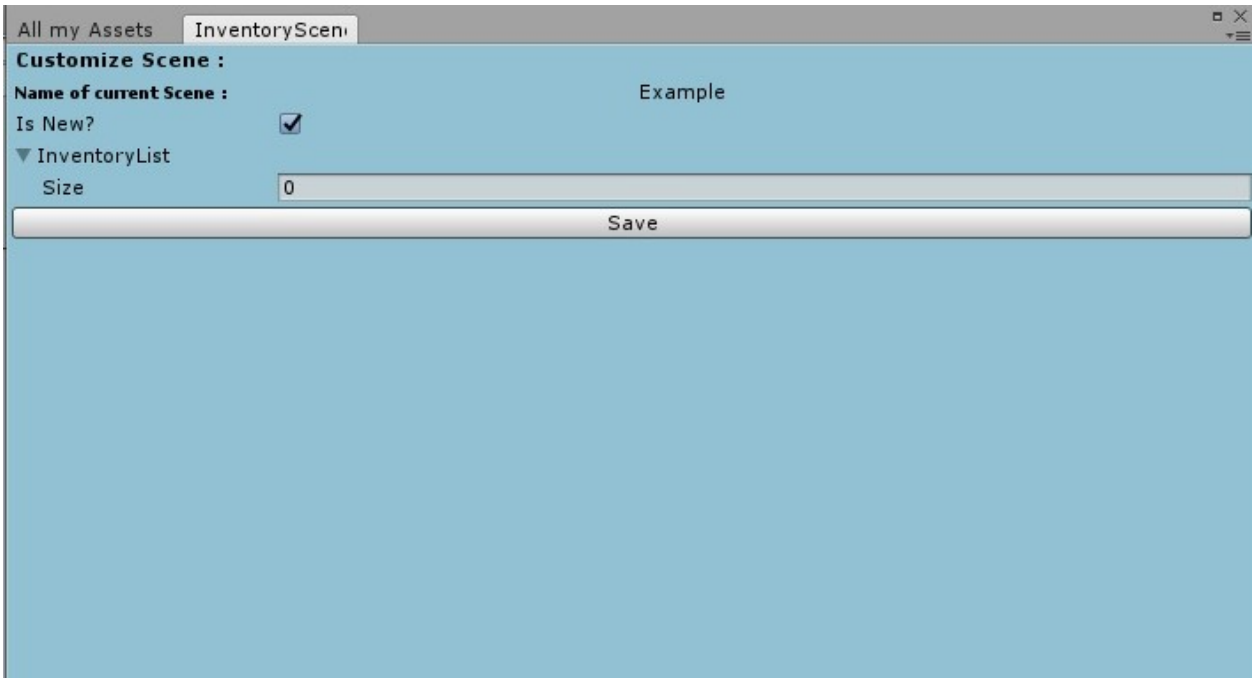


Figure 91: SceneWizard: Update Selected Scene Example

After the user, sets up the size of this version of the scene, which indicates how many items-surfaces this specific version will contain, the user can populate each element, with items-surfaces the project contains by clicking the small circle next to each object field, as we can see in the Figure84:

The last thing, is to save this scene version, by clicking the "Save" Button, which marks our scene as "not new" anymore. When clicking this Button the InventoryInspector window closes and the focus returns on the SceneWizard's main window. (See the Implementation code in Listing: 15

The code snippet below shows the implementation of the "Update Selected Scene" Button:

Listing 13: Saving the Scene Version after clicking Update Button.

```
1  if (GUILayout.Button(button1, style1Button)) //Button Update
2      {
3          //r.x = 0;
4          //r.y = 29;
5          //position = r;//Reposition the window so i can see the Inspector on the right
6          AssetDatabase.Refresh();
7          ParameterWindow2(inv.name);
8          AssetDatabase.Refresh();
9          if (targetObject != null)
10         {
11             targetObject.ApplyModifiedProperties();
12         }
13         else
14         {
15             EditorGUILayout.HelpBox("Please Select a List", MessageType.Error);
16         }
17     }
```

```
18      GUILayout.Space(5);
```

The new tab that gets popped, when "Update Selected Scene" Button is pressed, constitutes another custom Editor Window, we created for this purpose in a same way we created our main window, as we can see in the code snippet below:

Listing 14: Update Selected Scene: Editor Window Tab

```
1  public class InventorySceneInspector : EditorWindow
2  {
3      public Inventory inventory;
4      public SerializedObject targetObject;
5      public SerializedProperty theList;
6      public SerializedProperty isnew;
7      public string assetPath;
8
9      public static void ShowWindow()
10     {
11         EditorWindow window = EditorWindow.GetWindow(typeof(InventorySceneInspector));
12     }
13
14     public void OnEnable()
15     {
16         targetObject = new SerializedObject(Selection.activeObject);
17         assetPath = AssetDatabase.GetAssetPath(Selection.activeObject.GetInstanceID());
18     }
19     public void OnGUI()
20     {
21         if (GUI.changed)
22         {
23             Repaint();
24         }
25         if (Event.current.type == EventType.MouseMove)
26         {
27             Repaint();
28         }
29         Color color = new Color32(0, 191, 255, 65);
30         Texture2D tex = new Texture2D(1, 1);
31         tex.alphaIsTransparency = true;
32         //TopRow
33         tex.SetPixel(0, 0, color);
34         tex.Apply();
35         GUIStyle mySkin = new GUIStyle();
36         //mySkin.normal.background = tex;
37         GUI.DrawTexture(new Rect(0, 0, 1280, 720), tex);
38         EditorGUILayout.LabelField("Customize Scene :", EditorStyles.boldLabel);
39         GUILayout.BeginHorizontal();
40         EditorGUILayout.LabelField("Name of current Scene :", EditorStyles.miniBoldLabel
               );
41         EditorGUILayout.LabelField(targetObject.targetObject.name);
42         GUILayout.EndHorizontal();
```

When the "Save" Button gets pressed, all the modifications get saved and our Asset Database gets refreshed, to accept the changes. In the sequel, the current window closes and the user returns to the main menu. The code snippet below, shows us how we implemented this action:

Listing 15: Save Scene Version after Update Implementation

```
1  if (GUILayout.Button("Save")) // In order to save the new name
```

```
2           {
3               Debug.Log(targetObject.targetObject.name);
4               myInventory.isNew = false;
5               targetObject.ApplyModifiedProperties();
6               AssetDatabase.SaveAssets();
7               EditorWindow window = EditorWindow.GetWindow(typeof(InventorySceneInspector)
                    );
8               window.Close();
9           }
10      }
11      targetObject.ApplyModifiedProperties();
12      ...
13  }
```

"Delete Selected Scene" Button: When the user clicks on this Button, a new window pops up, asking the user to reassure the deletion of the selected version of a scene, as we can see in the figure below:

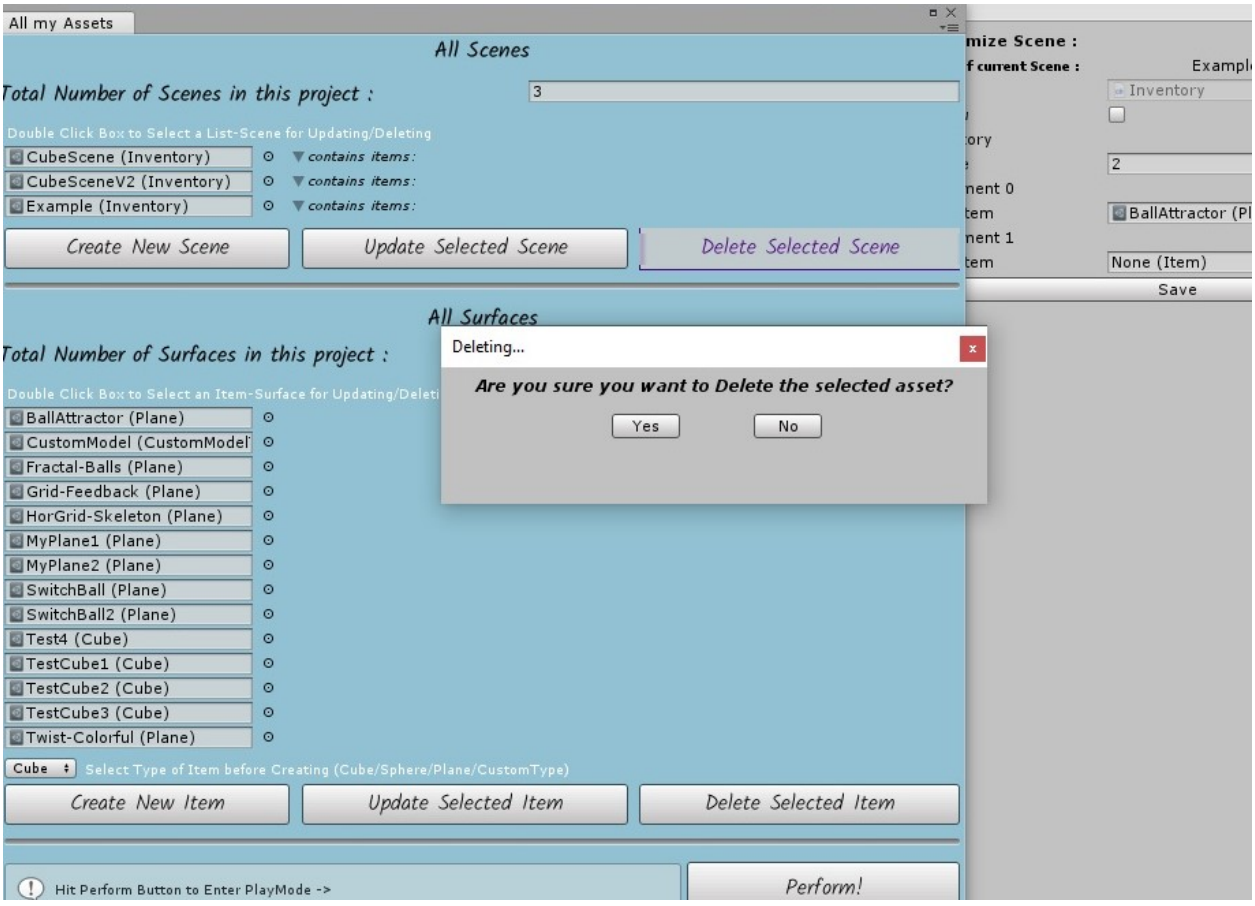


Figure 92: SceneWizard: Delete Selected Scene Example

The verifying window, is another custom editor window, we created to serve our purpose, in the same

way we created the previous two custom editor windows. This script, contains a public variable called "seletionsWinID", which in the "ShowWindow()" Function, is set to call the "GetInstanceID()" method. The latter, returns instance id of the current window, which is unique, in order to give us the ability to close the correct window, when the user is done with updating the selected Scene-Version.
The implemented code is shown in the two following Listings:

Listing 16: Delete Scene Version Check

```
1  public class DeleteCheckEditor : EditorWindow
2  {
3      public static int yes = -1;
4      public static int deletionWinID = 0;
5
6      public static DeleteCheckEditor delInspectorInstance
7      {
8          get { return GetWindow<DeleteCheckEditor>(); }
9      }
10
11     public static void ShowWindow()
12     {
13         EditorWindow deleteInspector = GetWindowWithRect(typeof(DeleteCheckEditor), new
               Rect(0, 0, 400, 100), true);
14         GUIContent titlecontent = new GUIContent("Deleting..."); //Creating some title
               content.
15         deleteInspector.titleContent = titlecontent;
16         deleteInspector.Show();
17         deletionWinID = deleteInspector.GetInstanceID();
18         Debug.Log(deletionWinID);
19     }
```

Similarly to our main window implementation, we create GUI styles, to display the titles, as well as the verification question. When the user, clicks “Yes” the selected asset file gets deleted by sending and event on the main window’s instance. This gets done by calling the "SendEvent()" Method, whick looks up the main window and passes the selected Event, which in our case is a Command Type Event, utilizing the miscellaneous GUI helper "EditorGUIUtility".
Otherwise, if the user clicks “No”, nothing gets deleted.
After the user selects each of the "Yes/No" Option Buttons, the verification window closes and the user returns focus on our main window, which gets repainted new. The second picture of the code for this window Listing:17, shows in detail how we implemented these actions.

Listing 17: Delete Scene Version Check OnGUI() Function

```
1  private void OnGUI()
2  {
3      GUIStyle style = new GUIStyle();
4      style.richText = true;
5      style.fontStyle = FontStyle.BoldAndItalic;
6      style.fontSize = 12;
7      style.alignment = TextAnchor.MiddleCenter;
8      GUILayout.Space(5);
9      EditorGUILayout.LabelField("Are you sure you want to Delete the selected asset?",
           style, GUILayout.ExpandWidth(true));
10     GUILayout.Space(10);
11     GUILayout.BeginHorizontal();
12     GUILayout.Space(125);
13     GUIStyle buttonYes = new GUIStyle(GUI.skin.button);
14     string forButtonYes = "Yes";
15     buttonYes.fixedWidth = 50;
```

```
16      buttonYes.richText = true;
17      GUIStyle buttonNO = new GUIStyle(GUI.skin.button);
18      string forButtonNo = "No";
19      buttonNO.richText = true;
20      buttonNO.fixedWidth = 50;
21
22      if (GUILayout.Button(forButtonYes, buttonYes))
23      {
24          yes = 1; // Set integer to 1 if yes was clicked
25          Debug.Log(yes);
26          InventoryInspector findMainWindow = GetWindow<InventoryInspector>(false, "
                InventoryInspector", true);
27          findMainWindow.Focus();
28          //InventoryInspector.InvInspector.Show();
29          int mainWinID = findMainWindow.GetInstanceID();
30          Debug.Log(mainWinID);
31          Debug.Log(findMainWindow.name);
32
33          findMainWindow.SendEvent(EditorGUIUtility.CommandEvent("Delete"));
34          Debug.Log(EditorGUIUtility.CommandEvent("Delete").commandName);
35          Debug.Log(findMainWindow.SendEvent(EditorGUIUtility.CommandEvent("Delete")));
36          //To close Help window after Validation
37          EditorWindow tempWind = GetWindow(typeof(DeleteCheckEditor));
38          tempWind.Close();
39
40      }
41      GUILayout.Space(50);
42      if (GUILayout.Button(forButtonNo, buttonNO))
43      {
44          yes = 0; //set integer to 0 if no was clicked
45          Debug.Log(yes);
46          //To close Help window after Validation
47          EditorWindow tempWind = GetWindow(typeof(DeleteCheckEditor));
48          tempWind.Close();
49      }
50      GUILayout.Space(125);// Total Width of this line in pixels = 400
51      GUILayout.EndHorizontal();
52
53  }
```

The main window receives the event, if needed and performs the appropriate actions for deleting the selected asset file from the project's folder. The current event that's being processed right now, is declared at the beginning of our main window's "OnGUI()" Method. In the Delete Button Click Check Field the compiler check if the currently received event, corresponding to the user's inputs, has the significant name of the "ValidateCommand" Event, set from the "DeleteCheckEditor" Script(See Listing:17).

The Listing:18, demonstrates the implementation of "Delete Selected Scene" Button.

Listing 18: Delete Scene Version Main Window Event

```
1  //Delete asset when button is pressed
2          if (GUILayout.Button(button3, style1Button))
3          {
4              Debug.Log(DeleteCheckEditor.yes);// To check selection but i do it in
                   Populate()
5              DeleteCheckEditor.yes = -1; //Reset the integer
6              DeleteCheckEditor.ShowWindow();
7          }
8          if (e.commandName == "Delete")
```

```
        {
            Debug.Log(e.commandName);
            Debug.Log("DeleteReceived");
            string pathToDeleteObj = "Assets/SceneItemsLists/";
            Debug.Log(AssetDatabase.IsValidFolder(pathToDeleteObj));
            if (AssetDatabase.Contains(inv))
            {
                //Debug.Log(pathToDeleteObj + inv.name);
                AssetDatabase.DeleteAsset(pathToDeleteObj + inv.name + ".asset");
                AssetDatabase.Refresh();
                DeleteCheckEditor.yes = -1; //Reset the integer
            }
            else
            {
                Debug.Log("Asset was deleted Successfully!");
            }
        }
```

In the same pattern, we developed corresponding actions for our surface-items. As we have already mentioned in the Use Cases chapter, the user can create a new surface-item, update it, or delete it. The implemented code for these buttons, is similar to the corresponding Buttons for the Scene Versions. The user also selects the type of Item-Surface he/she desires to create. We will inspect the implementation for each button below:

"Select Type" Button: When this Button gets clicked, a list of options drops down, guiding the user on what type of item he would like to create. This Button needs to be set to a type, every time the user needs to create a new Item-Surface. Each and every option of this Button corresponds to an integer called "index", which is set to values [0-4] after the user clicks an option. As a result, when we implement the "Create New Item" Button, we utilize switch-case statements[7], which act like streamline conditionals for our different types of Items. We can see how the button is displayed in the Figure:73

"Create New Item" Button: In a same manner, as with the Scene Versions, when this button gets clicked, the main window gets repositioned and the project's view focuses on the "Items" folder, giving the user the possibility to name the new Item, as we can see in the Figure:74.

The created item depends on what type of Item the user selects from the Drop-down Button. In the implemented code we split the actions into cases, to create the items appropriately according to the user's selected type, as we can see in the code snippet below:

Listing 19: Create New Item Button Script

```
 //Create Item Button was pressed
if (GUILayout.Button(button23, style1Button))
{
    r.x = 339;
    r.y = 28;

    position = r;//Reposition the window so i can see the newItem on the right
    string path1 = "Assets/Items/";
    string path2 = "Assets/Items/" + "AnItem.asset";
    switch (index)
    {
```

[7]They are useful for compare a single variable against a series of constants.

```
12          case 0: // Cube case
13              //if path exists creates asset file in the folder
14              if (AssetDatabase.IsValidFolder(path1))
15              {
16                  //Transform trans = null;
17                  //trans.position = new Vector3(0, 0, 0);
18                  Debug.Log(path);
19                  Cube itemGO = CreateInstance<Cube>();
20                  item = itemGO;
21                  itemGO.itemType = Item.ItemType.Cube;
22                  //ItemInstance itemyGO = new ItemInstance("AnItem.asset",1,trans,true
                        ,"","","");
23                  ProjectWindowUtil.CreateAsset(itemGO, path2);
24
25                  EditorGUIUtility.PingObject(itemGO);
26                  AssetDatabase.SaveAssets();
27
28                  //Directory.CreateDirectory(path);
29              }
30              //if path doesnot exist, creates folder and asset file in the folder
31              else
32              {
33                  Directory.CreateDirectory(path1);
34                  Cube itemGO = CreateInstance<Cube>();
35                  item = itemGO;
36                  itemGO.itemType = Item.ItemType.Cube;
37                  //ItemInstance itemyGO = new ItemInstance("AnItem.asset",1,trans,true
                        ,"","","");
38                  ProjectWindowUtil.CreateAsset(itemGO, path2);
39
40                  EditorGUIUtility.PingObject(itemGO);
41                  AssetDatabase.SaveAssets();
42              }
43              break;
44          case 1: // Sphere case
45          ...
46          ...
47          ...
```

"Update Selected Item" Button: When the user clicks on this Button, a new editor window pops up as a new tab, next to our main window.
The user needs to set up these parameters, in order to customize the selected surface, as we have described in the Use Cases. The names of the Visuals, the user will type, should match the corresponding names, given in TouchDesigner(See Figure:82). The position, rotation and scale points are given, according to the

real-scene set-up parameters, as already mentioned in the Use Cases section. However, even if the virtual surfaces, are not perfectly aligned in the projection, to the real surfaces, our tool allows the user to "correct" these imperfections, by manually aligning the surface's Mesh, during Play Mode and before the performance. We will discuss this implementation in detail in the next section 5.2.3. The "SelectPhysicalRepresentation" Object Field, is mostly useful when the user wants to spawn a custom model surface in the scene. The new tab that gets popped, when "Update Selected Scene" Button is pressed, constitutes another custom Editor Window, we created for this purpose in a same way we created our main window, as we can see in the code snippet below:

Listing 20: Update Selected Item Button Script

```
public class ItemInspector : EditorWindow {
    private Inventory myInventory;
    public SerializedObject targetObject;
    SerializedProperty list;
    SerializedProperty nameOfItem;
    private SerializedProperty spawnPositionPoints;
    private SerializedProperty spawnRotationPoints;
    private SerializedProperty itemType;
    private SerializedProperty physicalItem;
    private SerializedProperty itemSize;
    [SerializeField]
    public List<Vector3> spawns = new List<Vector3>();
    public string assetPath;
    public string physicalItemName;
    [SerializeField]
    public int size = 0;
    public int index = -1;
    public string[] options = new string[] { "HandsAboveHead", "Jump", "RaiseHand_Left",
         "RaiseHand_Right", "Punch_LeftHand", "Punch_RightHand", "SwipeToLeft_BothHands"
        , "SwipeToLeft_LeftHand", "SwipeToLeft_RightHand", "SwipeToRight_BothHands", "
        SwipeToRight_LeftHand", "SwipeToRight_RightHand", "SwipeDown", "SwipeUp" };
    public GUIContent content = new GUIContent ( "SelectPhysicalRepresentation" );
    bool showBtn = false;
    //private Font myfont = ;

    public static void ShowWindow()
    {
        EditorWindow window = EditorWindow.GetWindow(typeof(ItemInspector));
    }

    public void OnEnable()
    {

        targetObject = new SerializedObject(Selection.activeObject);
        assetPath = AssetDatabase.GetAssetPath(Selection.activeObject.GetInstanceID());
    }
```

We implemented the same style of colors as our main window, as we can see in the snippet below below:

Listing 21: Update Selected Item Editor Window Tab

```
 public void OnGUI()
    {
        if (GUI.changed)
        {
```

```
5               Repaint();
6           }
7           if (Event.current.type == EventType.MouseMove)
8           {
9
10              Repaint();
11          }
12          Color color = new Color32(0, 191, 255, 65);
13          Texture2D tex = new Texture2D(1, 1);
14          tex.alphaIsTransparency = false;
15          //TopRow
16          tex.SetPixel(0, 0, color);
17          tex.Apply();
18          GUIStyle mySkin = new GUIStyle();
19          mySkin.normal.background = tex;
20          GUI.DrawTexture(new Rect(0, 0, 1280, 720), tex);
21          targetObject.Update();
22          EditorGUILayout.LabelField("Customize Object :", EditorStyles.boldLabel);
```

For each case-type of an item we display the appropriate parameters. "GetType()" Method returns the type of the Selected active GameObject. If the type mathes the significant ones, the corresponding item's fields get displayed. the gesture options are implemented as a Drop-down Button, in a same manner with the "SelectionType" Button.
The following two figures show an example of the Cube Type:

Listing 22: Update Cube Item Implementation

```
1 if (Selection.activeObject.GetType() == typeof(Cube))
2           {
3               Cube myCube = (Cube)Selection.activeObject;
4               myCube.itemName = EditorGUILayout.TextField("Item name ", myCube.itemName);
5               myCube.visual = EditorGUILayout.TextField("Initial Visual name ", myCube.
                    visual);
6               GUILayout.BeginHorizontal();
7               EditorGUILayout.LabelField("Select Gesture from Gesture Database>",
                    GUILayout.ExpandWidth(true));
8               GUILayout.Space(5);
9               index = EditorGUILayout.Popup(index, options, GUILayout.MaxWidth(120));
10              switch (index)
11              {
12                  case 0:
13                      myCube.gesture = EditorGUILayout.TextField(options[index]);
14                      break;
15                  case 1:
16                      myCube.gesture = EditorGUILayout.TextField(options[index]);
17                      break;
18                      ...
19                      ...
20                      ...
21                      }
```

When the user wants to set two or more surfaces that will display the same initial visual, get triggered by the same gesture and display also the same triggered visual, he/she has the possibility to do so by adding a corresponding integer in the Size/Rotation and Scale Integer Field, indicating how many items will get spawned in a current Scene-Version with the same parameters.
It is reasonable that these integers match for a specific surface, so we implemented Warning messages for the user's facilitation. Furthermore, in order to display the Vector3 fields as lists, we utilized the "EditorGUILayout.PropertyField()" Method. Otherwise, the standard Method: "EditorGUILayout.Vector3Field()" did not

allow us to set multiple items.
In order maintain the correct physical model of an item, we check the value of an item's reference "physicalItem" property and display it accordingly for reassurance, as we can see in the second figure of the same example:

Listing 23: Update Fields Implementation

```
myCube.newVisual = EditorGUILayout.TextField("Triggered Visual ", myCube.newVisual);
            EditorGUILayout.BeginHorizontal();
            EditorGUILayout.PrefixLabel("Item Type :");
            EditorGUILayout.LabelField(myCube.itemType.ToString());
            EditorGUILayout.EndHorizontal();

            spawnPositionPoints = targetObject.FindProperty("spawnPoints");
            EditorGUILayout.PropertyField(spawnPositionPoints, new GUIContent("Position
                Points"), true);
            spawnRotationPoints = targetObject.FindProperty("spawnRotationPoints");
            EditorGUILayout.PropertyField(spawnRotationPoints, new GUIContent("Rotation
                Points"), true);
            itemSize = targetObject.FindProperty("spawnSize");
            EditorGUILayout.PropertyField(itemSize, new GUIContent("Scale"), true);
            if (spawnPositionPoints.arraySize != spawnRotationPoints.arraySize)
            {
                EditorGUILayout.HelpBox("Position and Rotation Vecors must match!",
                    MessageType.Warning);
            }
            else if (spawnPositionPoints.arraySize != itemSize.arraySize)
            {
                EditorGUILayout.HelpBox("Position and Scale Vecors must match!",
                    MessageType.Warning);
            }
            else if (spawnRotationPoints.arraySize != itemSize.arraySize)
            {
                EditorGUILayout.HelpBox("Rotation and Scale Vecors must match!",
                    MessageType.Warning);
            }
            itemType = targetObject.FindProperty("itemType");
            //EditorGUILayout.PropertyField(itemType, new GUIContent("Physical Item"),
                true);
            physicalItem = targetObject.FindProperty("physicalRepresentation");
            EditorGUILayout.PropertyField(physicalItem, content, GUILayout.ExpandWidth(
                true));
            targetObject.ApplyModifiedProperties();
            if (physicalItem.objectReferenceValue == null)
            {
                myCube.physRep = EditorGUILayout.TextField("Physical Item is missing");
            }
            else
            {
                myCube.physRep = EditorGUILayout.TextField("Selected Item >",
                    physicalItem.objectReferenceValue.name, GUILayout.ExpandWidth(false)
                    );
            }
```

Finally, when the user is ready with customizing the Item-Surface, the item gets saved by clicking the "Save&CloseTab". This Button Click Field implements code for renaming the file, according to the "Item Name" input TextField, marking the targeted object as dirty, refreshes and saves the asset. Additionally, the ItemInspector window closes and the main window gets focused again. The code that implements this process is shown in the figure below:

Listing 24: Save Updated Item

```
if (GUILayout.Button("Save&CloseTab")) // In order to save the new name
            {
                AssetDatabase.RenameAsset(assetPath, myCube.itemName);
                //AssetDatabase.CreateAsset(item, nameOfItem.ToString());
                EditorUtility.SetDirty(Selection.activeObject);
                targetObject.ApplyModifiedProperties();
                AssetDatabase.SaveAssets();
                EditorWindow window = GetWindow(typeof(ItemInspector));
                window.Close();
            }
```

"Delete Selected Item" Button: When the user clicks on this Button, a new window pops up, asking the user to reassure the deletion of the selected Item-Surface, in a same fashion we did with deleting the selected Scene-Version, as we can see in the figure:83.

The verifying window, is another custom editor window, we created to serve our purpose, in the same way we created the previous verification editor window for deleting a scene version.

Finally, the implementation of the Button that initializes the performance("Perform" Button), as well as the implementation of the visual separating lines on our main Editor Window, along with the guiding Help Box, are shown in the listing:25 below. To enter Play Mode by clicking the "Perform!" Button, we call the "EditorApplication.ExcecuteMenyItem()" Method, which Invokes the menu item in the specified path, in our case "Edit/Play".
To draw the visual line that separates the different areas in our main window, we call the "DrawUILine()" Method, which we implemented to draw a rectangular, initially by taking the actual control of the Rect, without relying on GUILayout, and later draw the Rect with the appropriate features, to look as it is.

Listing 25: Perform Button Implementation

```
GUILayout.Space(3);
        GUILayout.BeginHorizontal();
        EditorGUILayout.HelpBox("Hit Perform Button to Enter PlayMode ->", MessageType.
            Info, true);
        if (GUILayout.Button(button25, stylePlayButton, GUILayout.ExpandWidth(false)))
        {
            EditorApplication.ExecuteMenuItem("Edit/Play");
        }
        GUILayout.EndHorizontal();
        EditorGUILayout.EndScrollView();
    }
    void OnSelectionChange() { Populate(); Repaint(); }
    void OnEnable() { Populate(); }
    void OnFocus() { Populate(); }


    public static void DrawUILine(Color color, int thickness = 2, int padding = 10)
    {
        Rect r = EditorGUILayout.GetControlRect(GUILayout.Height(padding + thickness));
        r.height = thickness;
        r.y += padding / 2;
        r.x -= 2;
        r.width += 6;
        EditorGUI.DrawRect(r, color);
    }
```

5.2 Using SceneWizard Tool

5.2.1 Introduction

As we have already mentioned, our tool is used both in Edit and Play Mode, in order to set-up all the necessary parameters. As a result, after the user creates all surfaces and scene versions he/she would like to perform, the simulation needs to get initialized before the performance, in order to correct any imperfections caused by the image registration problem, which we discussed in the second chapter of our text. In this section we will discuss in detail, how we implemented the tool's functionality, as well as the actions the user can perform to avoid the image registration problem.

5.2.2 Functionality

As we previously mentioned in the Use Cases chapter, the user in Play Mode has the possibility to:

•Press "Space" → to iterate through all versions of scene existing in "SceneItemsLists" Folder.

•Press "S" → to Save the Mesh coordinates of a manipulated Surface.

•Press "R" → to Reset the state (initial visual) of the item's-surface's texture/visual after a gesture has been performed.

To apply this functionality, we will first discuss how we implemented the frontend part of our Assets Inventory System. The main script that controls the frontend representation of the items is "Slot" (a MonoBehaviour) and needs to be attached on an empty GameObject. In the provided sample Scene, "Slot" is attached on the empty GameObject called Spout, along with "Spout" Script.

When the script gets accessed, the code initializes all the significant Asset files, in our project, and the first (on the list of scenes) version of a scene gets spawned in our virtual Scene. To initialize the Scenes Asset, we utilize the "FindAssets()" Method, which returns an array of matching Assets, as GUIDs. The latter are Globally Unique Identifiers for each Asset, which are referenced from the Asset files to uniquely identify them within a single project and between different projects. Then, for each Asset in the GUIDs array, we load the asset as a Unity Object with "LoadAssetAtPath<>()", in order to be able to switch Scene-Versions during the performance. The next step is, to initialize the Scene-Version's contained items-surfaces and spawn them in our virtual, and consequently in our real scene.
The "isEdited" boolean variable assists in loading the correct Mesh for our Surfaces, in case the user has already mapped and saved the Surfaces of the virtual scene to the real scene.

It is reasonable to name the version of scenes appropriately in the sequence the user desires, since the list is ordered alphabetically. In the future work of our project we will facilitate this process by allowing the user to control which scene version to spawn and when. Precisely, after the compiler initializes all the asset files, InitializeScene(Inventory MyInvetory) function gets called, to begin spawning the first scene version, in our virtual scene, as we can see in the code snippet below:

Listing 26: Initializing the Assets Data Implementation

```
1 public void OnEnable()
2     {
```

```
        string[] guids = AssetDatabase.FindAssets("t: Inventory", null); //Collecting
            our asset info
        string[] paths = new string[guids.Length];
        for(int i=0; i<guids.Length; i++)
        {
            string pathToAsset = AssetDatabase.GUIDToAssetPath(guids[i]);
            if (!string.IsNullOrEmpty(pathToAsset))
            {
                invs[i] = (Inventory)AssetDatabase.LoadAssetAtPath<UnityEngine.Object>(
                    pathToAsset);
                isEdited[i] = false;
                Debug.Log(invs[i].name);
                q = i+1;
            }
        }
        InitializeScene(invs[p]);
    }
```

In the "InitializeScene()" Function, we firstly load all the Prefabs (3DModels) existing in the "Prefabs" Folder of our project, in order to get the Mesh of a specific 3D Model, in case the user selects to spawn a custom Model of a surface. This Model needs to allow access to its Mesh Component, concerning the success of this process.
For each element of the current Inventory (Scene Version), we access each Item-Surface with a new ItemInstance and get the item's parameter variable (with "GetItem()" Method, set from "ItemInspector" Editor Window, to later set these parameters as features of the Physical Representation of each Item-Surface in our virtual scene, as we can see in the listing below:

Listing 27: Access Assets Data

```
public void InitializeScene(Inventory MyInventory)
    {
        sizeList = MyInventory.inventory.Length;
        Debug.Log(sizeList);
        prefabs = Resources.LoadAll<GameObject>("Prefabs/");
        Debug.Log("HOW MANY PREFABS I HAVE?");
        Debug.Log(prefabs.Length);
        for(int j = 0; j < prefabs.Length; j++)
        {
            Debug.Log(prefabs[j].name);
        }
        for (int i = 0; i < sizeList; i++)
        {
            int currentElement = 0; //adjust position and rotation arrays simultaneously
            foreach (Vector3 spawn in MyInventory.inventory[i].item.spawnPoints)
            {
                ItemInstance instance;
                if (MyInventory.GetItem(i, out instance, MyInventory.inventory[i].item.
                    itemName, MyInventory.inventory[i].item.visual, MyInventory.
                    inventory[i].item.gesture, MyInventory.inventory[i].item.newVisual))
                {
                    Debug.Log(MyInventory.inventory[i].item.itemName);
```

For each Item-Surface, we create a GameObject in real-time and set the parameters to match the user's inputs from the Editor Window. Additionally, we attach the necessary components for displaying the surface, such as the "MeshRenderer" and "MeshFilter" Components, as well as set some parameters of the "Rigidbody" Component appropriately, so that out surface remains in its position. These parameters are the GameObject's position, rotations and size.

Moreover, we disable the "useGravity" Rigidbodies' feature, which controls whether gravity affects a Rigidbody, so that our Item behaves as in outer space (does not fall from the set y position). In addition to this we activate the "isKinematic" feature, which controls whether physics affects the Rigidbody. When this feature is enabled, is enabled, forces, collisions or joints will not affect the Rigidbody anymore. We did this, for the reason that, whenever any Surfaces have been tangent with each other, if "isKinematic = false" resulted in our Surfaces bouncing randomly around due to the collision point.
The code snippet below shows this implementation, inside "InitializeScene()" function's boy:

Listing 28: Front-end Representation of Assets

```
MyInventory.inventory[i].item.physicalRepresentation = new GameObject(MyInventory.
    inventory[i].item.itemName);
MeshRenderer[] defaultObjs = MyInventory.inventory[i].item.physicalRepresentation.
    GetComponentsInChildren<MeshRenderer>();
Debug.Log("SubPrefabs");
Debug.Log(defaultObjs.Length);
MyInventory.inventory[i].item.physicalRepresentation.transform.position = spawn;
MyInventory.inventory[i].item.physicalRepresentation.transform.localEulerAngles =
    MyInventory.inventory[i].item.spawnRotationPoints[currentElement]; //or transform.
    rotation = Quaternion.Euler(spawnRotationPoints)
MyInventory.inventory[i].item.physicalRepresentation.transform.localScale = MyInventory.
    inventory[i].item.spawnSize[currentElement];
                                                        //MyInventory.inventory[i].item.
    physicalRepresentation.AddComponent<PhysicalInventory>();
MyInventory.inventory[i].item.physicalRepresentation.AddComponent<Rigidbody>().
    useGravity = false;
MyInventory.inventory[i].item.physicalRepresentation.GetComponent<Rigidbody>().
    isKinematic = true;
MeshFilter filter = MyInventory.inventory[i].item.physicalRepresentation.AddComponent<
    MeshFilter>();
Mesh Mesh = filter.Mesh;
```

Furthermore in the "InitializeScene()", our compiler checks if the Meshes of the Items have been modified and saved previously, in order to spawn the correct Mesh for each Item-Surface. This is done by utilizing the "File.Exists(path)" Function, which determines whether the specified file exists, given the path. If the file exists, we load the already saved Mesh, which is basically an Asset file too, but specified as a Mesh Type Asset. However, if the current loading Item-Surface has not yet been modified and saved, the compiler spawns a default Mesh, according to the selected Item Type, followed by adding "Collider" Components, according to the Item's Type, which define the shape of an Object(Item-Surface) for the purposes of physical collisions. These Components are necessary, since they contain the Object's Material Properties, needed for displaying Texture on the Object.
We can see this implementation in the listing below:

Listing 29: Mesh of Surface according to Item Type

```
 if (File.Exists("Assets/Meshes/"+MyInventory.inventory[i].item.itemName+".asset"))
{
    filter.Mesh = (Mesh)AssetDatabase.LoadAssetAtPath("Assets/Meshes/"+MyInventory.
        inventory[i].item.itemName+".asset", typeof(Mesh)) as Mesh;
}
else if (isEdited[p] == false)
{
    if (MyInventory.inventory[i].item.itemType == Item.ItemType.Cube)
    {
        filter.Mesh = SetCubeVertices(Mesh);
        MyInventory.inventory[i].item.physicalRepresentation.AddComponent<BoxCollider>()
            ;
```

```
11
12      }
13      else if (MyInventory.inventory[i].item.itemType == Item.ItemType.Sphere)
14      {
15          filter.Mesh = SetSphereVertices(Mesh);
16          MyInventory.inventory[i].item.physicalRepresentation.AddComponent<SphereCollider
                >();
17      }
18      else if (MyInventory.inventory[i].item.itemType == Item.ItemType.Plane)
19      {
20          filter.Mesh = SetPlaneVertices(Mesh);
21          MyInventory.inventory[i].item.physicalRepresentation.AddComponent<MeshCollider
                >();
22      }
```

If the Meshes have not been edited and saved yet, our code checks, the type of the Item-Surface and generates the geometry of a default Mesh, according to the primitive type Meshes. We set each Mesh features, with respect to the type of primitive. These features include:

•**Vertices:** Each vertex is a point in 3D space.

•**Normals:** The directional vector of a vertex or a surface. This characteristically points outward, perpendicular to the Mesh surface.

•**Lines/Edges:** The invisible lines that connect vertices to one another.

•**Triangles:** Formed when three vertices are connected by edges.

•**UV Map:** Maps a material to an object to give it texture and color.

The anatomy of a 3D Object starts with its Mesh. The construction of this Mesh starts with its vertices. The invisible lines that connect these vertices form triangles, which define the basic shape of the object. Normals and UV data then provide shading, color and texture. These Mesh data is stored in the "MeshFilter" Component, and the "MeshRenderer" uses this data to draw the object in the scene.

as we can see in the following two figures the example of Cube primitive's Mesh:

Listing 30: Mesh of Surface according to Item Type: Plane Type Example

```
1  public Mesh SetPlaneVertices(Mesh Mesh)
2  {
3      Mesh myMesh = Mesh;
4      //myMesh.Clear();
5
6      float length = 1f;
7      float width = 1f;
8      int resX = 2; // 2 minimum
9      int resZ = 2;
10
11     #region Vertices
12     Vector3[] vertices = new Vector3[resX * resZ];
13     for (int z = 0; z < resZ; z++)
14     {
15         // [ -length / 2, length / 2 ]
16         float zPos = ((float)z / (resZ - 1) - .5f) * length;
17         for (int x = 0; x < resX; x++)
18         {
19             // [ -width / 2, width / 2 ]
```

```
            float xPos = ((float)x / (resX - 1) - .5f) * width;
            vertices[x + z * resX] = new Vector3(xPos, 0f, zPos);
        }
    }
    #endregion

    #region Normales
    Vector3[] normales = new Vector3[vertices.Length];
    for (int n = 0; n < normales.Length; n++)
        normales[n] = Vector3.up;
    #endregion

    #region UVs
    Vector2[] uvs = new Vector2[vertices.Length];
    for (int v = 0; v < resZ; v++)
    {
        for (int u = 0; u < resX; u++)
        {
            uvs[u + v * resX] = new Vector2((float)u / (resX - 1), (float)v / (resZ - 1)
                );
        }
    }
    #endregion

    #region Triangles
    int nbFaces = (resX - 1) * (resZ - 1);
    int[] triangles = new int[nbFaces * 6];
    int t = 0;
    for (int face = 0; face < nbFaces; face++)
    {
        // Retrieve lower left corner from face ind
        int i = face % (resX - 1) + (face / (resZ - 1) * resX);

        triangles[t++] = i + resX;
        triangles[t++] = i + 1;
        triangles[t++] = i;

        triangles[t++] = i + resX;
        triangles[t++] = i + resX + 1;
        triangles[t++] = i + 1;
    }
    #endregion

    myMesh.vertices = vertices;
    myMesh.normals = normales;
    myMesh.uv = uvs;
    myMesh.triangles = triangles;

    myMesh.RecalculateBounds();
    //myMesh.Optimize();
    return myMesh;
}
```

However, if we have a Custom Model Type, the code initializes the Item's Mesh, as reported by the prefab's "MeshFilter" Component, imported from our "Prefabs" Folder, in the Project View, by assigning the "sharedMesh" from the Model to our Item's Mesh. Note that the prefab need to guarantee access to this specific Component for succeeding this process.
The code implementing this process is shown below:

Listing 31: Mesh of CustomType Surface

```
else if (MyInventory.inventory[i].item.itemType == Item.ItemType.CustomModelType)
{
    foreach (GameObject prefab in prefabs)
    {
        Debug.Log(MyInventory.inventory[i].item.physRep);
        Debug.Log(prefab.name);
        if (MyInventory.inventory[i].item.physRep == prefab.name)
        {
            //MeshRenderer prefabMeshRenderer = prefab.GetComponent<MeshRenderer>();
            MeshFilter prefabMesh = prefab.GetComponent<MeshFilter>();
            filter.Mesh = prefabMesh.sharedMesh;
            MyInventory.inventory[i].item.physicalRepresentation.AddComponent<
                MeshCollider>();
        }
    }
    Debug.Log("-------------------CustomModel-----------------------");
}
```

In order for Spout Protocol to display our textures properly, we needed first to add “SpoutReceiver” Component for each Item-Surface, in addition to the “MeshStudy” Component which allows us to modify the items’ Mesh, as shown in the code snippet below:

Listing 32: Adding Significant Components Implementation

```
    myMaterial = MaterialFactory.MaterialCreator();
    MyInventory.inventory[i].item.physicalRepresentation.AddComponent<MeshRenderer>().
        material = myMaterial;
    MyInventory.inventory[i].item.physicalRepresentation.AddComponent<SpoutReceiver>();
    MyInventory.inventory[i].item.physicalRepresentation.AddComponent<MeshStudy>();
    }
currentElement++;
```

Moreover, we needed to create a Class that generates a material on Run-time, for each item, so that Spout can display the textures. For this reason, we created "MaterialFactory" Class, which loads a valid Material from the Resources folder. The valid Materials were copied from the Demo folders of Spout unitypackage. We can see the "MaterialFactory" Class in the figure below:

Listing 33: Material Factory Class

```
public class MaterialFactory : MonoBehaviour {
    public Shader shader;
    public Material material;
    //public Texture texture;
    //public Color color;
    void Start()
    {
        Renderer rend = GetComponent<Renderer>();
    }
   public static Material MaterialCreator()
    {
        return new Material(Shader.Find("Unlit/Texture")); ;
    }
}
```

Next, our "Update()" function, controls all the functionality of our tool. More specifically, in the Update() we check, if the item’s texture has been spawned appropriately, by utilizing the "hasSpawnedTexxture"

boolean variable, which is set to true when the "SpawnItemTexture()" Function gets accessed, as we can see in the listing below:

Listing 34: Checking Spout Receivers

```
 void Update()
{
    if ((Spout.Spout.isReceiving == true)&& (hasSpawnedTexxture == false))
    {
        SpawnItemTexture(invs[p]);
    }
    ...
}
```

If the Item-Surface has not yet had a texture, "SpawnItemTexture()" function gets called. This function finds all the significant Items-surfaces in our current scene and accesses Spout's Component, by finding all the Scene Objects that contain "SpoutReceiver" Component, in order to set the textures name according to the Initial Visual's name, the user has previously typed in our Tool's Editor. Below we can see this implementation in our code:

Listing 35: Checking Spout Textures

```
public void SpawnItemTexture(Inventory MyInventory)
{
    sizeList = MyInventory.inventory.Length;
    for (int i = 0; i < sizeList; i++)
    {
        //Spout.Spout.instance.addListener(TexShared, TexStopped);
        foreach (Vector3 spawn in MyInventory.inventory[i].item.spawnPoints)
        {
            ItemInstance instance;
            if (MyInventory.GetItem(i, out instance, MyInventory.inventory[i].item.
                itemName, MyInventory.inventory[i].item.visual, MyInventory.inventory[i
                ].item.gesture, MyInventory.inventory[i].item.newVisual))
            {
                Debug.Log(MyInventory.inventory[i].item.itemName);
                Debug.Log(MyInventory.inventory[i].item.visual);
                //newCode
                SpoutReceiver[] findMyself = FindObjectsOfType<SpoutReceiver>();
                for (int j = 0; j < findMyself.Length; j++)
                {
                    if (findMyself[j].name == MyInventory.inventory[i].item.itemName)
                    {
                        findMyself[j].GetComponent<SpoutReceiver>()._sharingName =
                            MyInventory.inventory[i].item.visual;
                    }
                }
                Debug.Log(MyInventory.inventory[i].item.visual);
            }
        }
    }
    hasSpawnedTexxture = true;
}
```

Afterwards, in a similar manner "Update()" checks, if all the gestures set from the Editor have been loaded into the memory, as long as the "GestureListener" is initialized, so this listener, can "listen" only to the corresponding gestures set from SceneWizard Editor, as we can see in the figure below:

Listing 36: Checking Gesture Strings

```
if (isListening == false)
{
    isListening = BodySourceManager.IsKinectInitialized();
    Debug.Log(isListening);
    Debug.Log(GestureManager);
    Debug.Log("-is GestureSet?");
    Debug.Log(isGestureSet);
    if ((isListening)&&(isGestureSet==false)) {
        SetItemGesture(invs[p]);
    }

}
else if(isListening == true)
{
    if (isGestureSet == false)
    {
        SetItemGesture(invs[p]);
    }

}
```

Similarly to the visuals, to set the gestures information for the current Scene Version, we call "SetItemsGestures()" function, to read the strings of the corresponding gestures names, as we can see below:

Listing 37: Setting Gesture Strings

```
public void SetItemGesture(Inventory MyInventory)
{
    sizeList = MyInventory.inventory.Length;
    for (int i = 0; i < sizeList; i++)
    {
        //Spout.Spout.instance.addListener(TexShared, TexStopped);
        foreach (Vector3 spawn in MyInventory.inventory[i].item.spawnPoints)
        {
            ItemInstance instance;
            if (MyInventory.GetItem(i, out instance, MyInventory.inventory[i].item.
                itemName, MyInventory.inventory[i].item.visual, MyInventory.inventory[i
                ].item.gesture, MyInventory.inventory[i].item.newVisual))
            {
                Debug.Log(GestureManager);
                Debug.Log(MyInventory.inventory[i].item.gesture);
                //CommonGestures.Add(MyInventory.inventory[i].item.gesture);
                foreach (VisualGestureListenerInterface listener in BodySourceManager.
                    Instance.visualGestureListeners)
                {
                    listener.GestureInProgress(i, MyInventory.inventory[i].item.gesture)
                        ;
                }
            }
        }
    }
    isGestureSet = true;
    Debug.Log(isGestureSet);
}
```

The "Update()" function, also monitors the keyboard inputs from the user. If the user presses “Space” key on the keyboard, the compiler loads the next scene version and re-initializes everything. This is done with an assisting integer variable which increases every time the user presses the "Space" key, and returns to

0 when all the Scene-Versions have been spawned at least once. To check if all the Scene-Versions have been accessed, we check if the value of the assisting integer has surpassed the size of our Inventory List (total of the Scene-Versions).
We can see this implementation in the code snippet below:

Listing 38: Monitoring the Keyboard for User Inputs

```
if (Input.GetKeyDown(KeyCode.Space))//Load Next Scene
{
    Debug.Log("-Space Was Pressed-");
    p++;
    Debug.Log(q);
    if (p >= q) // Accessed Whole List
    {
        p = 0;
        Debug.Log(p);
    }
    SpoutReceiver[] oldscene = FindObjectsOfType<SpoutReceiver>();
    Debug.Log(oldscene.Length);
    for (int o = 0; o < oldscene.Length; o++)
    {
        Debug.Log(oldscene[o]);
        Destroy(oldscene[o].gameObject);
    }

    Debug.Log("Load Next Scene");
    Debug.Log(p);
    hasSpawnedTexxture = false;
    isGestureSet = false;
    InitializeScene(invs[p]);
}
```

If the user presses “S” key on the keyboard, the current state of the current Scene Version’s Items’ Meshes gets saved on our Project’s folder “Meshes”, in order to resolve the Image Registration issue, when in time for the performance. This indicates that if the user manually aligns and saves the surfaces in the virtual world, upon the time for the Projection Mapping Performance, the user only needs to press the “Perform!” Button and the virtual scene will be already set and aligned ready for the performance. We can see the body of the "Update()" function, implementing the "Save" action below:

Listing 39: Pressing "S" Key Input Implementtion

```
if (Input.GetKeyDown(KeyCode.S)) //Save Mesh
{
    Debug.Log("-SAVE Mesh-");
    MeshStudy[] oldMesh = FindObjectsOfType<MeshStudy>();
    SaveMesh(invs[p]);
    //Save Mesh Function Call
}
```

When the user hits the "S" key, "SaveMesh()" Function gets called, in order to save the current Mesh parameters of all the Items-Surfaces of the currently spawned Scene-Version and mark the Scene-Version as "edited". This is done by accessing the Items one by one, get the significant Component "MeshStudy", which is the Script that manipulates our Meshes, as well as "MeshFilter". Later, for each item validated by its name, we assign it to an assisting variable array (editMesh[]) to count the number of the Items. Then, for all the counted Items we save the edited Mesh, accessed from the "MeshStudy" Component, in the "Meshes" foled in our project. If this folder doesnot already exist, the compiler creates one and saves the Meshes

as Asset files, to later refresh the Asset Database by utilizing the previously implemented "SaveAssets()" Method.
The function that saves our Meshes for each object is demonstrated in the figure below:

Listing 40: Saving Mesh Deformation Implementation

```
public void SaveMesh(Inventory MyInventory)
{
    string path = "Assets/Meshes/";
    sizeList = MyInventory.inventory.Length;
    isEdited[p] = true;
    for (int i = 0; i < sizeList; i++)
    {
        foreach (Vector3 spawn in MyInventory.inventory[i].item.spawnPoints)
        {
            ItemInstance instance;

            if (MyInventory.GetItem(i, out instance, MyInventory.inventory[i].item.
                itemName, MyInventory.inventory[i].item.visual, MyInventory.inventory[i
                ].item.gesture, MyInventory.inventory[i].item.newVisual))
            {
                MeshStudy[] editMesh = FindObjectsOfType<MeshStudy>();
                MeshFilter mf = MyInventory.inventory[i].item.physicalRepresentation.
                    GetComponent<MeshFilter>();
                for (int k = 0; k < editMesh.Length; k ++)
                {
                    if (editMesh[k].name == MyInventory.inventory[i].item.itemName)
                    {

                        MyInventory.inventory[i].item.spawnMesh = editMesh[k].cMesh;
                        if(!(AssetDatabase.IsValidFolder(path + MyInventory.inventory[i
                            ].item.itemName)))
                        {
                            AssetDatabase.CreateAsset(editMesh[k].cMesh, path +
                                MyInventory.inventory[i].item.itemName+".asset");
                            AssetDatabase.SaveAssets();
                        }
                    }
                }
            }
        }
    }
}
```

To implement the function above, we needed to create a Script called “MeshStudy”, a class we will discuss in the sub-section below.

Finally, we will discuss the implementation of the triggered visuals from performing gestures, as well as the resetting action. In our “CustomGestureManager” script, we implemented some code, to provide the user with the ability of triggering the desired visuals, while performing specific selected gestures, which are provided in our Gesture Database, as well as, “Resetting” the state of the items-surfaces (as if returning to the initial visual after having performed a gesture). When a gesture event arrives, the function “_gestureFrameReader_FrameArrived()” gets executed. In the body of this function, which we have previously discussed, we access the Items-Surfaces of the current version of a scene (in a same way we perform their access in all our Methods) and switch the visual-texture of the item that contains the specific gesture that has been performed, by assigning the appropriate visual name to "SpoutReceiver" significant variable ("_sharingName"), as we can see below:

Listing 41: Gesture Detected Event Implementation

```
Debug.Log("ACCESSING ITEMS");
//Spout.Spout.instance.addListener(TexShared, TexStopped);
foreach (Vector3 spawn in invs[p].inventory[i].item.spawnPoints)
{
    Debug.Log("FOR EACH ITEM");
    ItemInstance instance;
    if (invs[p].GetItem(i, out instance, invs[p].inventory[i].item.itemName, invs[p].
        inventory[i].item.visual, invs[p].inventory[i].item.gesture, invs[p].inventory[i
        ].item.newVisual))
    {
        Debug.Log("GET ITEM");
        Debug.Log("DOES ITEM HAVE ssPOUT rECEIVER?");
       // Debug.Log(MyInventory.inventory[i].item.physicalRepresentation.GetComponent<
           SpoutReceiver>());
        Debug.Log("IS DETECTED GESTURE ASSOCIATED WITH ITEM?");
        Debug.Log(invs[p].inventory[i].item.gesture == gesture.Name);
        if (invs[p].inventory[i].item.gesture == gesture.Name)
        {
            Debug.Log("ITEMS GESTURE = DETECTED GESTURE");
            SpoutReceiver[] findMyself = FindObjectsOfType<SpoutReceiver>();
            for (int k = 0; k < findMyself.Length; k++)
            {
                if (findMyself[k].name == invs[p].inventory[i].item.itemName)
                {
                    Debug.Log("Find item With same Name");
                    findMyself[k].GetComponent<SpoutReceiver>()._sharingName = invs[p].
                        inventory[i].item.newVisual;
                    Debug.Log("REFRESH TEXTURE IF NEEDED");
                    findMyself[k].GetComponent<SpoutReceiver>()._ForceTextureUpdate();
                }
            }
        }
    }
}
```

Note that the visual that comes next, the triggered visual, matches the one the user types in SceneWizard's editor window, the same happens with the gesture. This information get accessed in our script via the "GetGestureUserStrings()" Function, which adds all the gestures found in the Editor to the "gestureNames" String List. An assisting boolean variable ("isGesturesSet" helps us check if the List has been filled during the "Update()").
Below we can see this implementation:

Listing 42: Initialize Database according to Gesture Strings from SceneWizard's UI

```
public void GetGestureUserStrings(Inventory MyInventory)
{
    sizeList = MyInventory.inventory.Length;
    Debug.Log("SizeList Initialized");
    for (int i = 0; i < sizeList; i++)
    {
        Debug.Log("Line1");
        //Spout.Spout.instance.addListener(TexShared, TexStopped);
        foreach (Vector3 spawn in MyInventory.inventory[i].item.spawnPoints)
        {
            Debug.Log("Line2");
            ItemInstance instance;
            if (MyInventory.GetItem(i, out instance, MyInventory.inventory[i].item.
                itemName, MyInventory.inventory[i].item.visual, MyInventory.inventory[i
```

```
                ].item.gesture, MyInventory.inventory[i].item.newVisual))
            {
                Debug.Log("Line3");
                gestureNames.Add(MyInventory.inventory[i].item.gesture);
                Debug.Log(gestureNames.ToString());
            }
        }
    }
    isGesturesSet = true;
}
```

In the "Update()" function we check if the gestures have been set, otherwise we call the "GetGestureUserStrings()" function again, for the current scene version. We similarly count the Scene-Versions to control which Scene will get spawned, according to the number of times the use presses the "Space" key. If the user presses "Space", we clear the "gestureNames" List and re-initialize it with the next Scene-Version's set gestures for each Item-Surface. Simultaneously, we dispose the "gestureFrameSource" and pause the gesture listener, in order to maintain a smooth Kinect functionality. as we can see below:

Listing 43: Monitoring Users Inputs in the Gesture Detector's Implementation Script

```
if (Input.GetKeyDown(KeyCode.Space))
    {
        Debug.Log("-InGESTURES-Space Was Pressed");
        p++;
        Debug.Log(q);
        if (p >= q) // Accessed Whole List
        {
            p = 0;
            Debug.Log(p);
        }
        Debug.Log(p);
        gestureNames.Clear();
        gestureDB.Clear();
        _gestureFrameSource.Dispose();
        _instance._gestureFrameReader.IsPaused = true;

        isGesturesSet = false;
        GetGestureUserStrings(invs[p]);
        InitVisualGestures();
    }
    if (isGesturesSet == false)
    {
        GetGestureUserStrings(invs[p]);

    }
    ...
}
```

Finally, to implement the "Resetting" option, we monitor the user's input in the "Update()" method of "CustomGestureManager" script and call the "Reset()" function if "R" key has been pressed, for the current scene version that is spawned, like this:

Listing 44: Monitoring Users Reset Request

```
if (Input.GetKeyDown(KeyCode.R))
{
    Debug.Log("-Reset was pressed-");
    Reset(invs[p]);
```

```
}
```

In the "Reset()" function, we access the current scene version's items-surfaces and switch the item's texture back to the Initial Visual the user had set from SceneWizard's editor window. Below we can see the implementation of "Reset()":

Listing 45: Reset Surfaces Initial Visuals and Gestures

```
private void Reset(Inventory MyInventory)
{
    Debug.Log("reseting ITEMS");
    sizeList = MyInventory.inventory.Length;
    for (int i = 0; i < sizeList; i++)
    {
        Debug.Log("ACCESSING ITEMS");
        //Spout.Spout.instance.addListener(TexShared, TexStopped);
        foreach (Vector3 spawn in MyInventory.inventory[i].item.spawnPoints)
        {
            Debug.Log("FOR EACH ITEM");
            ItemInstance instance;
            if (MyInventory.GetItem(i, out instance, MyInventory.inventory[i].item.
                itemName, MyInventory.inventory[i].item.visual, MyInventory.inventory[i
                ].item.gesture, MyInventory.inventory[i].item.newVisual))
            {
                Debug.Log(MyInventory.inventory[i].item.itemName);
                Debug.Log(MyInventory.inventory[i].item.visual);
                //newCode
                SpoutReceiver[] findThem = FindObjectsOfType<SpoutReceiver>();
                for (int j = 0; j < findThem.Length; j++)
                {
                    if (findThem[j].name == MyInventory.inventory[i].item.itemName)
                    {
                        findThem[j].GetComponent<SpoutReceiver>()._sharingName =
                            MyInventory.inventory[i].item.visual;
                        findThem[j].GetComponent<SpoutReceiver>()._ForceTextureUpdate();
                    }
                }
            }
        }
    }
}
```

5.2.3 Run-time Mesh Manipulation

When projecting on an arbitrary 3D surface, no matter how the projector is positioned and oriented towards the surface, the resulting image will mostly look distorted. However, there is a point from which the projected image looks perfectly aligned and that is the point position of the projector. Hence, in order to achieve an undistorted look, on an arbitrary surface, we simply have to provide the projector with an image that depicts a view onto that surface from its own (the projector's) position. Considering the real projector being a camera in Unity's virtual space, viewing a virtual replica of the real projection surface. If we project the image this virtual camera "sees", it would fit exactly and look undistorted on the real surface. By defining the camera's (the one that has "SpoutSender" Component attached) parameters, such as the field-of-view, according to our projector's characteristics, as well as setting up the positions, rotations and scaling of the items-surfaces appropriately, using SceneWizard, we trying to achieve exactly this. Although, in theory this works perfectly, the chances are that our virtual scene's parameters will never exactly match our real world

parameters, as some of them, such as the projector's orientation, are hard to measure precisely. Thus, we needed to add some extra functionality in our tool, so as to alleviate this issue.

SceneWizard contains scripts that allow the user to manipulate each Item's – Surface's Mesh, manually and during Run-time, imitating the state-of-the-art tools of VideoMapping. When the user enters Play Mode, he/she has the ability to click on the Item-Surface in Unity's Scene View and select it. As long as the item is selected the user can see some blue point dots, on each and every edge of the item. (See Figures: 86 and 87.)

To achieve this, we needed to add two more scripts to support our tool: "MeshStudy" and "MeshInspector". To begin with, breaking down our implementation, first we need to puzzle out the anatomy of our 3D objects. Conceptually, a Mesh is a construct used by the graphics hardware to draw complex stuff, without a Mesh we cannot see any item in the Scene, as if it is invisible. Meshes contain a collection of vertices that define points in 3D space, plus a set of triangles – the most basic 2D shapes – that invisibly connect these points. In addition to these, normals and UV data provide shading, color and texture. This Mesh data is stored in the Mesh filter, and the Mesh renderer uses this data to draw the object in the scene.

When the user enters Play Mode, our "Slot" script, spawns our Scene versions and attaches the necessary components on our Run-time Objects. These include also the "MeshStudy" Component Script, as we have seen in a previous listing: 32 of our code implementation in the previous sub-section. This script safely clones our Object's Mesh without overwriting its original form, by making a copy of the Mesh from "MeshFilter.sharedMesh" property and assign it back to MeshFilter, in order to acquire the Meshe's attribute's values, as we can see in the code snippet below:

Listing 46: Mesh Deformation Implementation

```
public void InitMesh()
{
    mySelf = this.gameObject;
    oMeshFilter = mySelf.GetComponent<MeshFilter>();
    oMesh = oMeshFilter.sharedMesh;

    cMesh = new Mesh();
    cMesh.name = "clone";
    cMesh.vertices = oMesh.vertices;
    cMesh.triangles = oMesh.triangles;
    cMesh.normals = oMesh.normals;
    cMesh.uv = oMesh.uv;
    cMesh.RecalculateNormals();
    oMeshFilter.Mesh = cMesh;

    vertices = cMesh.vertices;
    triangles = cMesh.triangles;
    isCloned = true;
    Debug.Log("Init \& Cloned");
}
```

Our "MeshStudy" script collaborates with "MeshInspector" script. The second is used to present our Item's – Surface's vertices with the blue colored dots. Since Vertices are stored as an array of Vector3's and triangles are stored as an array of integers that correspond to the indices of its array of vertices, we present those vertices with the implemented code, as seen in the listings below. The "handleTransform" variable gets the Transform values (Position, rotation and scale.) of our Item's Mesh and "handleRotation" gets the current Pivot Rotation mode, which indicates the rotation of the Tool handle[8]. The compiler then, loops

[8]Tools is a Class used to manipulate the tools used in Unity's Scene View.

through the Mesh's vertices and draws the point by calling "ShowPoint()" function. In the body of this function, we convert the vertex's local position into world space and set the color, size and position of each dot, with Handles Utility class. Handles are the 3D controls that Unity uses to manipulate items in the Scene view, useful to us for manipulating procedurally-generated Scene content, such as the Items'-Surfaces' Mesh. "Handles.FreeMoveHandle()" performs an unconstrained movement handle to facilitate the dragging action. This Method returns a Vector3 of the new value modified by the user's interaction with the handle. If the user has not moved the handle, it will return the same value as passed into the function. At the same time, GUI() monitors any changes to detect dragging action. Finally, while dragging a vertex, the "Mesh.DoAction()" will receive its index and Transform values as parameters. Since the vertex Transform values are in world space, we convert them back to local space with "InverseTransformPoint()":

Listing 47: Editing Mesh Implementation

```
void EditMesh()
{
    handleTransform = Mesh.transform;
    handleRotation = Tools.pivotRotation == PivotRotation.Local ? handleTransform.
        rotation : Quaternion.identity;

    for (int i = 0; i < Mesh.vertices.Length; i++)
    {
        ShowPoint(i);
    }
}

private void ShowPoint(int index)
{
    Vector3 point = handleTransform.TransformPoint(Mesh.vertices[index]);

    if (Mesh.moveVertexPoint)
    {
        //draw dot
        Handles.color = Color.blue;
        point = Handles.FreeMoveHandle(point, handleRotation, Mesh.handleSize, Vector3.
            zero, Handles.DotHandleCap);

        //drag
        if (GUI.changed)
        {
            Mesh.DoAction(index, handleTransform.InverseTransformPoint(point));
        }
    }
    else
    {
        Handles.color = Color.cyan;
    }
}
```

The "DoAction()" function, is placed in "MeshStudy" script and performs the dragging by detecting the vertices that share the same position, by calling the "PullSimilarVertices()" Method, as we can see in the code snippet below. This Method, detects the vertices that share the same position, so that when the user pulls only one, the other vertices will not stay behind, and our Mesh will not break and returns those a list of indices as the target vertex. Furthermore, it loops through that list and update the related vertices with "newPos". Then, it assigns the updated vertices back to "cMesh.vertices". Then "RecalculateNormals()" to re-draw the mesh with the new values.

Listing 48: Dragging Vertices Implementation

```
public void DoAction(int index, Vector3 localPos)
{
    PullSimilarVertices(index, localPos);
}

private void PullSimilarVertices(int index, Vector3 newPos)
{
    Vector3 targetVertexPos = vertices[index];
    List<int> relatedVertices = FindRelatedVertices(targetVertexPos, false);
    foreach (int i in relatedVertices)
    {
        vertices[i] = newPos;
    }
    cMesh.vertices = vertices;
    cMesh.RecalculateNormals();
}
```

5.3 Kinect – Building the Gesture Database

5.3.1 Introduction

In advance of implementing Kinect Gesture Recognition and building a Gesture Database, we need to know everything about the sensor's hardware, as well as software, in order to set it up correctly and use its components appropriately. In the sections below we will initially and briefly talk about Kinect's Set-up Mechanism, to continue with building up our custom Gesture Database, using Kinect's features. In the current section (5.1.) we will break down all the implementation of training our Depth Sensor, the algorithm that was used for the training, as well as the built-up and analysis of the Gesture Database.

5.3.2 Kinect Set-up Mechanism

As we have mentioned in the chapter 2.6.4 state-of-the-art for Kinect, its hardware contains an RGB camera, a depth sensor and a four-microphone array, which are able to provide depth signals, RGB images, and audio signals simultaneously. With respect to the software, several tools are available, allowing users to develop products for various applications. In this project have been used Visual Gesture Builder, as well as, Kinect Studio.

5.3.3 Training Kinect via Visual Gesture Builder

With the purpose of controlling projected visuals we implemented a database with custom gestures that we decided. The decision of the gestures was made after an evaluation process of "think aloud", where few architects that utilize Video Mapping tools and perform Projection Mappings have been asked, which gestures they think is comfortable and simultaneously useful to implement. This section provides with an overview of all the steps that were followed in order to build up the Gesture Database.

5.3.4 Implemented Gestures

In the table1 below we present the final implemented gestures, in order to include interaction in a Projection Mapping performance.

Gestures
HandsAboveHead
Jump
RaiseHand_Left
RaiseHand_Right
Punch_LeftHand
Punch_RightHand
SwipeToLeft_BothHands
SwipeToLeft_LeftHand
SwipeToLeft_RightHand
SwipeToRight_BothHands
SwipeToRight_LeftHand
SwipeToRight_RightHand
SwipeDown
SwipeUp

Table 1: SceneWizard's Gesture Database.

The previous gestures are discrete (boolean values). This fact fits best in our current concept of instantly triggering different visuals after the corresponding gesture gets performed. In our future work, we will include Continuous Gestures in order to manipulate visuals continuously and be able to adjust even more parameters of the texture-visual, such as the deformation of the Mesh of the object, its tiling etc.

5.3.5 Recording – Collection of Data

In order to train the camera, to know when our gestures get performed, we needed to record clips, where we perform each individual gesture separately by using Kinect's tool, Kinect Studio. The clips were recorded in .raw format (extended raw file .xrf). We chose this format because, if Kinect happens to change some of the underlying algorithms in generating depth or skeleton, our skeleton data might actually become invalid, if they are process clips (extended event files .xef) and we need to be able to regenerate that with the newest version of depth for our future work on SceneWizard. In addition to that, Visual Gesture Builder requires process clips for the training sessions (tagging, invisualize). As a result, we needed to convert all our clips into process clips, using a provided tool, called "KSConvert.exe". The streams recorded, have been: the depth, IR and body frame data streams from the sensor array. Furthermore, we enriched the clips with a variety of concepts and situations, like, wearing different clothes, recording in different distances or angles to the sensor, with different backgrounds, as an effort in generalizing the performance of the same gesture.

5.3.6 Training VisualGestureBuilder

Once our data collection has been ready, the recorded data was divided into two parts: one for building/training (used for the actual detection) and another for analyzing/testing the gestures (used to test the detector built in the training). A key to continue the process is to use different clips for these two parts in order to test how good the Machine-Learning is at detecting a behavior that it hasn't seen, since the accuracy of the ML algorithms cannot be measured by testing it on the same data that it has been trained with (or it can but the clip will be already known to the detector).Approximately the 80% of the data was used for training and the 20% for testing. After converting the data into process clips, we launched the Visual Gesture Builder tool. Such tool provides a data-driven solution to gesture detection through machine learning. This means that gesture detection is turned into a task of content creation, rather than code

writing. A small gesture database was built using VGB, and using the database has very low run-time costs in terms of memory overhead and CPU processing. The database was slowly built by importing our clips appropriately and in the sequel, these clips were tagged frame by frame to specify a falling incident's true positive and false positive moments. Despite the fact that VGB software is available for developers, it only facilitates the machine learning and training process, not the detection phase. Thus, a system was developed in Unity, to utilize the training set produced by VGB software in order to detect the fall incidents, which we will discuss in the next chapter. In the figures below, we can see some example of tagging clips, by using VGB (Figure:93, Figure:94, Figure:95).

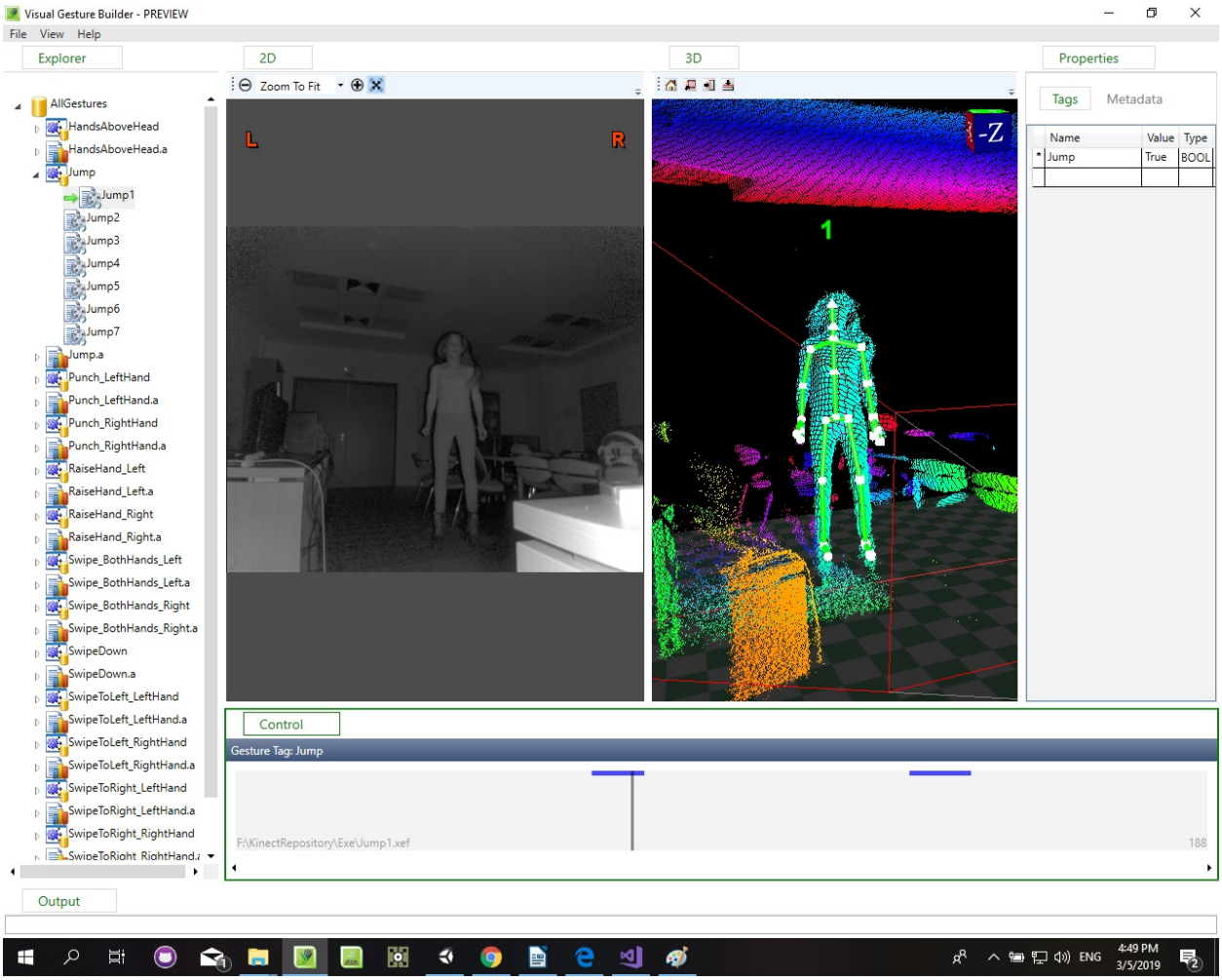


Figure 93: Example Tagging Frames in VGB: True(boolean) frame while performing Jump gesture.

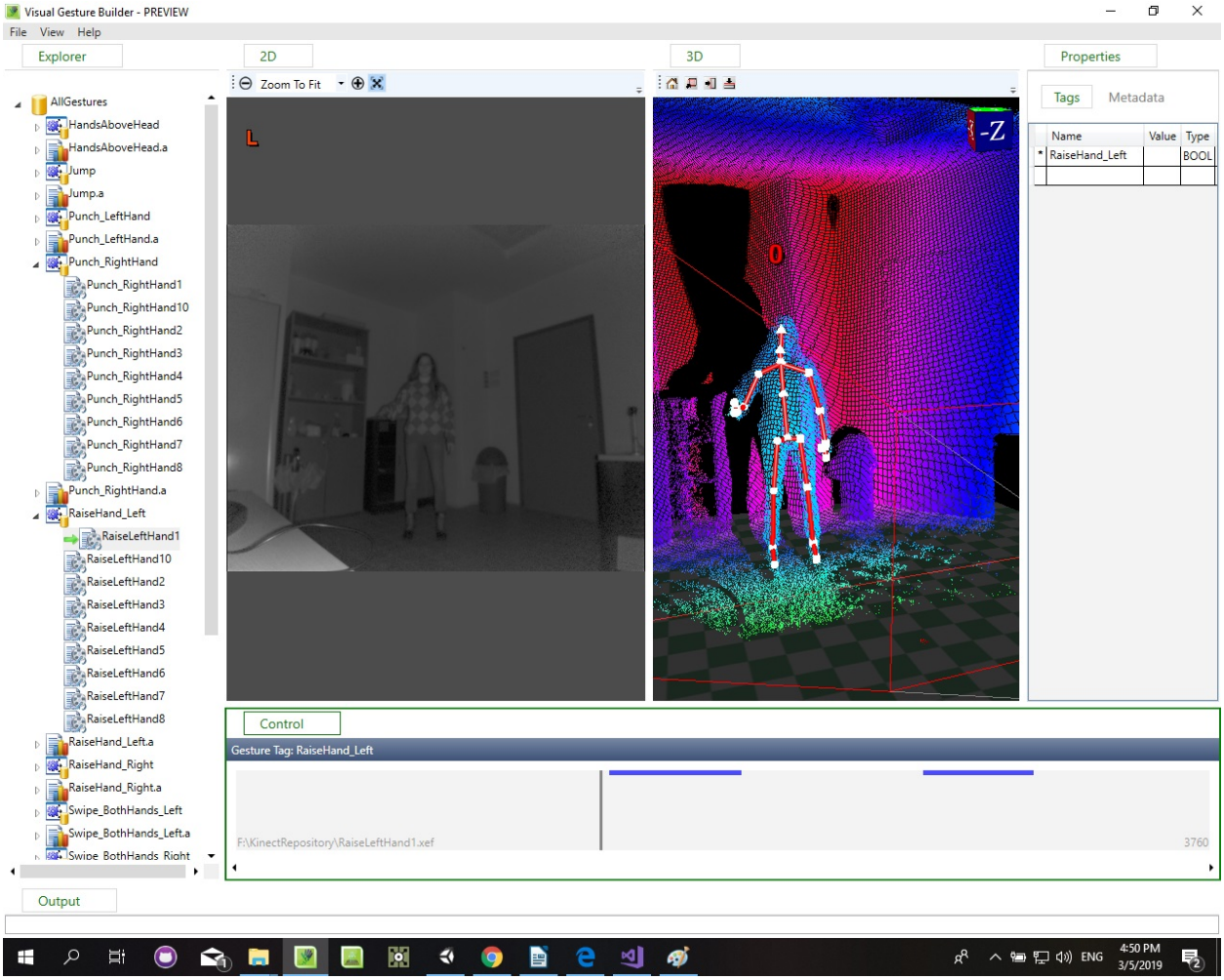


Figure 94: Example Tagging Frames in VGB: False(boolean) frame at the beginning of RaiseHand gesture.

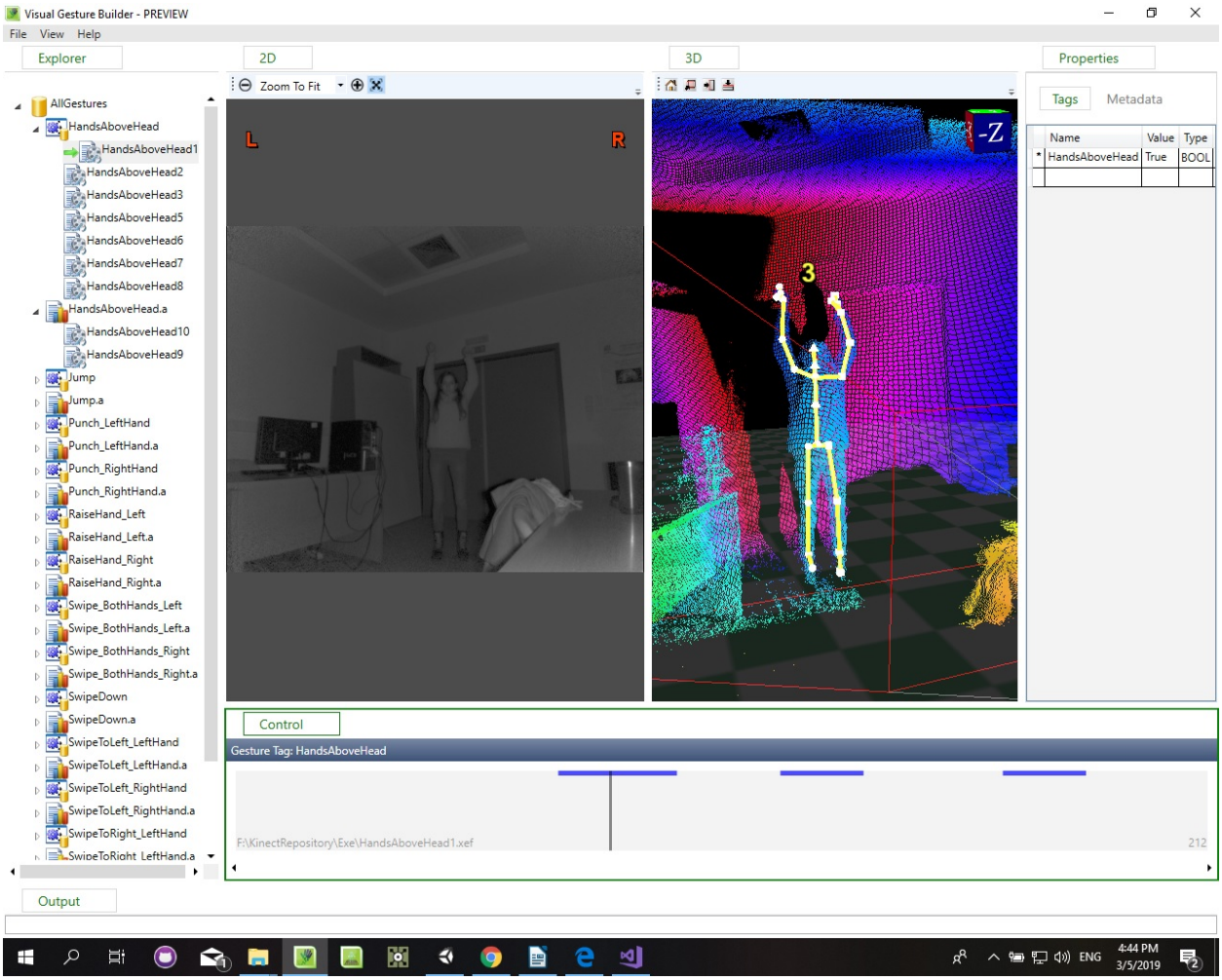


Figure 95: Example Tagging Frames in VGB: True(boolean) result while HandsAboveHead gesture.

5.3.7 Machine Learning

Machine learning (ML) is the scientific study of algorithms and statistical models that computer systems use to effectively perform a specific task without using explicit instructions, relying on patterns and inference instead. It is seen as a subset of artificial intelligence. Machine learning algorithms build a mathematical model of sample data, known as "training data", in order to make predictions or decisions without being explicitly programmed to perform the task. The algorithm has the ability to automatically learn to recognize complex patterns in data, by learning from empirical data examples, and the result is that it can classify data that it has not yet observed in an unsupervised learning process. There are many different approaches to machine learning, such as boosting algorithms, neural networks, decision trees, support vector machines (SVM), Bayesian networks, and so on. In this project, the Adaptive boosting algorithm was used to train discrete gestures.

Adaptive Boosting Algorithm

AdaBoost, short for Adaptive Boosting, is a machine learning meta-algorithm formulated by Yoav Freaund and Robert Schapire, who won the 2003 Goedel Price for their work. It can be used in conjunction with many other types of learning algorithms to improve performance. The output of the other learning algorithms ('weak learners') is combined into a weighted sum that represents the final output of the boosted classifier. AdaBoost is adaptive in the sense that subsequent weak learners are tweaked in favor of those instances misclassified by previous classifiers. AdaBoost is sensitive to noisy data and outliers. In some problems it can be less susceptible to the overfitting problem than other learning algorithms. The individual learners can be weak, but as long as the performance of each one is slightly better than random guessing, the final model can be proven to converge to a strong learner.

In our case, the Adaptive Boosting machine learning algorithm was used to determine when the user was performing a certain gesture. Such detection technology produces a binary or discrete result whether the actor is performing the gesture or not. During the training process it accepted input tags, boolean values, which marked the occurrence of a gesture. This marking or tagging is used to evaluate, whether or not a gesture is actively happening and determines the confidence value of the event. Boosting is an approach in machine learning based on the idea of creating a highly accurate prediction rule by combining many relatively weak and inaccurate rules in order to build up a strong classifier. Intuitively, for a "trained" classifier to be effective and accurate to its predictions, it should meet the following conditions:

- It should b trained on adequate training examples.
- It should provide a good fit to those training examples (low training error).
- It should be "simple".

These conditions, we took into account while in the training process.

As we mentioned before, the algorithm during training time accepts input tags, boolean values, which mark the occurrence of a gesture, such as a hit. The following table describes the most important input parameters that we used when tagging a gesture. We enter these parameters in the Project Settings grid, as shown in the Visual Gesture Builder training project.

Table 2: VGB Input Values

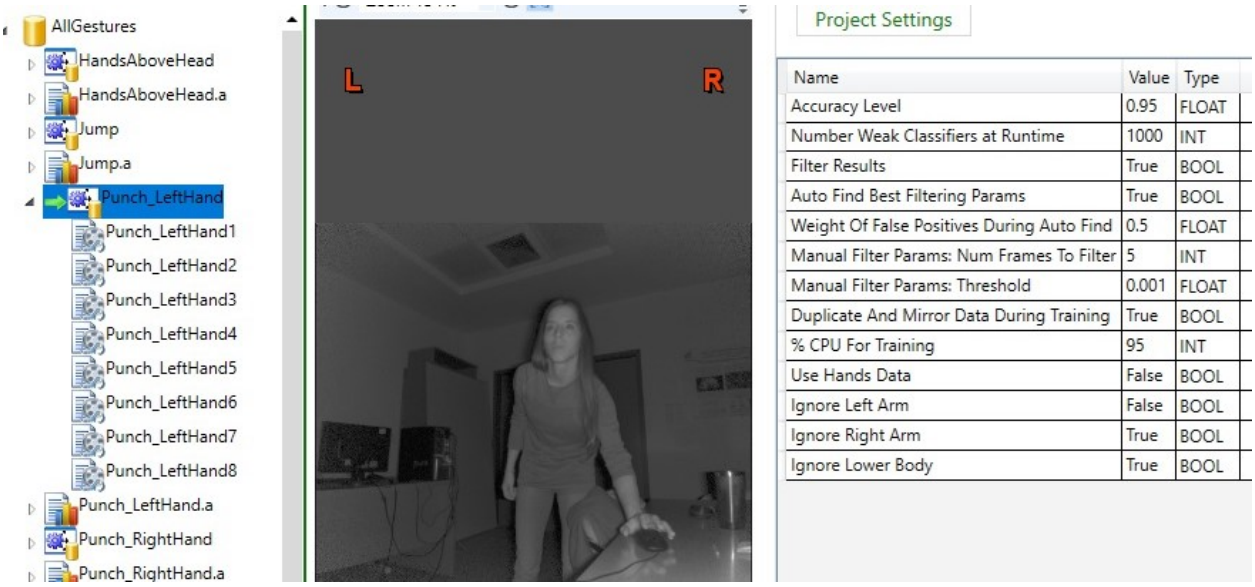
Property Name	**Type**	**Description**
Accuracy Level	FLOAT	A floating point value in the range [0..1], which controls how accurate the results are, but also affects the training time. The higher the accuracy, the longer the training time.
Duplicate And Mirror Data During Training	BOOL	Body data can be duplicated and mirrored in order to have a larger set of training data.
% CPU For Training	INT	A value in the range of [0..100] which indicates the percentage of CPU resources that the trainer should use for training.

Continued on next page

Table 2 – continued from previous page

Property Name	Type	Description
Use Hands Data	BOOL	Default to false. This means by default the hand states are not used for training and detection. For training and detection to use the hand states, set this property to true.
Ignore Left Arm	BOOL	Default to false. This means by default the following left-arm joints are used for training.(Elbow, Wrist, Hand, Tip, Thumb). For training to ignore the left-arm joints, set this property to true. This is useful when we train a right-hand gesture and when the signals from the left hand need to be ignored.(In our case for example for the gesture Swipe- ToRight_RightHand)
Ignore Right Arm	BOOL	Default to false. This means by default the following right-arm joints are used for training.(Elbow, Wrist, Hand, Tip, Thumb).For training to ignore the right-arm joints, set this property to true. This is useful when we train a left-hand gesture and when the signals from the right hand need to be ignored. (In our case for example for the gesture SwipeToLeft_LeftHand)
Ignore Lower Body	BOOL	Default to false. This means by default the following lower-body joints are used for training.(Knees, Ankles, Feet). For training to ignore the lower-body joints, set this property to true. This is useful when you train a gesture that uses upper body only and when you want a gesture to be applicable for both seated and standing positions.(In our case for example for the gesture HandsAboveHead, SwipeUp, SwipeDown etc.)

In the following figure:96 we can see an example of the Project Setting for the gesture "Punch_LeftHand":



Name	Value	Type
Accuracy Level	0.95	FLOAT
Number Weak Classifiers at Runtime	1000	INT
Filter Results	True	BOOL
Auto Find Best Filtering Params	True	BOOL
Weight Of False Positives During Auto Find	0.5	FLOAT
Manual Filter Params: Num Frames To Filter	5	INT
Manual Filter Params: Threshold	0.001	FLOAT
Duplicate And Mirror Data During Training	True	BOOL
% CPU For Training	95	INT
Use Hands Data	False	BOOL
Ignore Left Arm	False	BOOL
Ignore Right Arm	True	BOOL
Ignore Lower Body	True	BOOL

Figure 96: Example of Inputs in Project Settings in VGB.

5.3.8 Gesture Analysis – Gesture Database Analysis

Before building our database, we can analyze each gesture, with the clips intended for the analysis process. When we analyze the clips, the Detector generates automatic tags on the timeline of the clips, where it "thinks" the gestures has been performed. This way we can test how well our Detector has been trained. After inspecting the tags, we can decide if we would like to keep the automatic tags, or move the clips to training, in the case the tags have been inaccurate.

During the building process, the appropriate machine learning algorithms processed the tagged data. After the gesture database was built, it was imported into Unity's project's folder called Streaming Assets: "Assets/Streaming Assets/AllGestures.gbd". The Figure97 below represents the data that was used to build and analyze the discreet RaiseHand_Right gesture. In this case, the handstate is ignored, as well as the left arm and only the upper parts of the body are taken into account, all the joints that belong to the lower part are excluded.

Figure 97: Building the Gesture Database with VGB.

While training and building the database, the system takes into account all the positive frames that have been tagged as positive examples of the gesture and also the negative ones that represent the frames that

do not belong to it as shown in Figure below. In our case, we did not mark any negatives, rather than we let all the other frames be considered as false. Tagging negative frames is more useful in specific cases: for example when we have the gesture "Seated", when we do not want a person seating with crossed legs to be recognized as a "Seated" gesture, then we mark those frames as negatives.(See Figure:98)

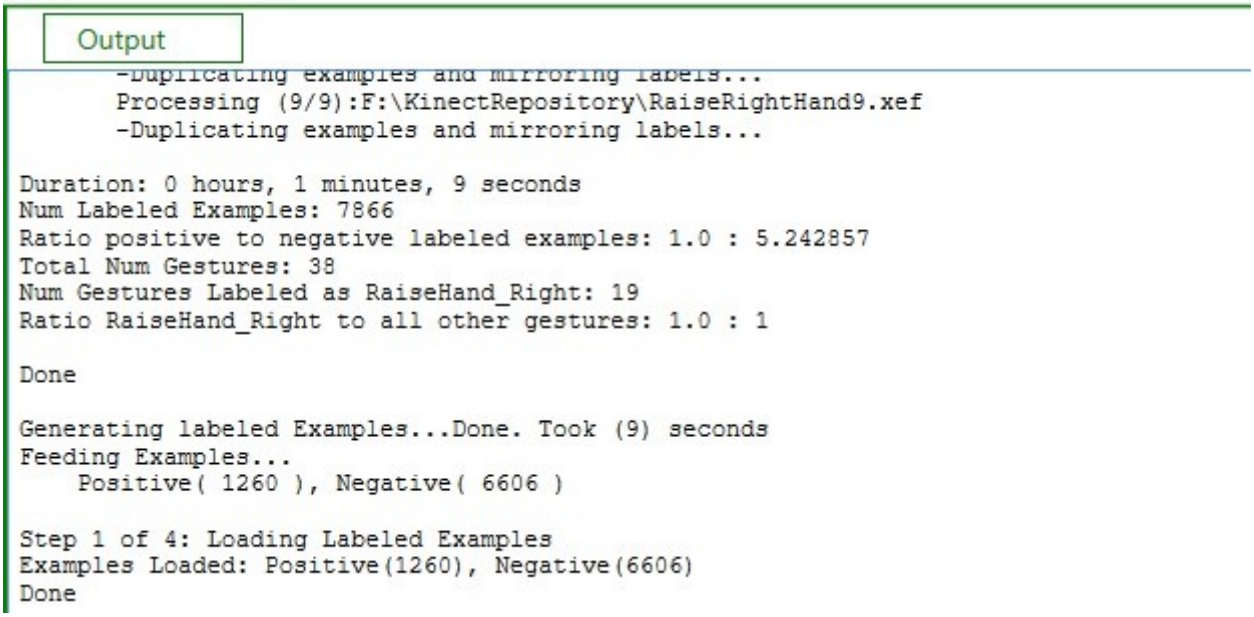


Figure 98: Data generated by the Building Process of Gesture Database in VGB.

After the examples have been fed, the training begins and a pool of weak classifiers is generated. The Figure:99 represents the weak classifiers, used to characterize and distinguish the discreet "RaiseHand_Right" gesture. In this case, 38 different features are taken into account while looking for a repetitive pattern that was located in all the positive examples of the gesture. Each feature had attached a certain number of weak classifiers depending on how important they were in the description of the gesture.

Figure 99: Building Process of Gesture Database in VGB: Generating Weak Classifiers.

Once the weak classifiers are generated depending on the importance weight of each feature, a strong

classifier is built by combining the multiple weak ones with a higher accuracy when taking decisions. For doing so, the Adaptive boosting algorithm, was applied.(See Figure:100)

```
Step 3 of 4: Training Strong Classifier
       -Evaluating classifier data for each example skeleton frame...
Done
       -Running AdaBoost using 8 (of 8 available) hardware threads...
Done
       Num weak classifiers: 1000
       Duration: 0 minutes, 1 seconds
```

Figure 100: Building Process of Gesture Database in VGB: Training String Classifier.

Afterwards, (See Figure:101) a filter is applied, to analyze a short range of frames at once instead of analyzing the whole clip. Additionally, a detection threshold is determined. Lower values of this threshold could increase true positives, at the risk of increasing false positives. Instead, higher values could decrease false positives, at the risk of decreasing true positives.

```
Step 4 of 4: Optimizing detection parameters
       -Generate matrix of test results using different detection parameters...
       Filtering 5 frames using detection threshold 0.049000
       Duration: 0 minutes, 0 seconds
Done
```

Figure 101: Building Process of Gesture Database in VGB: Optimizing Detection Parameters.

Finally, (See Figure:102) we can observe in which level each weak classifier has intervened in the final classification of the "RaiseHand_Right" gesture. This information includes details of features considered by the algorithm including a top ten features list. The features are very enlightening and give an insight into each particular gesture. In each row we can notice the "fValue", which is a ratio of two variables, where one is the variability between groups and the second is the variability within each group. This fValue reveals the discriminative power of each feature independently from others. The "alpha" value is the weight applied to each classifier as determined by the adaptive boost algorithm. Thus, the final output is just a linear combination of all of the weak classifiers. The classifiers were trained one at a time. After each classifier had been trained, the probabilities of each of the training examples appearing in the training set for the next classifier have been updated. The first classifier was trained with equal probability given to all training examples. After it has been trained, the output weight (alpha) for that classifier is computed.

```
Top 10 contributing weak classifiers:
       DiffPositionY( WristRight, SpineShoulder ) using inferred joints, fValue >= 0.000000, alpha = 1.583493
       DiffPositionZ( WristRight, SpineBase ) rejecting inferred joints, fValue >= 0.000000, alpha = 0.789892
       DiffMuscleForceY( HipLeft, HipRight ) rejecting inferred joints, fValue >= -1.000000, alpha = 0.691714
       Angles( WristRight, ElbowRight, ShoulderRight ) using inferred joints, fValue < 122.000000, alpha = 0.650298
       DiffPositionY( WristRight, SpineMid ) using inferred joints, fValue >= 0.000000, alpha = 0.520530
       Angles( Head, ShoulderRight, HipRight ) rejecting inferred joints, fValue >= 140.000000, alpha = 0.478763
       MuscleTorqueZ( ShoulderLeft ) rejecting inferred joints, fValue >= 1.899998, alpha = 0.465958
       MuscleTorqueX( ShoulderLeft ) rejecting inferred joints, fValue < -0.100003, alpha = 0.465259
       Angles( SpineMid, Head, HandTipRight ) rejecting inferred joints, fValue >= 48.000000, alpha = 0.416347
       DiffPositionX( HandRight, WristRight ) using inferred joints, fValue < 0.000000, alpha = 0.407041

Done

Training...Done. Took (22) seconds
Saving Gesture...
Saving Gesture...Done. Took (0) seconds
SUCCESS: The build completed successfully and has been saved to F:\KinectRepository\VGB\RaiseHand_Right picTest.gba.
```

Figure 102: Building Process of Gesture Database in VGB: TOP 10 Classifiers Features.

5.3.9 Testing – Live Preview

When creating a gesture prototype, a small set of training clips (between 10 and 20) were sufficient. In order to know, though, when enough example data have been established, we needed to determine the amount of false positives and false negatives when testing the database.(See Figure:103) An example of a false positive is when a user performs gestureB, but gesture A is detected. Instead, a false negative is when a user performs gestureA, but detection fails to identify that gesture A has been performed. The number of these two errors are two different values that needed to be interpreted separately since the one is not inverse of the other. The results of analyzing the database provided the count of errors from false positives and false negatives as shown in Figure below.

```
Testing on Training Data
    Raw Per Frame Results:
       % Accuracy True Positives: 100.000000 % (1260/1260)
       % Error False Positives: 0.000000 % (0/6602)
    Filtered Per Gesture Results:
       % Accuracy True Positives: 100.000000 % (19/19)
       % Error False Positives: 0.000000 % (0/19)
```

Figure 103: Testing Gesture Database in VGB.

In addition to testing our Gesture Database results, Kinect VGB features one more tool we can utilize: the Live Preview. From the File Options in VGB, we can open the LivePreview, load an analysis file (.gba) of a gesture and check the confidence of its performance in a graphic diagram. In the Figure:104 below, we present the Live preview of the gesture “HandsAboveHead”.

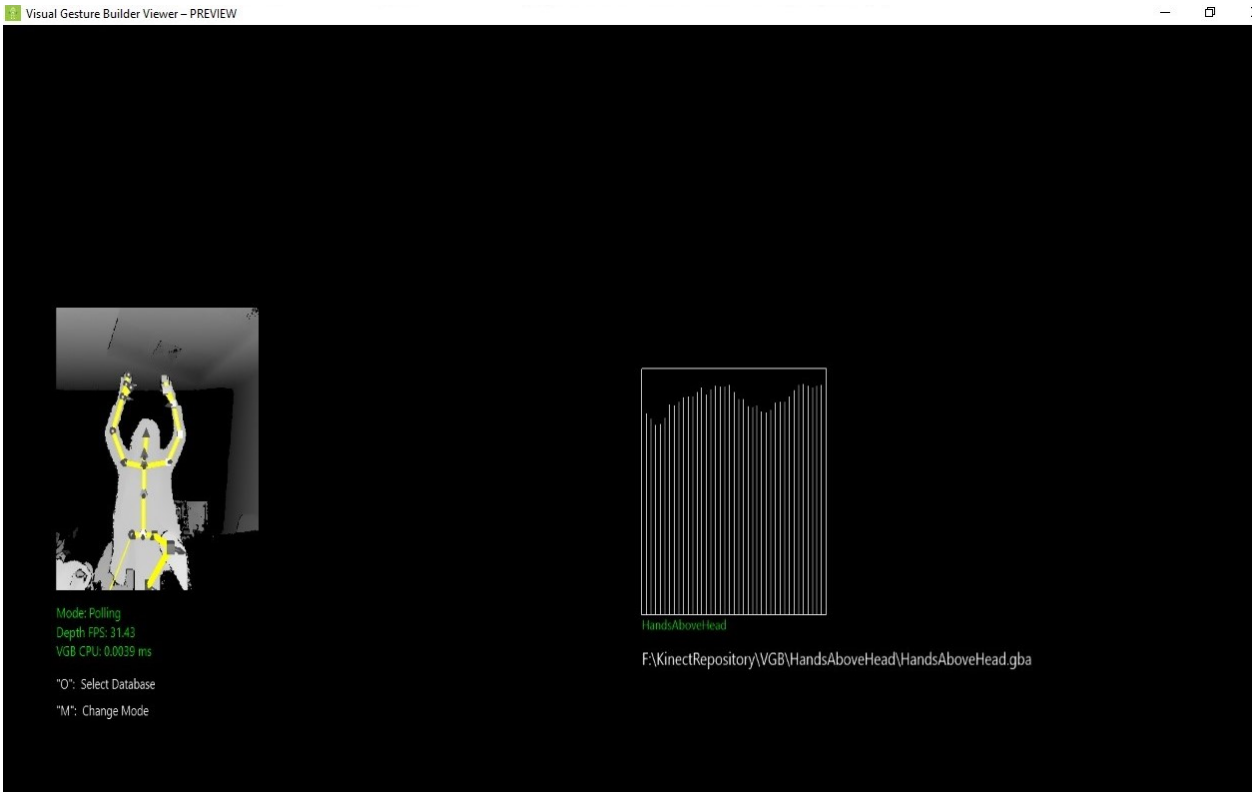


Figure 104: Testing Gesture Database with Live Preview.

5.4 Connecting Kinect with Unity

5.4.1 Introduction – Kinect System Overview

In the present work we implemented an interaction framework, where the user is able to interact with visual effects, projected on arbitrary surface by performing the corresponding gestures in front of a Kinect sensor. For the sensor the accordingly SDK was used, to train and recognize gestures with machine learning. This section covers an overview of the software needed to build up the Kinect system and how we implemented it in Unity.

The entire system is divided into three parts, with the main part being in Unity. (See Figure:105 below) The real space where the user, performs the gestures and receives visual feedback accompanied by a soundtrack. Each time he or she has performed correctly a gesture, the system detects it and triggers a visual. The Kinect sensor provides depth, infrared, color and skeleton data to the graphic engine.

In order to link the used hardware and software, we need to implement the appropriate packages in our main tool, as well as in our Operation System too.

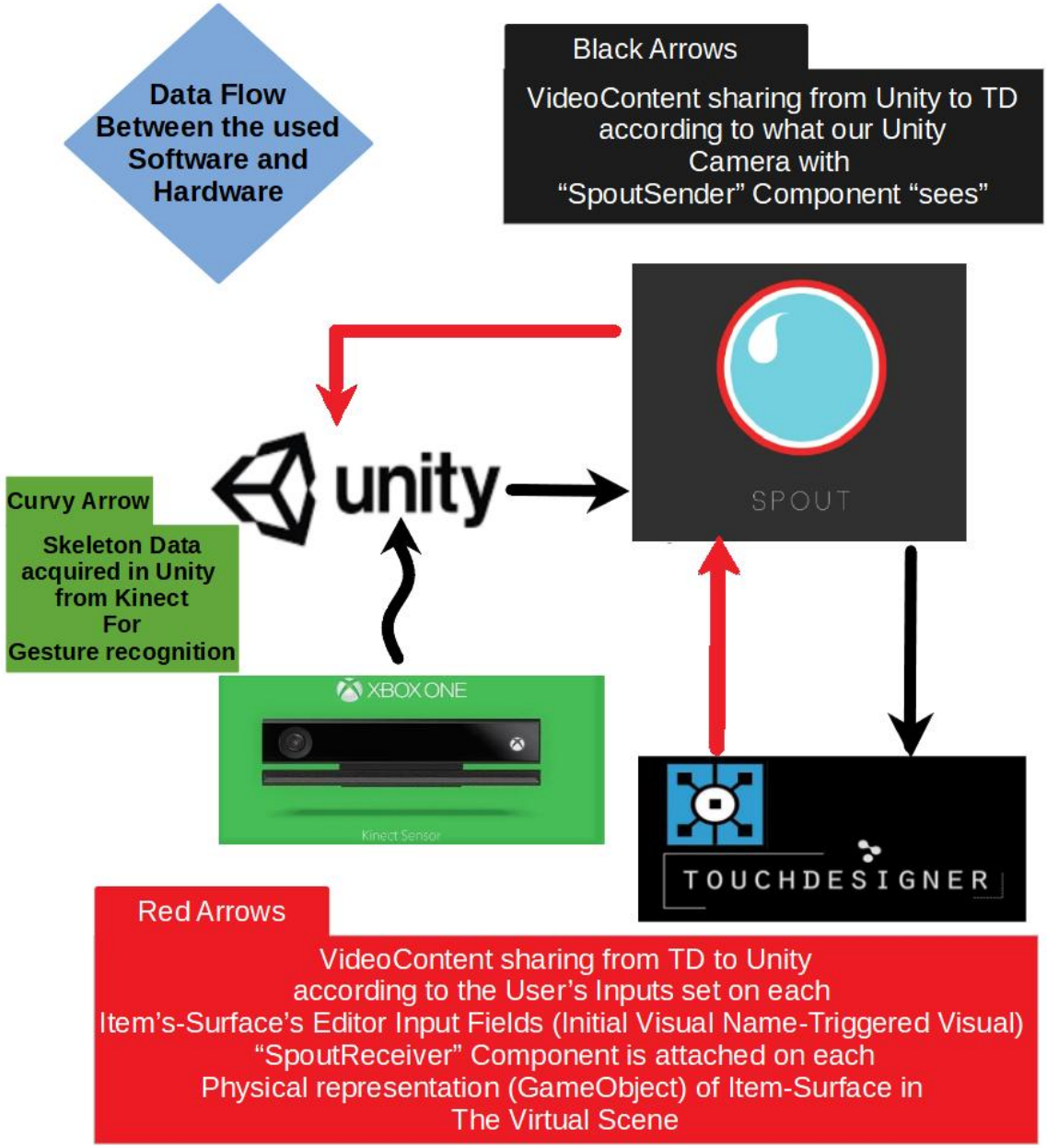


Figure 105: SceneWizard Data Flowchart.

Kinect MS SDK for Windows (we used version 2): The SDK provides the tools and APIs, both native and managed, that we needed to develop our Kinect-enabled application for Microsoft Windows. Developing

Kinect-enabled applications is essentially the same as developing other Windows applications, except that the Kinect SDK provides support for the features of the Kinect, including color images, depth images, audio input, and skeletal data. The SDK includes:

•Drivers and technical documentation for implementing Kinect-enabled applications using a Kinect for Windows sensor.

•Reference APIs and documentation for programming in managed and unmanaged code. The APIs deliver multiple media streams with minimal software latency across various video, CPU, and device variables.

•Samples that demonstrate good practices for using a Kinect sensor.

•Example code that breaks down the samples into user tasks.

The Kinect for Windows SDK 2.0 provides three different API sets that can be used to create Kinect-enabled applications. A set of Windows Runtime APIs is provided to support the development of Windows Store applications. A set of .NET APIs is provided to support the development of WPF applications. And a set of native APIs is provided to support applications that require the performance advantages of native code.

In order to implement Kinect SDK in Unity, we need to import three Unity plugins: the basic one, a face recognition plugin and a gesture builder plugin, each wrapping functionality from Kinect v2 SDK. To specify the process, plug-ins on Windows are DLL files with exported functions. After importing the plugins, two folders are created in our project: “Plugins” Folder and “Standard Assets” Folder. The first, contains the appropriate DLL files with the recognition algorithms and the second contains the appropriate classes to implement the functionality of Kinect.

The Unity package is a custom package and can be imported via Assets → Import Package → Custom Package. The .zip file contains the following Unity packages:

•Kinect.2.0.1410.19000.unitypackage

•Kinect.Face.2.0.1410.19000.unitypackage and

•Kinect.VisualGestureBuilder.2.0.1410.19000.unitypackage .

The first package is necessary for all the general Kinect functions and also includes some tutorials about how to use them. The second package contains the Kinect Face scripts and plugins, which we do not need for our implementation of the prototype. The third one contains the visual gesture builder. When including these packages in Unity, it is important that the "Kinect.2.0.1410.19000.unitypackage" is imported before the VisualGestureBuilder packages can be used. If it is not imported first, the project will possibly not work correctly due to dependencies’ issues. After importing the package, several more scripts are imported including the following:

•Standard Assets/CollectionMap

•Standard Assets/EventPump

•Standard Assets/ExceptionHelper

•Standard Assets/INativeWrapper

•Standard Assets/KinectBuffer

•Standard Assets/NativeObjectCache

•Standard Assets/NativeWrapper

•Standard Assets/SmartGCHandle

•Standard Assets/ThreadSafeDictionary

•Standard Assets/Editor/KinectCopyPluginDataHelper

5.4.2 Integrating Kinect MS SDK in Unity

After our Gesture Database has been successfully built, the next step is to implement the Gesture Detection in Unity. Besides the imported Kinect packages we will need to generate new scripts, in order to import and read appropriately the custom Gesture Database, as well as successfully detecting our gestures. The first thing to do was to get the Kinect sensor data into our scene. To do this we created a new C# script in the project and copied the code from the KinectView sample in the file BodySourceManager.cs. The code retrieves the sensor and opens a reader on it for the body data and also reads the current frame's data in the scripts update() method. The update() method is called as part of the game loop so we are 'pulling' the Kinect skeleton data each time continuously. The class also has a method GetData() to allow the data to be accessed by other scripts. We created an empty GameObject called BodySourceManager and associated the BodySourceManager.cs script with it so the code to set up the Kinect would be executed.(See Figure:106 below)

Figure 106: Unity GameObjects Hierarchy: BodySourceManager Empty GameObject

The code of this class was modified, in order to fulfill our expectations for gesture recognition, as well as the tool's functionality (when resetting for example). When the project enters the "Play Mode", an instance of this script gets initialized, as we can see in the project's console(See Figure:107):

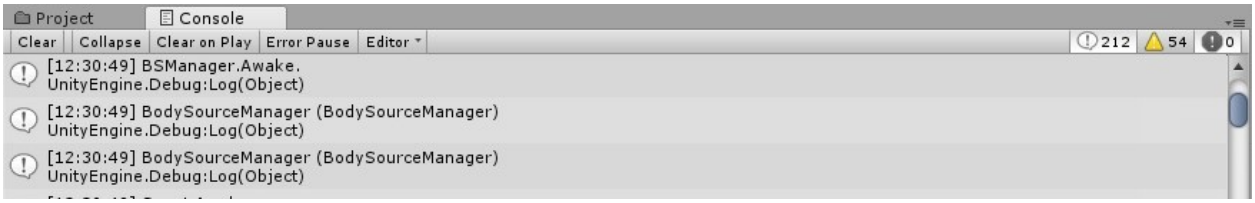


Figure 107: Unity Project's Console: BodySourceManager Initialization

Initially, the "Start()" Function gets executed, which initializes the Sensor and opens the Frame Reader by creating a Frame Reader for the Body Frame Source, accessing the SDK's "BodyFrameSource" Class(represents a source of body frames from a KinectSensor), as well as opens the Sensor accessing KinectSensor Class(represents Kinect Sensor Device). Later in the "Update()" Method, we acquire the skeleton data continuously by capturing the frames from the "BodyFrameReader" Class(represents a Reader for Body frames). The Body Data are also continuously getting refreshed. For each Body the Sensor is capturing we set a "trackingID" and initialize the detection listener for the valid IDs.(See Figure:108)

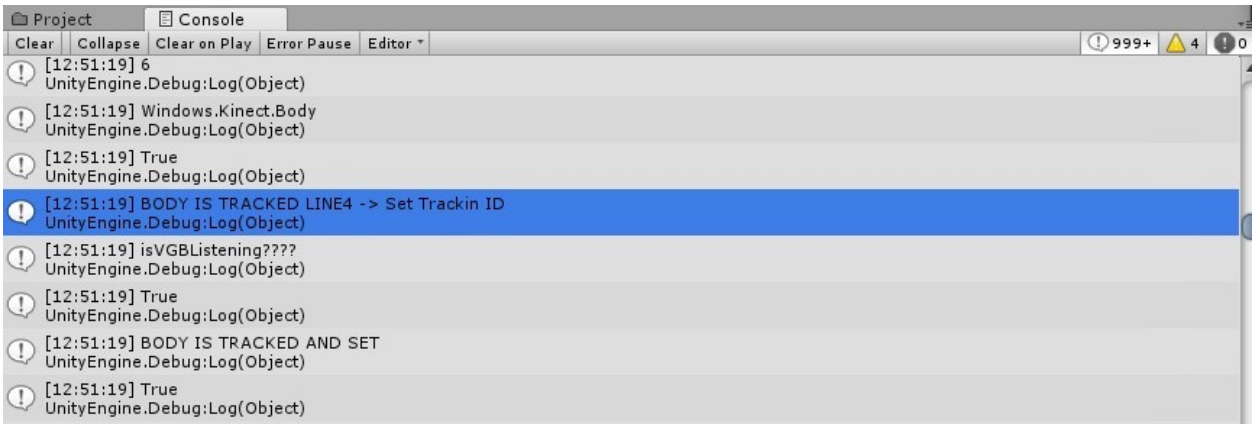


Figure 108: Unity Project's Console: Tracked Body

In the body of the "Update()" Function, we also check if the user is switching the scene version, by pressing the "Space" key on the keyboard, to pause the listener. Finally, when we quit the application, we dispose the Frames, the Reader and close the Sensor, as well as unload the script's instance from the memory.

The next step was to create the "CustomGestureManager" Script, via which the gesture database is loaded and the gestures are detected. The incorporated code in this script works as a detector for the trained gestures. This script was also attached on an empty GameObject, with the same name.

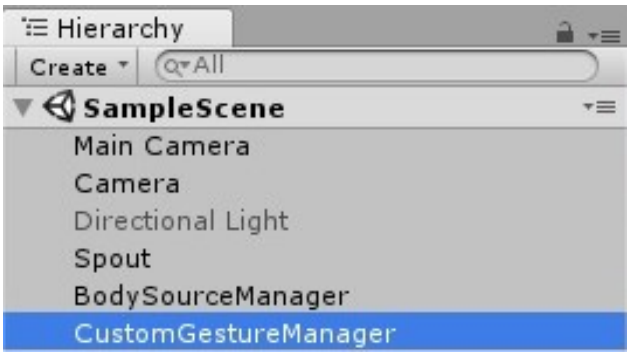


Figure 109: Unity GameObjects Hierarchy: BodySourceManager Empty GameObject

The code in the body of this script is using Events and registers to these to acquire the skeleton data required for the gesture recognition. This script is created in such way, so that the gesture database can be loaded after reading strings through a .txt file, which contains the name of the .gbd file of our gesture database. We did this, so that, if a user desires to upload his own custom database, he/she can just switch the name in the .txt file (contained in the Assets/Resources/ folder), import his/her own .gbd file in the "Streaming Assets" folder and perform his/her own custom gestures. It also helps in the future expansion of our database, by extending the database if adding more gestures. When the script gets compiled the database gets initialized by importing the appropriate .gbd file and the code registers to receive events when new data arrives. In order to identify the gestures when the data arrives, we hold references to our Gesture objects, by adding the in a Gesture List. The references we hold match the strings of gestures that have been set through the Editor, from user inputs.
The code snippet below demonstrates the process we described:

Listing 49: Gesture Database Initialization

```
1
2  _instance.gestureDBNames[i] = VisualGestureBuilderDatabase.Create(Application.
       streamingAssetsPath + "/" + lines[i] + ".gbd");
3      Debug.Log(_instance.gestureDBNames[i]);
4      Debug.Log(VisualGestureBuilderDatabase.Create(Application.streamingAssetsPath + "/"
           + lines[i] + ".gbd"));
5      bool bAllGestures = (gestureNames.Count == 0);
6      Debug.Log("gestureNames.Count");
7      Debug.Log(gestureNames.Count);
8      _gestureFrameSource = VisualGestureBuilderFrameSource.Create(_kinect, 0);
9      foreach (Gesture gesture in _instance.gestureDBNames[i].AvailableGestures)
10     {
11         Debug.Log("LOADING DATABASE GESTURE");
12         bool bAddGesture = bAllGestures || gestureNames.Contains(gesture.Name);
13
14         if (bAddGesture)
15         {
16             string sGestureName = gesture.Name;
17             _gestureFrameSource.AddGesture(gesture);
18             theGesture = gesture;
19             Debug.Log(_instance);
20             //_instance.Gestures[i] = gesture;
21             gestureDB.Add(gesture);
22             Debug.Log(gesture.Name);
23
24             //Debug.Log(theGesture.Name);
```

```
25
26              if (!gestureNames.Contains(sGestureName))
27              {
28                  _instance.gestureNames.Add(sGestureName);
29              }
30          }
31      }
32  }
33  // open the reader
34  _instance._gestureFrameReader = _gestureFrameSource != null ? _instance.
        _gestureFrameSource.OpenReader() : null;
35  if (_gestureFrameReader != null)
36  {
37      _instance._gestureFrameReader.IsPaused = true;
38      Debug.Log("GestureFrameReader is PAUSED");
39  }
40  isDatabaseInitialized = true;
41  Resources.UnloadUnusedAssets();
42  Debug.Log("GestureDatabase is initialized");
43  return true;
44  }
```

Since we need a trackingID of the tracked body the sensor identified from skeletal data for gesture detection, we pause the frame reader until we have a valid trackingID (this will be retrieved from the skeleton data in "BodySourceManager" script, see the code snippet below). If the tracked body is valid, we check if the listener is open, we subscribe to the "gestureFrameReader.FrameArrived" Event, to monitor the occurrence of a significant gesture, otherwise we we initialize the listener.

Listing 50: Setting a Tracking ID. Initialize Gesture Listener.

```
1  if (body == null)
2  {
3      _trackingId = 0;
4      continue;
5  }
6  if (body != null && body.IsTracked)
7  {
8      Debug.Log("BODY IS TRACKED LINE4 -> Set Trackin ID");
9      _trackingId = body.TrackingId;
10     if (isVGBListening == false)
11     {
12         Debug.Log("isVGBListening????");
13         isVGBListening = CustomGestureManager.Instance.IsVisualGestureInitialized();
14         Debug.Log(isVGBListening);
15         if (isVGBListening)
16         {
17             GestureManager.SetTrackingId(body.TrackingId);
18             Debug.Log("BODY IS TRACKED AND SET");
19         }
20     }
21     break;
22 }
```

Our code follows the familiar pattern followed by the Kinect SDK of creating a frame source and opening a reader on that source, which can be used to register for data events. Specifically, once the sensor detects a valid body, it subscribes to the Event “FrameArrived” and “listens” if a gesture is being detected. The code snippet below, demonstrates this subscription to Kinect’s Event.

Listing 51: Set Tracking ID, Subscribe to Gesture Event. Initialize Gesture Listener.

```
public void SetTrackingId(ulong id)
    {
        _gestureFrameReader.IsPaused = false;
        _gestureFrameSource.TrackingId = id;
        _gestureFrameReader.FrameArrived += _gestureFrameReader_FrameArrived;
        primaryUserID = id;
    }
```

When the gesture data arrives, we identify the performed gesture and trigger the appropriate visuals, utilizing Methods provided by Kinect SDK in a desired manner. This part of the implementation we previously discussed in sub-section 4.2.2 and page 121, Listing:41. On the code snippet below we can see the gesture identification:

Listing 52: Gesture Event.

```
void _gestureFrameReader_FrameArrived(object sender,
    VisualGestureBuilderFrameArrivedEventArgs e)
    {
        Debug.Log("DETECTION ACCESSED");
        sizeList = invs[p].inventory.Length;
        Debug.Log(invs[p].name);
        VisualGestureBuilderFrameReference frameReference = e.FrameReference;
        using (VisualGestureBuilderFrame frame = frameReference.AcquireFrame())
        {
            if (frame != null && frame.DiscreteGestureResults != null)
            {
                Debug.Log("Frame != Null?");
                Debug.Log(frame != null);
                Debug.Log("frame.DiscreteResults != null?");
                Debug.Log(frame.DiscreteGestureResults != null);
                DiscreteGestureResult result = null;
                Debug.Log(frame.DiscreteGestureResults.Count);
                Debug.Log(frame.DiscreteGestureResults);
                Debug.Log(frame.DiscreteGestureResults.Keys);
                //if (frame.DiscreteGestureResults.Count > 0)
                //{
                Debug.Log("DETECTION OF The GESTURE");
                Debug.Log(frame.DiscreteGestureResults.Count);
                //Debug.Log(frame.DiscreteGestureResults[theGesture]);
                foreach (Gesture gesture in gestureDB)
                {
                    Debug.Log(gesture.Name);
                    Debug.Log(frame.DiscreteGestureResults[gesture].Detected);
                    if (frame.DiscreteGestureResults[gesture].Detected == false)
                    {
                        Debug.Log("frame results.Detected = false");
                        continue;
                    }
                    else if (frame.DiscreteGestureResults[gesture].Detected == true)
                    {
                        Debug.Log("Detected Gesture:");
                        Debug.Log(gesture.Name);
                        result = frame.DiscreteGestureResults[gesture];
```

Finally, the script contains further functionality in the Update() function, which is continuously checking the user's keyboard actions, in order to re-initialize the gesture database, according to the version of the scene. Since each scene version and each surface-item may contain different gestures, we initialize the gesture

database according to the user's choice of gestures from our tool's Editor Window.

5.5 Connecting Unity with a VideoMapping Software

In this section, we will discuss, how we connected Unity with our Video Mapping Software, in specific TouchDesigner, in order to send and receive video frames between these two programs/apps. We used TouchDesigner to generate and create the video content (visuals) and using Spout Protocol, we can obtain these visuals in Unity, as textures. Normally, in Unity the Mesh geometry of an object only gives a rough approximation of the shape, while most of the fine detail is supplied by Textures. A Texture is just a standard bitmap image that is applied over the Mesh surface. We can think of a texture image as though it were printed on a rubber sheet that is stretched and pinned onto the Mesh at appropriate positions. Textures are applied to objects using Materials. Materials use specialized graphics programs called Shaders to render a texture on the Mesh surface. Shaders can implement lighting and colouring effects to simulate shiny or bumpy surfaces among many other things. They can also make use of two or more textures at a time, combining them for even greater flexibility.

Spout is a video sharing system for Windows that allows applications to share frames in real time without incurring significant performance overhead. It's supported by several applications (MadMapper, Resolume, TouchDesigner etc.) and frameworks (Processing, openFrameworks, etc.). It works in a similar way to Syphon for Mac, and it's similarly useful for Projection Mapping and VJing. Apart from Spout Protocol (Syphon for MAC), there is also Network Device Interface(NDI), a high performance standard, that allows anyone to use rel-time, ultra low-latency video on existing IP video networks. Keijiro Takahashi has implemented KlakNDI, a Unity plugin that allows sharing video frames between computers using NDI. According to him, it depends on if someone is going to use a single computer, he should use Spout. Otherwise, if connecting multiple computersis needed, NDI is more efficient. Spout is a superior solution for local interoperation. It's faster, low latency, more memory efficient and better quality. It's recommended using Spout unless multiple computers are involved. As a result, we downloaded Spout protocol for Windows, as well as Spout unitypackage.

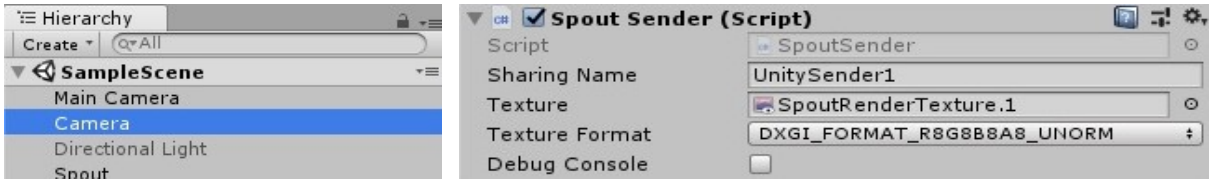
5.5.1 Integrating Spout Protocol in Unity

To integrate Spout Protocol in Unity, we needed to download the appropriate package to import in our project. We used Benjamin Kuperberg's, original plugin from his Github repository. The significant folder of the plugin that contains the necessary scripts, which initialize the video sharing option, is "UnitySpoutDemo". More specifically, in the folder path "Assets/UnitySpoutDemo/Assets/Spout/Scripts" we find the three main scripts we will need for our project: "Spout", "SpoutSender" and "SpoutReceiver". To access the scripts' methods, we needed to perform some modifications on their code, such as casting the methods as public, which thankfully did not break any behaviors or caused any sort of corruption.

The first thing to do, after importing the package is to follow its manual instructions. We need a Spout component in our scene, so we created an empty GameObject with the name "Spout" and attached on it the script with the same name.(See Figure:110 below)

Figure 110: Unity GameObjects Hierarchy: Spout Empty GameObject

Afterwards, we need to set up a camera that will be "watching" our scene and streaming the frames to TouchDesigner, in order to perform the Projection Mapping, without the annoying GameView task-lines of Unity, which appear in the layout of the image we want to project. To setup a camera in our Unity scene, we create a Camera GameObject in our project's hierarchy and attach the "SpoutSender" component script on it. In the component's parameters, we set the sharing name this Sender texture will have. (Has to be unique on our system so Spout can provide this texture to client in TouchDesigner under this name) In the "Texture" parameter we have to select the RenderTexture that our Camera uses for rendering as well as a "TextureFormat". Depending on Unity's current DirectX mode and the graphics card of the system the appropriate option is needed in the Texture Format of the Spout Senders. In our project, we selected one of the default Textures contained in Spout's Demo, "Spout.Render.Texture.1" and DXGI_FORMAT_R8G8B8A8_UNORM Texture format.



(a) Unity GameObject Hierarchy: Camera GameObject

(b) Camera Component

Figure 111: Camera Settings

Finally, we need to attach the script "SpoutReceiver", on each and every Item-Surface, any version of the scene contains, in order to "dress" the object with our selected visual. We have a script that mainly controls the behavior of our scene, called "Slot". This script, whenever an item -surface gets spawned in our scene, it attaches on it the component "SpoutReceiver". This component has a Mesh Renderer and a Material attached. (The Material should not be used by other SpoutReceivers). Additionally there are two more parameters: 'Select sender' parameter, which if the user has only one external Spout texture source can be set to 'Any' option, so the first registered Spout texture is displayed. For more complex set-ups with multiple texture shares the user should explicit set a name of the texture wants to display. (Select sender 'Other(specify)' and enter the name of your Spout texture share name). Depending on the name the user uses in the external SpoutSender app, 'Sender name' property of SpoutReceiver component has to be set appropriately. In our case we implemented some code, which we will discuss in detail in the next Section,

which adjusts the sender's name parameter, according to the user's selection. These selection the user can perform in SceneWizard's Editor Window. The code snippet below, demonstrates how the crucial script is attached, for every Item-Surface, on the current version of a scene.

```
}
//MyInventory.inventory[i].item.physicalRepresentation.AddComponent<MeshRenderer>().material = materi
myMaterial = MaterialFactory.MaterialCreator();
MyInventory.inventory[i].item.physicalRepresentation.AddComponent<MeshRenderer>().material = myMateri
MyInventory.inventory[i].item.physicalRepresentation.AddComponent<SpoutReceiver>();
```

Figure 112: Attaching Spout Component via Script

5.5.2 Integrating Spout Protocol in TouchDesigner

As we have already mentioned, we need to share video frames between our utilized software, Unity and TouchDesigner. In order to do that, we need to initialize Spout Protocol in Touch Designer, additionally to Unity. Luckily, TouchDesigner application already includes Spout Protocol support. TouchDesigner provides Spout In and Out TOPS that work in much the same way as the DirectX in and Out TOPS or the Movie In TOP. The Spout In TOP is a texture source. Operation is simple and will be clear to anybody familiar with TouchDesigner. To load a Spout In TOP, we can right click on the main window and select "Add Operator", then from the Operator selection dialog, the "TOP" tab. We can click once on "Spout In" and move the selection rectangle to the main window and click again to place it. Initially, we might see a warning that could relate to a memory leak bug in older NVIDIA drivers. Spout In is a receiver and requires a Spout sender. Adding a Spout Out TOP works in the same way. Spout Out is a sender, so connect a texture source to Spout Out. The sender is named "TouchDesigner Spout" by default. We can change it as required by entering another name in the entry field. This name will represent the name of the visual that this specific Spout Sender will share to Unity and will represent a unique visual name that a user can use in the SceneWizard's Editor Window. On the fisures below, we can see an example of s Spout In and Spout Out Tops in TouchDesigner, which represent the corresponding senders and receivers in TouchDesigner. Note that a Receiver in TouchDesigner represents a Sender from Unity (same video content) and vice-versa.

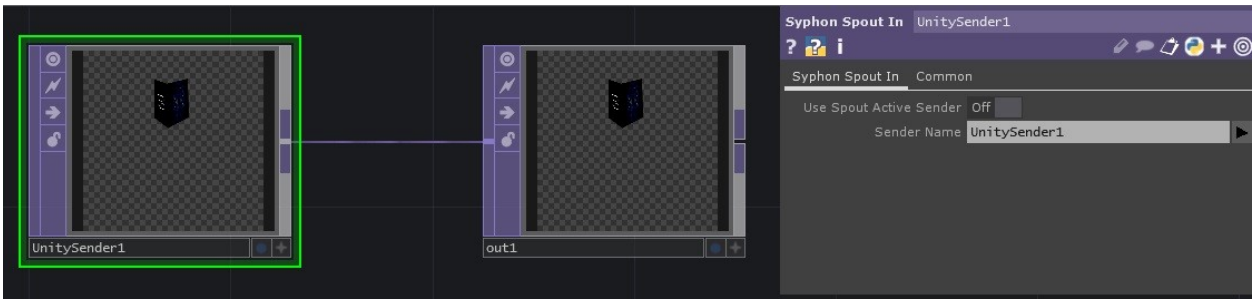


Figure 113: TouchDesigner: Spout Input (Syphon Spout In) Operator

We have created several visuals for demonstrating our tool, SceneWizard and for each and every one we gave unique names, that mostly characterize the visual, so that we can easily remember and type the names into our tool's Editor Window. We can see some example of the created visuals in the subsection 6.4.2 below.

6 User Study - Demo

6.1 Introduction

In this section we will discuss about the user study that has been engaged, in order to assess our built system, as well as the evaluation process of the usability of SceneWizard and its results. The process was divided in two parts:

1. Evaluation of SceneWizard Tool.
2. Evaluation of Gesture Database.

In addition to this, we will describe the process of preparing a Demo for SceneWizard. The Demo was created, in order to show the tool's capabilities as well as, demonstrate how Interactivity can add up an extra dimension in Projection Mapping Performances.

6.2 Evaluation of SceneWizard

To evaluate our tool, we operated the "Think Aloud" evaluation protocol. This is a protocol used to gather data in usability testing in product design and development, in psychology and a range of social sciences (e.g., reading, writing, translation research, decision making, and process tracing). The think-aloud method was introduced in the usability field by Clayton Lewis, while he was at IBM, and is explained in Task-Centered User Interface Design: A Practical Introduction by C. Lewis and J. Rieman. To accomplish this process, we requested to discuss with a number of users from the Architect's School Department of our Technical University in Crete, who are interested in performing Projection Mapping, as well as expanding their work in Interactive Performances. We stipulated these users in utilizing our SceneWizard and produced handwritten notes according to their reactions and demands.

During the evaluation process the users, after given the appropriate instructions, have been granted to prepare a simple Interactive Projection Mapping project, using SceneWizard, without any further assistance. The users who participated in this process were three. All the users had never been in touch with SceneWizard

before, but stumbled upon merest mistakes, mostly due to being unfamiliar with Unity. In addition to this, we developed an online survey questionnaire, under a VJ community of Facebook. More specifically, we posed a question concerning the percentage of VJs that utilize Unity. The question received more than 50 comments and numerous likes, indicating that Unity is being widely used from the broader community of VJs. Moreover, we should take into account that this community group contains around 13.000 members, but not all members could see our posted question, since Facebook works with "ranking points" and broadcasts an individual's posts according to how "engaged" he/she is to this specific group. The supervisor of our tool had been constantly present during the evaluation process, in order to encourage the users to express their thoughts and pose questions concerning the usability of the tool at the end of the trial. The users have been very helpful for the optimization of our tool, since they proposed interesting new suggestions they would aspire to SceneWizard.

6.2.1 Requested Operations for Think Aloud Evaluation

1. Launch Unity3D.
2. Open SceneWizard tool.
3. Create an Item-Surface.
4. Update an Item-Surface.
5. Create a Scene Version.
6. Update Scene Version by populating it with the Item-Surface.
7. Perform.
8. Exit Play Mode.
9. Delete Item-Surface.
10. Delete Scene Version.

The requested operations cover the whole basic functionality of SceneWizard. Furthermore, the evaluation process was performed before the official presentation of SceneWizard, in order to avoid any mistakes during the official performance.

6.2.2 Adjustments to SceneWizard after the Evaluation Process

The adjustments that resulted from the evaluation process are presented below. All the users' observations have been reviewed and SceneWizard was amended accordingly. The problems the users faced:

1. Unprecise alignment between the virtual-environment scene and the real-environment scene.
2. Difficulties in filling up the Text fields and Integer fields of our Editor Window.

The first issue had been a main concern during the development of SceneWizard, as it also constitutes an issue of Projection Mapping, the image registration issue. To resolve this issue we added extra scripts in our implementation code, to allow manual alignment of each item's surface via unity's Scene View. For the second difficulty the users faced, we propose further experimentation on Unity cross-platform engine, so that the users get more keen on utilizing Unity, thus SceneWizard.

6.3 Adversities with Kinect Depth Sensor

The main goal of the user study is to evaluate the effectiveness and accuracy of the present work, concerning our custom Gesture Database. To achieve this, we gathered our fellow colleagues from "MUSIC" lab, to perform specific gestures in front of Kinect Sensor, while performing with SceneWizard, according to the Gesture Database we built for our tool and posed questions orally regarding how comfortable, accurate and easy it was to use our interactive system, as well as which was the error rate and the latencies.

6.3.1 Evaluation of our Gesture Database

In this evaluation process, 5 testers took part. During the experiment each one of them performed the requested gestures separately and individually. Firstly, the users were instructed which gestures could be used and how they needed to perform them. Afterwards, they had to get familiarized with the usage of the system by repeating the sequence of gestures several times. In the midst of the process, we tried adding more users, to monitor Kinect's attitude, as well. The users performed gestures, while viewing our Demo Scene. The gestures they performed correspond to the ones we use in our Demo: "HandsAboveHead", "Punch_LeftHand", "Jump" and "Punch_RightHand".

6.3.2 Results of the Evaluation of our Gesture Database

The use of the Kinect sensor by the volunteers was more flexible than expected. The sensor detected well all the volunteers, no matter their body shape, or clothing, even though the Sensor had never been trained on their figure avatars. However, when the users needed to perform a gesture that is similar, to a gesture corresponding a different Item-Surface in the present scene version, Kinect "fired" both gesture events, even though only one had been performed. As a result, since the Demo Scene contains two items-surfaces in a scene version, both of them switched their visual-texture. Moreover, when adding more than one users, our Sensor locked on the first user and monitored his actions, waiting for a gesture event. However, when we removed the first user, even though Kinect should lock on the next user, as momentarily detected, the detector could not monitor the next user's actions, but instead it monitored the first user when he reappeared on Kinect's field of view.

The user's reviews have been mostly positive, since when they performed the interactions individually the detector worked mostly with success. The majority of them thought that the gestures have been accurate enough and seemingly easy to learn. Nonetheless, when using the sensor repeatedly, few errors occurred, these were due to the fact that the gestures were either incorrectly or not recognized and have been resolved by re-initialized Unity.

As a result, we summarize the adversities we faced, during the evaluation process. A conspicuous difficulty when using Machine Learning Algorithms to train Kinect on gesture detection was the fact that the user had to perform the gestures exactly, or at least very similar, to the ones that were used to train the system. If the performed gestures had not been very alike, the detector had complications on recognizing the gestures and thereby, the interaction had not been occurred. This can be sorted out by making the user spend some time training and getting familiarized with the sequence of gestures until these are well enough performed. Another issue that had to be taken into account was the fact that gestures that use similar body joints or movements confound the sensor. For example when trying to perform a "SwipeDown" gesture, it is possible the sensor detects the "SwipeUp" or "HandsAboveHead" gesture, since to swipe down the user needs to raise his/her hands first. To resolve this issue we need further and more frame-precise training of our detector, or use hand states to indicate the beginning of different gestures, as for instance, train the "SwipeUp" with a closed hand state and "SwipeDown" with an open hand state and so on. Finally, a noticeable weakness of

the detector was that it did not recognize in the same way a certain gesture performed by different people with varied physiology, shape and color. Initially, the issue was resolved by adding more clips on the training sessions, using a variety of concepts as in wearing different clothes, recording in various background and environments, as well as in various positions in the field-of-view of Kinect. Ideally, the problem can be swiped out nearby completely if we add even more clips recorded with several people, with a variety of clothes and different colors. Kinect has its own limitation too. The body joints defined by Kinect are not so precise as if when compared with a marker-based system. KinectV2 detects human presence by recognizing the body posture, but in some cases it might not create a valid skeleton. This was solved by locating the actor sufficiently far from the sensor, for a full skeleton recognition.

6.4 Preparation for SceneWizard – Demo

6.4.1 Introduction

In persuance of presenting our Tool, which was created in and for Unity, we needed to prepare a Demo, which flaunts SceneWizard. To achieve this, we impersonated the user as an artist-designer, that performs in an Interactive Projection Mapping. The Demo manages to serve all of the Use Cases we discussed in chapter 4. We first sit in the Designer's/Artist's position, utilizing SceneWizard, to set up an interactive Projection Mapping performance. Secondly, we assume the role of the End-User, as we sit in front of our Depth Sensor and perfom an extragavanza of gestures in front of our constructed scene. In this section we will discuss, how we prepared our Demo and setting-up our Scene (real world & virtual world) for the performance.

6.4.2 Creating Visuals on TouchDesigner

The artists or designers, while Vjing, create and manipulate imagery in real-time through technological mediation and for an audience, frequently in synchronization to music. This results in live multimedia performance. Often using a video mixer, VJs blend and superimpose various video sources into a live motion composition. In recent years, electronic musical instrument makers have begun to make specialty equipment for VJing. In our case, the controller of visuals will be the user's body movements. There is a variety of tools a VJ can use to create visuals, as we have already mentioned in the state-of-the-art (section 2). We concluded on using TouchDesigner, as an endeavor to create real-time interactive multimedia content.

TouchDesigner isn't an application that is ready on start to perform actions that may seem simple in other applications. Yet, TD is a node-based visual programming language, that needs plenty of exercise and experimentation, to get acclimated to it. We worked on TD, for a few months before we have been able to produce a worthwhile result. With the assistance of the freely available online tutorials from brilliant artists-developers, such as Matthew Ragan and Elburz Sorkhabi, amongst many others, we created several visual effects, from which eight will be presented in our Demo performance. Below we can see some examples of our visuals in TD, as seen from the background of TD:

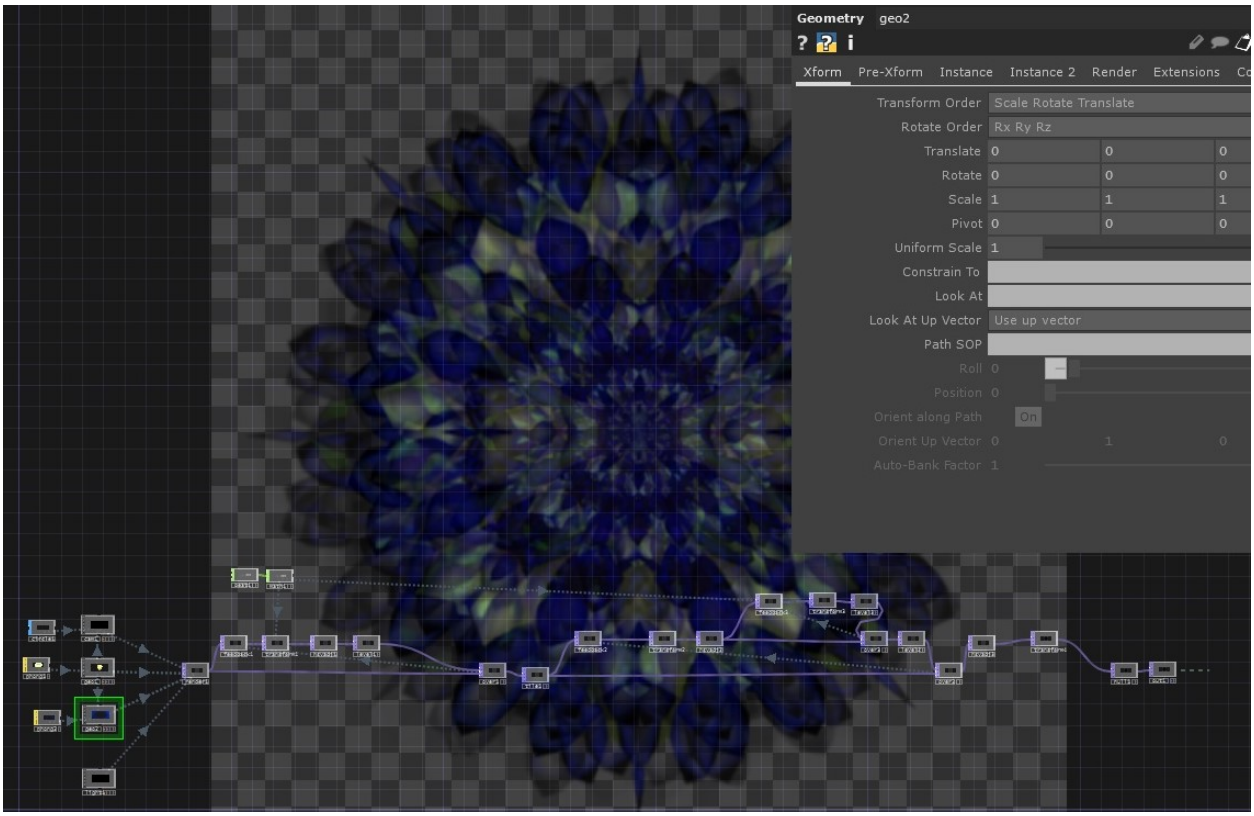


Figure 114: TouchDesigner: Example 1, FractalFlower

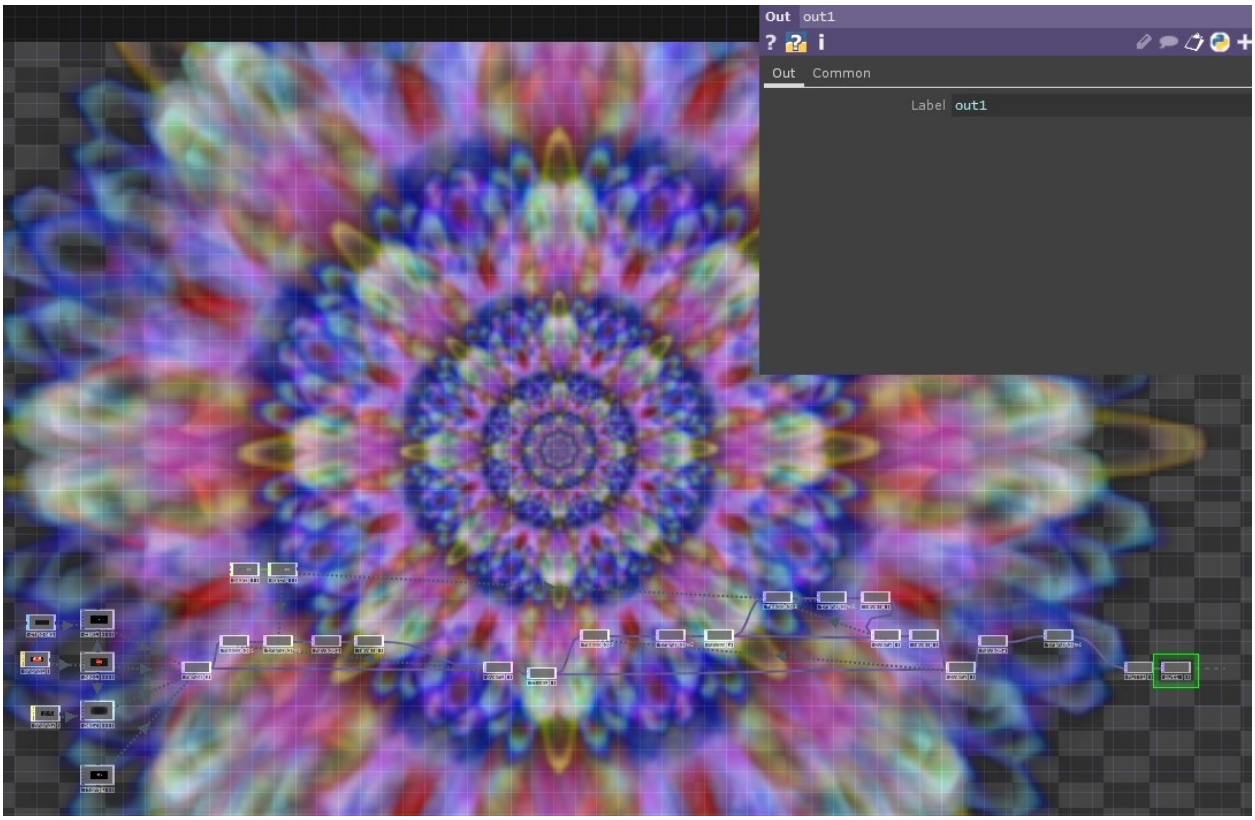


Figure 115: TouchDesigner: Example 2, FractalFlower.

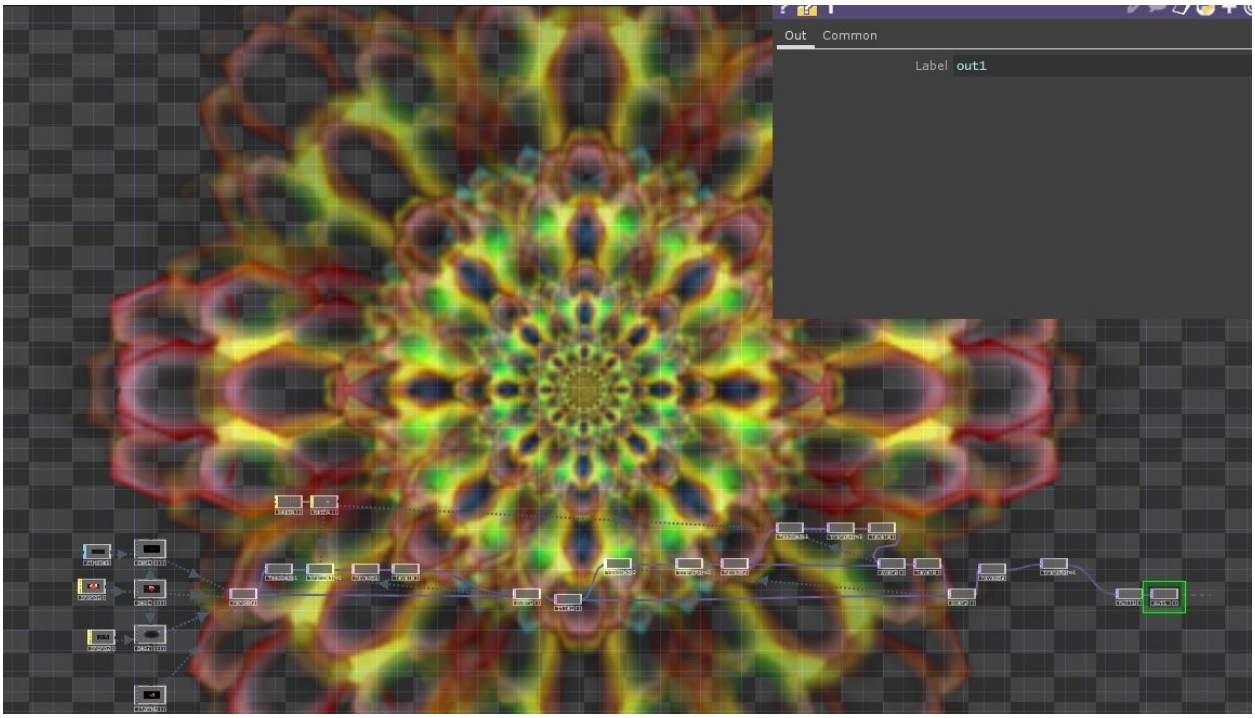


Figure 116: TouchDesigner: Example 3, FractalFlower

Fractal/Flower Example: In this example, we created an animated fractal in a flower shape, using several Operators of TD. In a Geometry COMP, we created a circle and applied noise effect, by setting up the appropriate SOP parameters, as seen in the figure below:

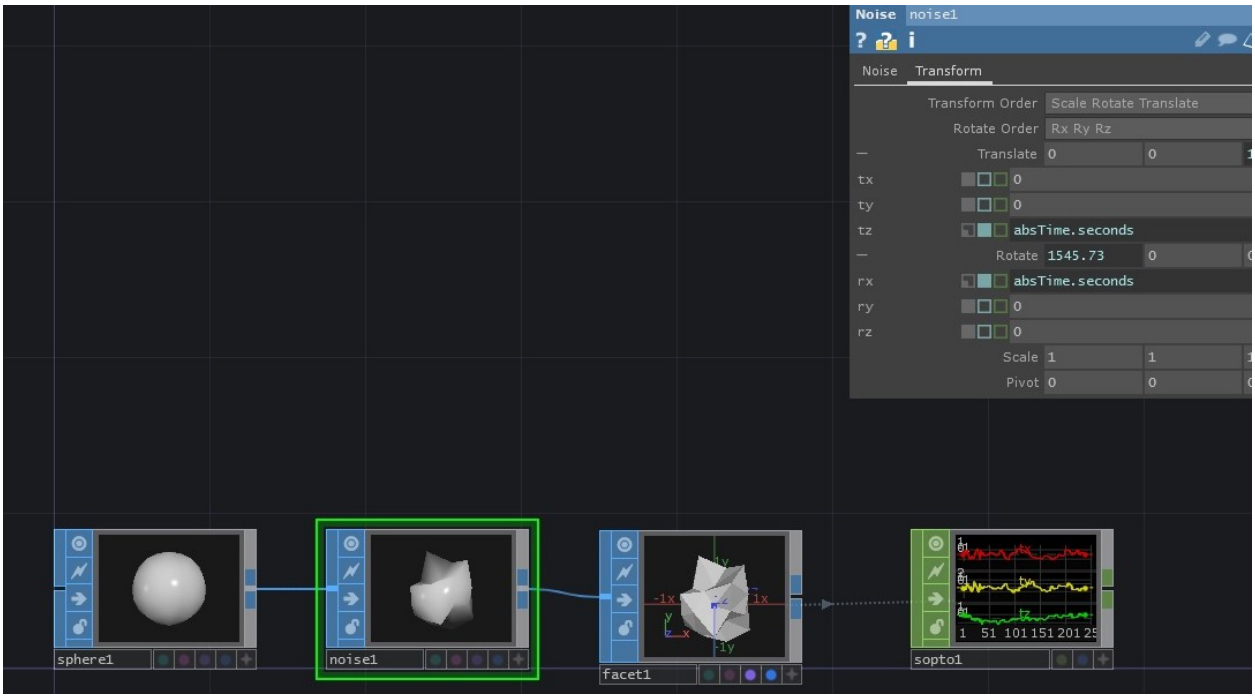


Figure 117: TD Example forming Geometry with SOP operators

Later, using a MAT operator we applied different shading and color texturing on the object, as seen in the figure below:

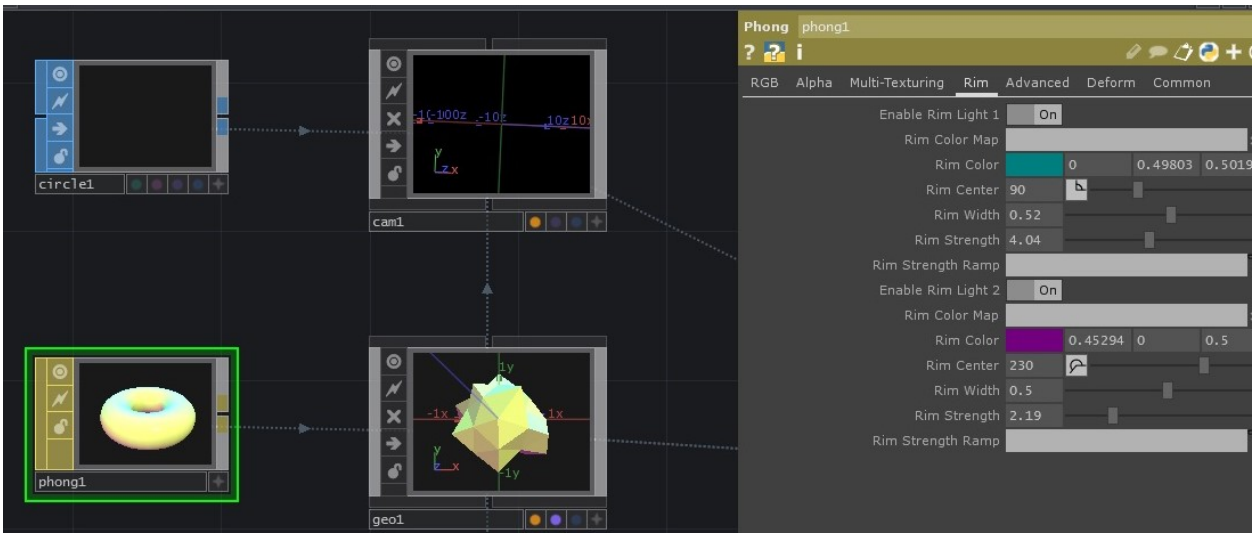


Figure 118: TD Example adding Texture with MAT operators

Then, using a TOP operator, we render what our Camera COMP “sees”, as we see in the figure below:

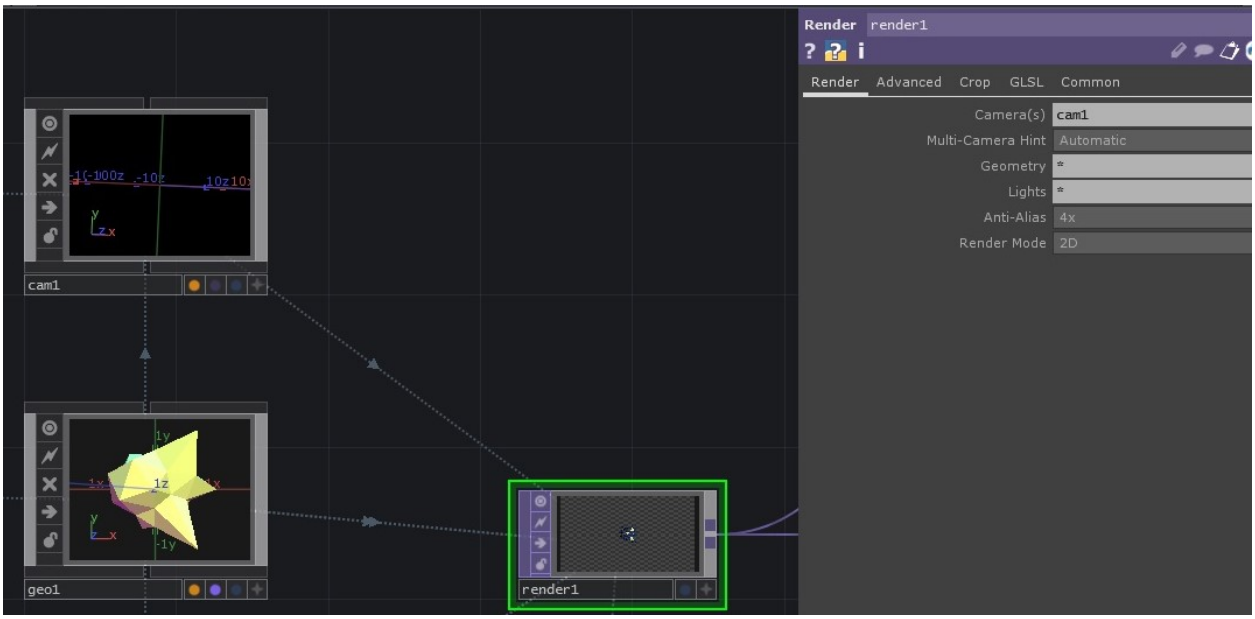


Figure 119: TD Example TOP render operator

Then, using CHOPs and TOPs, we apply several filters that will finally produce an animation according to the desirabe result, as we see below:

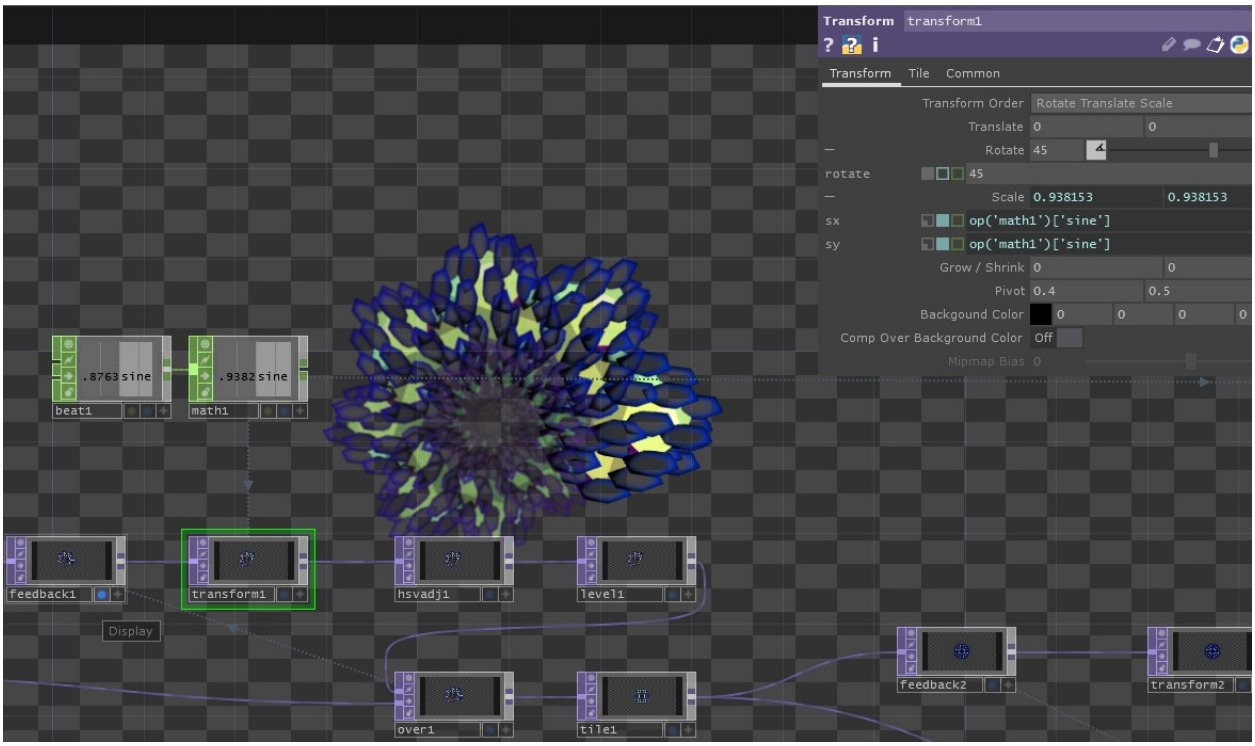


Figure 120: TD Example animating with CHOP operators

In a similar manner we worked, in order to produce all our visual effects, as seen in the example pictures below:

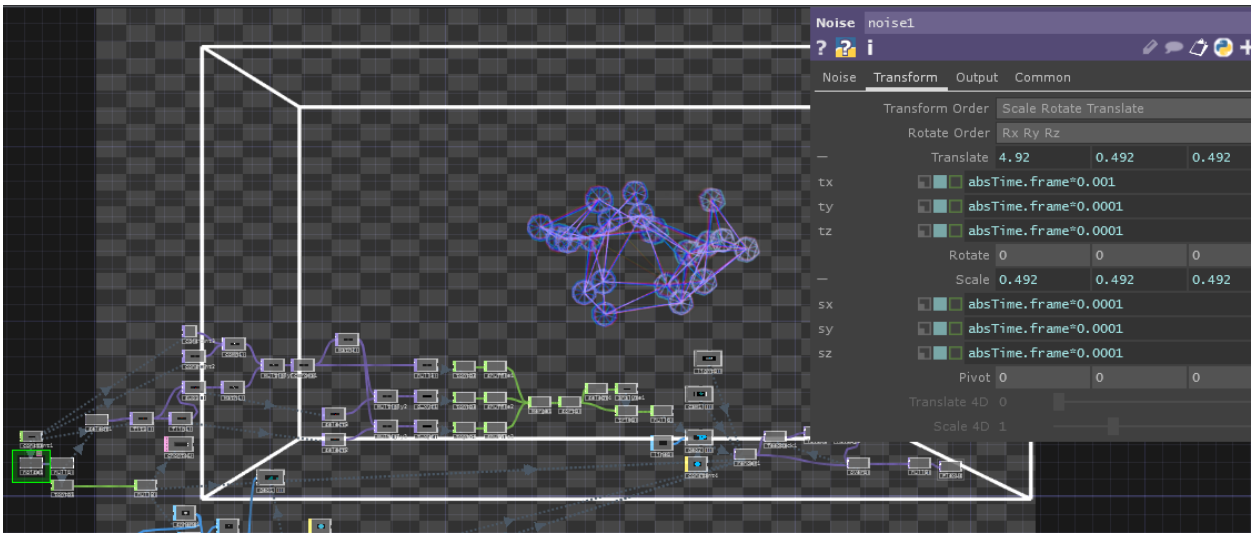


Figure 121: TD Example Plexus of Deforming Spheres

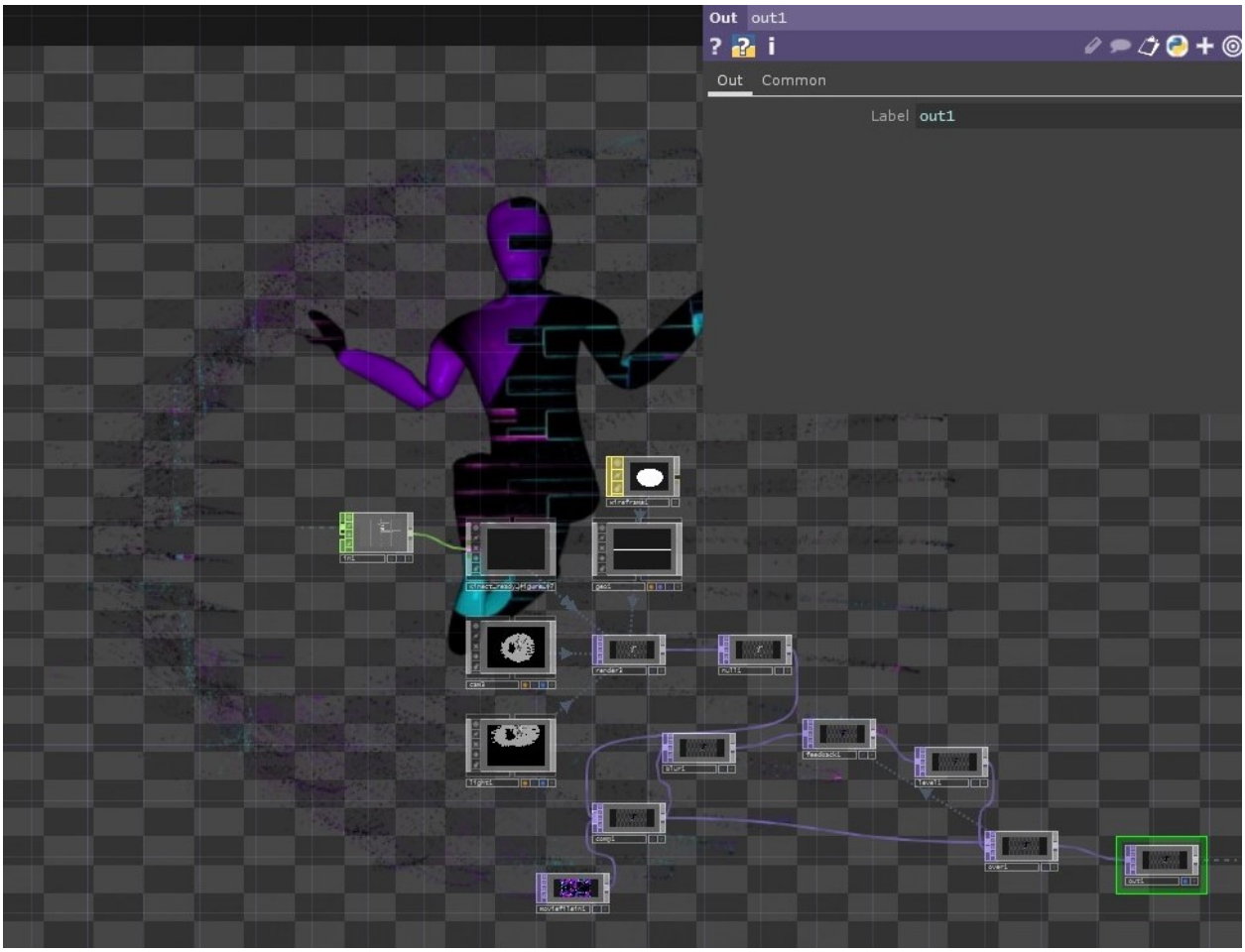


Figure 122: TD Example Kinect Avatar with Animating Texture

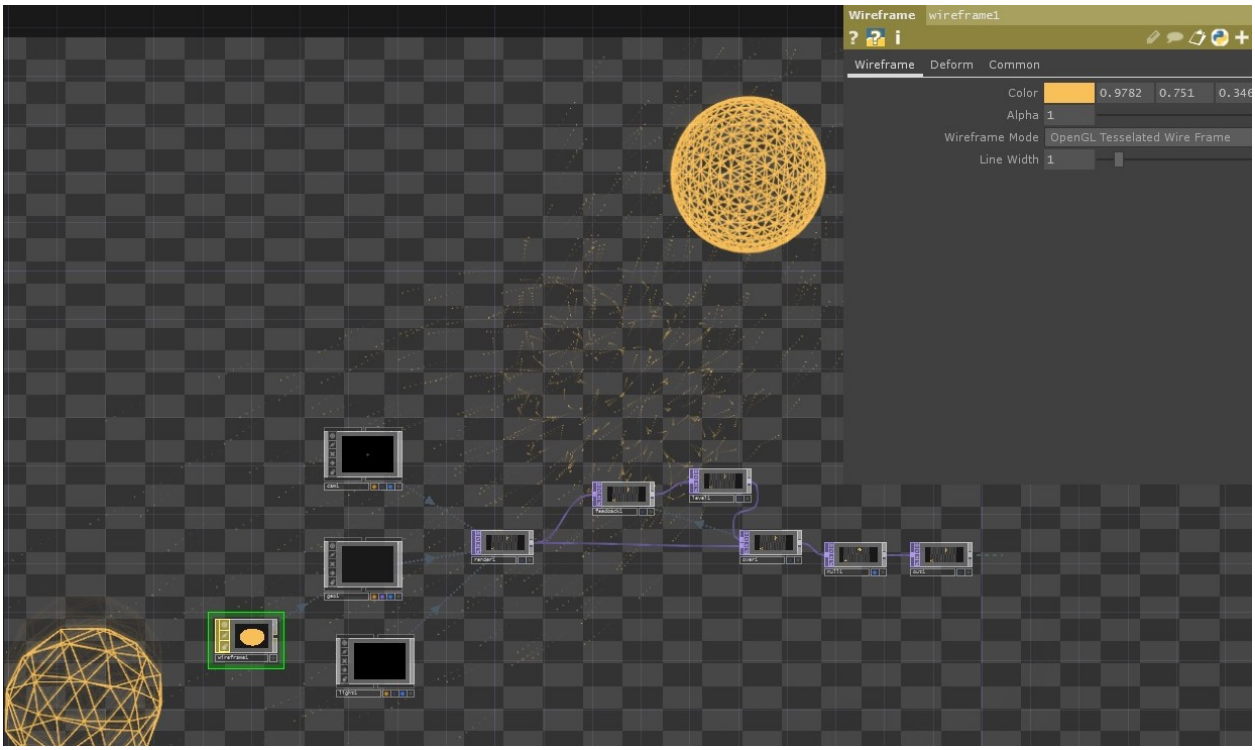


Figure 123: TD Example Mesh Spheres tracking hands positions, attracting particles

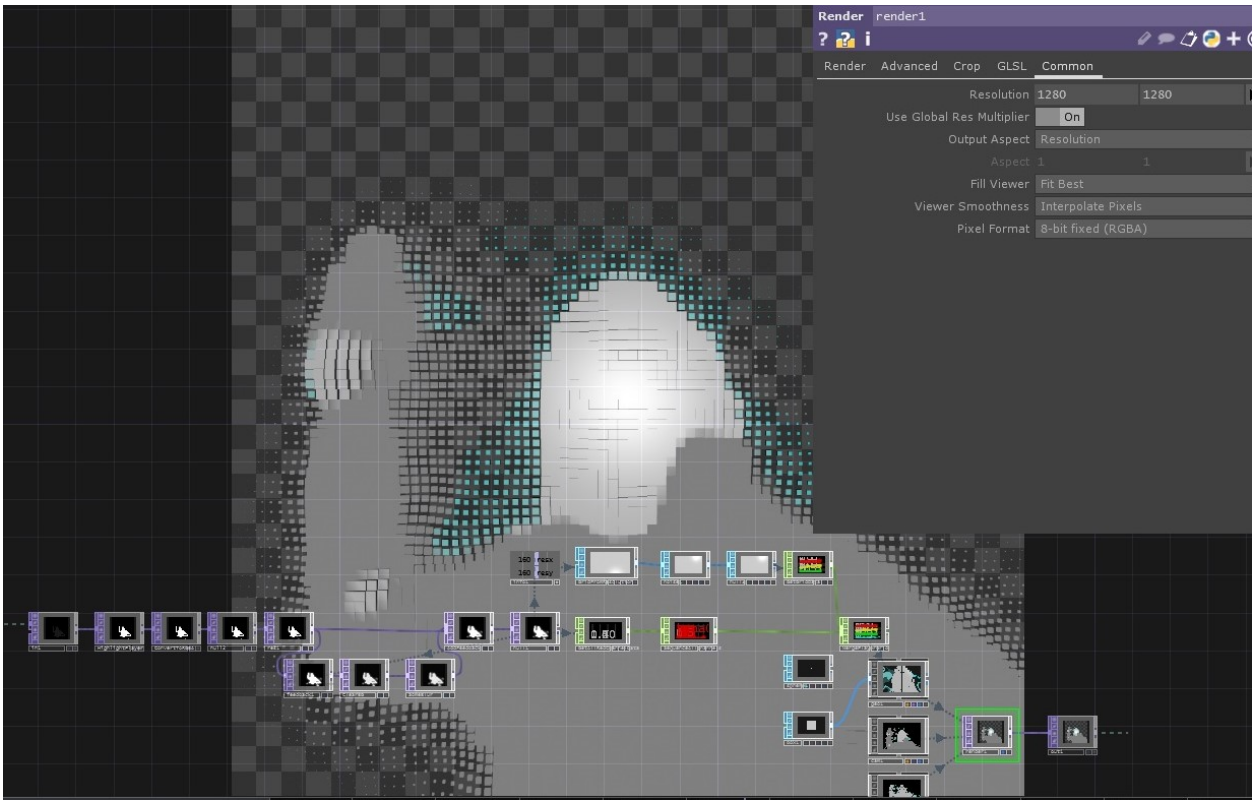


Figure 124: TD Example Kinect Visual GridPoints

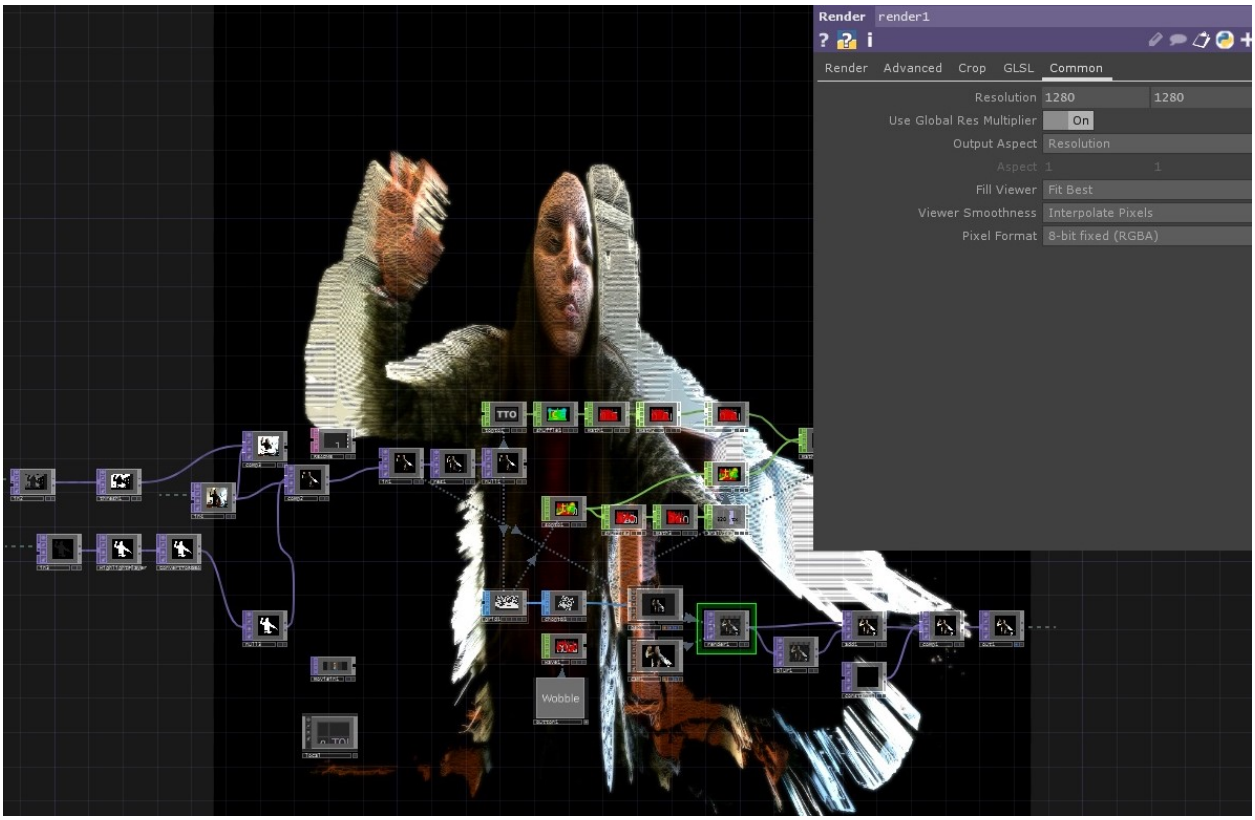


Figure 125: TD Example Kinect Visual Horizontal Voxel

Figure 126: TD Example Twisting Cube

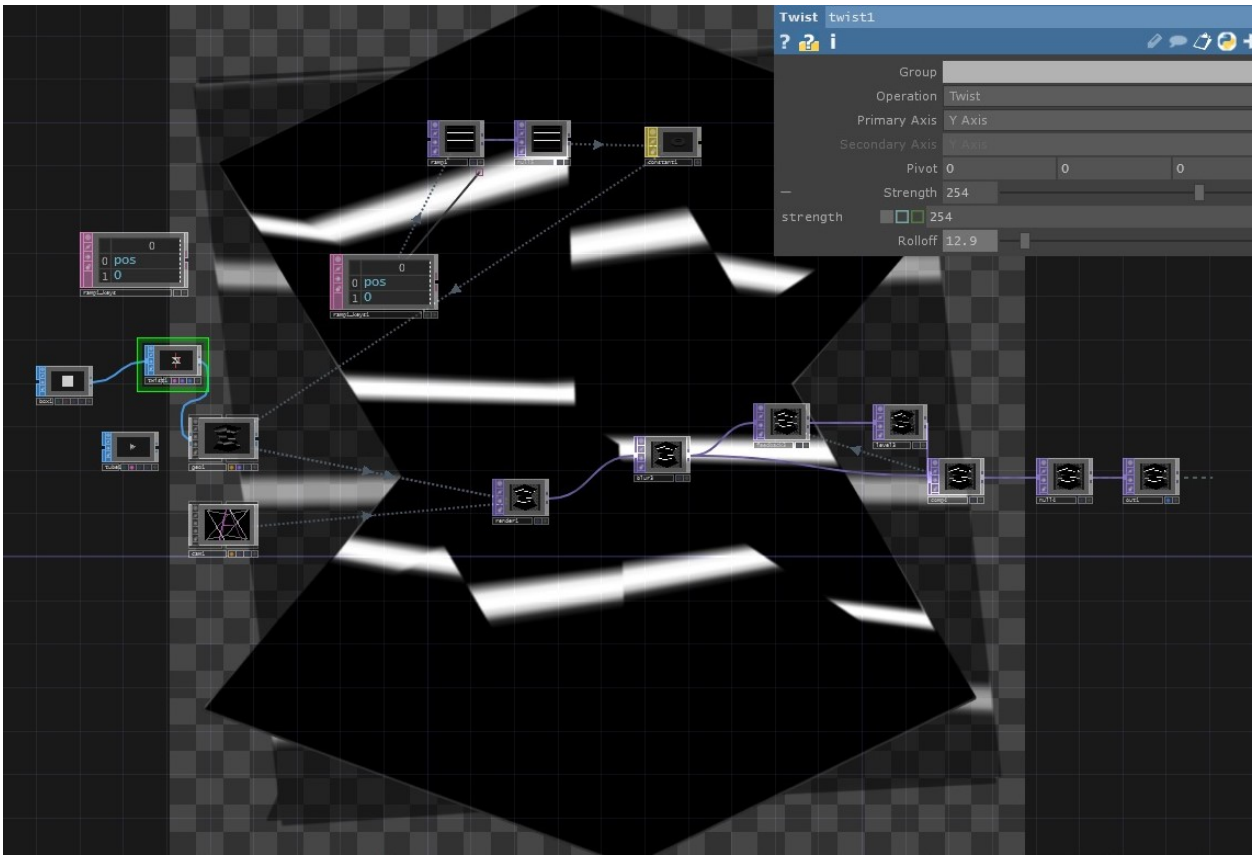


Figure 127: TD Example Animating Cubes

The final composition is a big network of processing nodes, most of which contain other nodes networks, as we can see in the following figure:

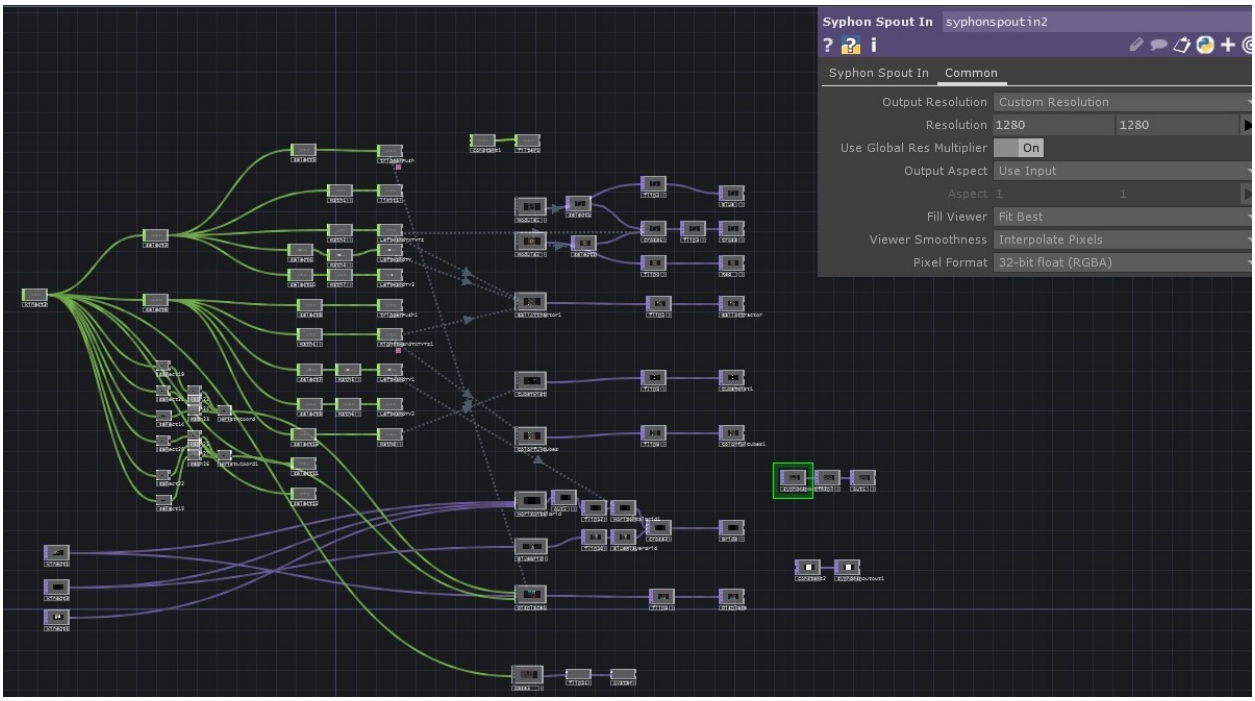


Figure 128: TD Example Final Network

6.4.3 Physical scene setup

To continue with our Demo, we needed to decide on a Canvas, videlicet an arbitrart 3D surface. To try staying within budget, we focused on smaller details that will catch the viewer's attention, instead of creating a giant set-up, which is usually the concept in Projection Mapping. Thus, we constructed a cardboard box from scratch using cardboard paper and tape. To display, bright images and in the correct photometric parameters, we covered the necessary sides of the box with white sheet paper, as we can see in the figure below:

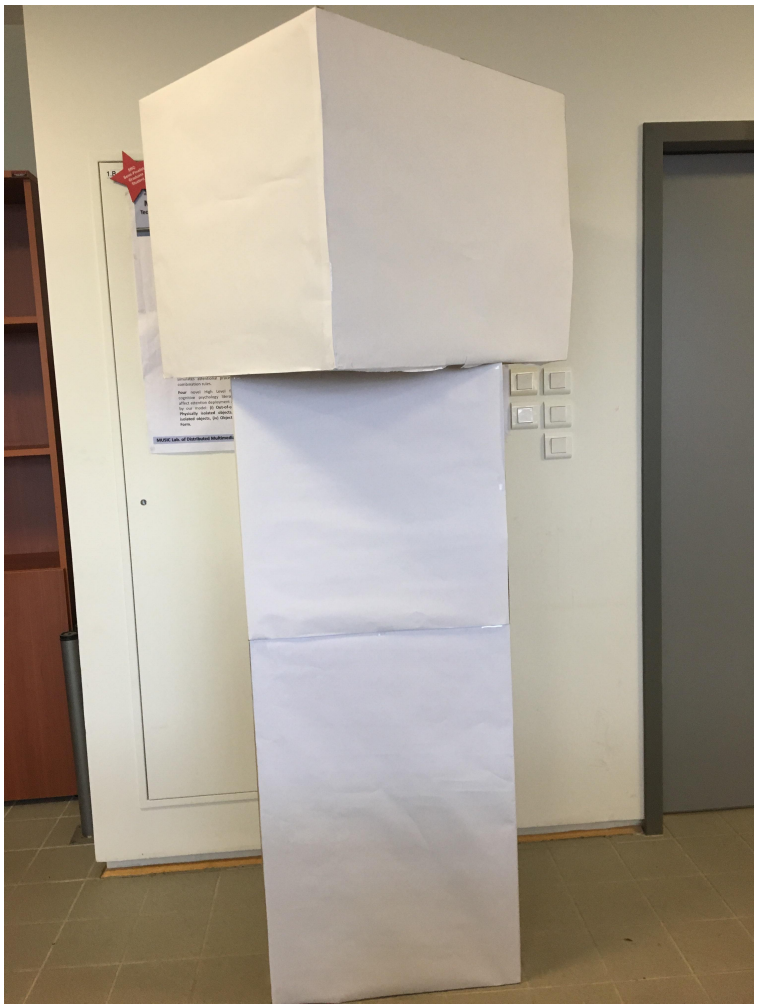


Figure 129: Physical Scene Set-Up: Cardboard Cubes coated in White Sheet Paper.

To perform our Interactive Projection Mapping project, we scale oout the TD network and add a "Window"-Comp Operator, where we specify the number of the monitor that will be utilized for the projection and press the "Perform" Button contained in this operator, to enter the Performing Mode of TouchDesigner, as we can see in the figure below.

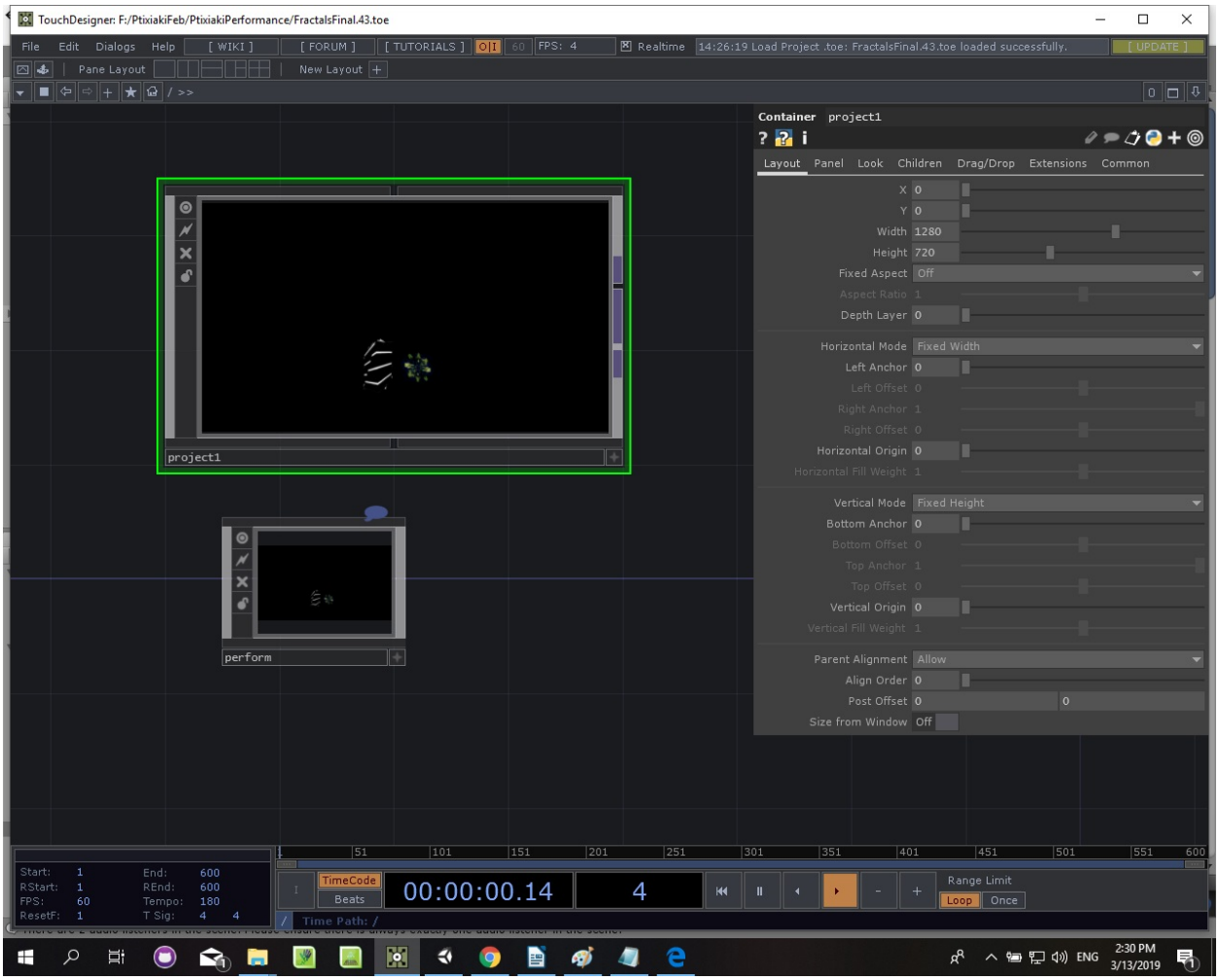


Figure 130: TD: Performance Operator.

6.4.4 Using the SceneWizard

The upper boxe's size is 60cm x 75cm, which in Unity's scaling units will be represented as 1(unity)m x 1.25(unity)m, correspondingly. The position and rotation points are set, according to the distance between the projector and our cardboard box. Thus, using SceneWizard's editor window, we create four different items-surfaces. Each item represents one of the two visible (from the projector) sides of the box, hence we choose "Plane" as a type for our items.

The next thing is to type the appropriate visual names, as well as select a gesture for each item. On the figures below we can see an example of setting up an item and how the fields must match between the used software.

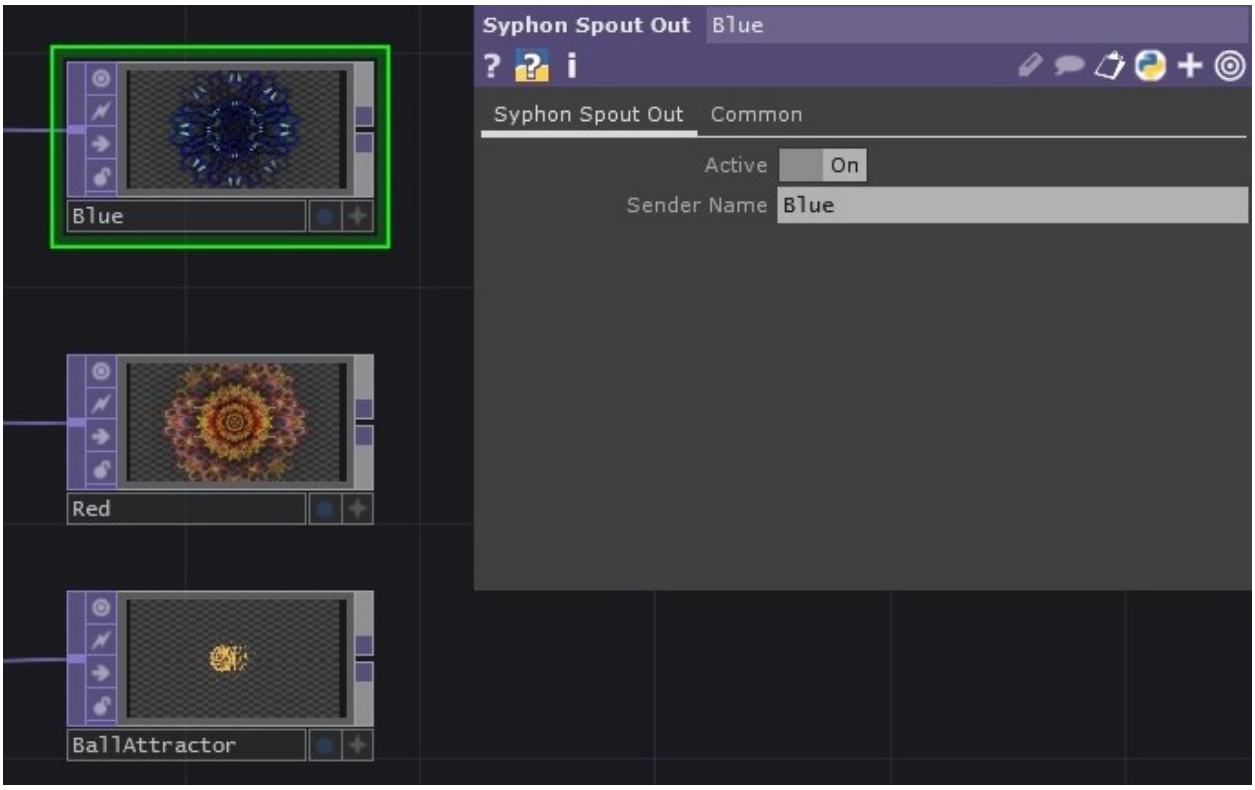


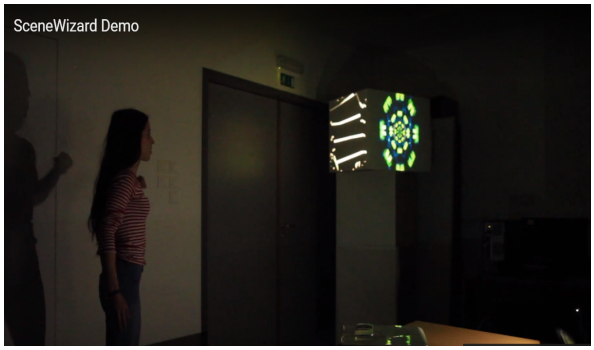
Figure 131: TouchDesigner Spout Output (Syphon Spout Out) Operators

6.4.5 Virtual scene setup through Play Mode

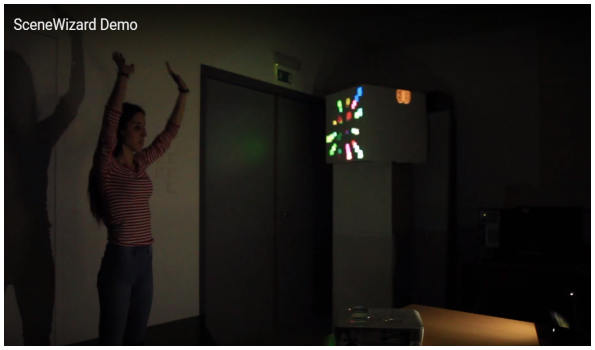
The last step before actually performing our Interactive Projection Mapping project, is to manually align our items-surfaces, using SceneWizard, to correct any details in registering the images accordingly. Below we can see an example of us, trying to set-up our virtual scene, using SceneWizard's tools.

6.4.6 Performing

Finally, we reached our goal in designing an Interactive Projection Mapping performance, using SceneWizard. The video below, demonstrates our performance: https://www.youtube.com/watch?v=Pxq6_FkXQy4&t=189s below, we can see some snapshots, while performing our Demo.



(a) Initial Interactive & Mapped Scene



(b) Switch Visual on Right Plane with Gesture: "HandsAboveHead"

Figure 132: Demo Scene Example 1

(a) Switch Visual on Left Plane with Gesture:"Punch_LeftHand"

(b) Next Scene-Version with "Space" Key

Figure 133: Demo Scene Example 2

Figure 134: Second Scene-Version After Gestures

7 Conclusion

7.1 Summary

In our a tool we have designed a new SDK, created for Unity Game Engine, that artists, designers and programmers can use to compose an Interactive Projection Mapping performance, or installation. Our tool was created, as a means of incorporating gesture based interactions with projected visual effects on arbitrary surfaces. As a result, there are still several issues of Projection Mapping that remain unresolved, such as the Scene Calibration, which nearby completely automates the image registration techniques. However, our main concern was to introduce a system for Unity, in order to include user interactions with projected visuals, using their body movements. SceneWizard introduces this concept. We designed a system with specific architecture for organizing Interactive Projection Mapping projects, as well as tools that partially automate the process of the registrating projected images from virtual to real world. We also designed a gesture database, to automate the process of performing gestures. We mapped these tools on a User Interface within Unity and provided a full system that can be easily adhered to any project in unity, simply by importing our unitypackage file, as well as the packages needed for the sensor recognition and videosharing settings.

7.2 Future Work

Notwithstanding the foregoing, there are also few issues we faced that could not be resolved in our current time frame. We will outline these parts here in hopes of future improvement from other developers.

7.2.1 Expand and Improve Gesture Database

As we discussed in the User Study section, we encountered several issues with Kinect Depth Sensor. One of those for example, was when adding more than one users, fact that perplexed the situation and resulted in a "disorientation" of the sensor. Moreover, our gesture database is mainly a sample database of gestures, which can be furtherly expanded and trained or even enriched with additional gestures. We can improve the detection and optimize our tool by adding miscellaneous and varicolored clips, to feed our algorithmic classifier with more information. This will undoubtedly result in a more precise and distinct

detection of our gestures. Additionally, accumulating different gestures in the same gesture database results in a more efficient detector, but also expand our interaction possibilities. Regardless, any supplementation in the gesture database, can only result in the amplification of the gesture detector.

7.2.2 Expand Type of Interactions

Since SceneWizard is devoted on implementing discreet gestures, it would be beneficial if we shift our focus on using continuous gestures too. When creating continuous gestures, we acquire additional data, such as floating point values, which can be used to obtain the exact progress value of the gesture. These data can subsequently be used in the code implementation in Unity, as means of regulating alternative parameters, such as the Texture's parameters. When performing continuous gestures, we can enhance the interactions and provide more enthralling results. In any case, adding continuous gestures in our database, will only assist our tool, both in the detection process, as well as in satisfactory received from a variety of interactions.

7.2.3 Automatic Scene Calibration with Kinect

As we mentioned before, there are various techniques that automate the process of the image registration issue of Projection Mapping. Applications like RoomAlive utilize projector-depth camera units, which are individually auto-calibrating, self-localizing, and create a unified model of the room with no user intervention. RoomAlive captures and analyses a unified 3D model of the appearance and geometry of the room, identifying planar surfaces like walls and the floor. This is used to adapt the augmented content to the particular room. In a similar manner, using Kinect Depth sensor we can automate the process of the current manual image-to-surface alignment, simply by calibrating our scene and referencing the objects identified in the scene, to our tool. By actualizing this, we save much course of work and time.

7.2.4 Dynamic Projection Mapping with SceneWizard

While Projection Mapping has been and active research field for quite a while, it focused predominantly on projecting on static objects. However, rapid advances in technology with upcoming high-speed projection hardware, when combined with the advantages of different optimized algorithms permit us to project on dynamic components. These components can represent either moving projector and cameras, or a scene that rigidly transforms during the projection, as in its position or form. We already discussed in the state-of-the-art the latest research applications in dynamic Projection Mapping, as studied from Grundhoffer's report. We can furtherly expand our tool, by implementing new methods that utilize Kinect Depth Sensor, or even use more than one Kinect Sensors to achieve Dynamic Projection Mapping, by detecting the moving components and estimating their positions. This will enhance the future potential of SceneWizard and provide us chance to create more complex performances and shows that will astound the croud.

7.2.5 Swapping out Spout Protocol

In order to have visual effects to present our tool, we needed to learn and utilize one common application, which is immensely used by the VJ community. In our case this application, specifically visual programming language, has been TouchDesigner. However, we decided to produce our tool for Unity cross-platform. As a result, we urged to seek a tool that will connect these two software for us, by sharing video content amongst them. In spite of this, Unity's support team is working hard to provide us with astonishing tools on creating visuals within Unity. At Unite LA 2018, their team unveiled the Visual Effect Graph, a tool for building real-time effects in Unity. Thus, we can consider utilizing just one software, while preparing an Interactive Projection Mapping show. This fact will result in an optimized performance, since while executing both software at the same time increases the computational complexity for our CPU and GPU and burdens our operation system.

7.2.6 Swapping out Unity environment

Since SceneWizard is a tool, created in and for Unity, it would be exceptionally beneficial, if we mapped all of SceneWizard's tools that currently work in Unity Editor, to a built User Interface that can be executed as a standalone application. Ideally, we would transfer the Scene View of Unity in our application and create buttons and controls to present SceneWizard and perform the appropriate actions to compose an interactive Projection Mapping project. This would be propitious for users that have never been in touch with Unity before and would act as an introductory tool to Unity's capabilities and resources. Additionally, there is an immersive demand from the VJ community on a simple application that could perform Projection Mapping and Unity is a great tool for creating applications. Therefore, this additive touch would cultivate our tool into a more approachable and engaging version.

7.2.7 Evaluating Alternatives to our Tools

Finally, when building SceneWizard we used several tools, based on a predifined architecture, such as Spout Protocol. Any tools we developed were either inspired from other applications, mimicking their functionality(run-time Mesh manipulation), or came to us during development(building a ScriptableObjects' System). Now that we have a general idea of what we want to achieve we would like to test some alternative solutions to the tools we developed.

In addition, we would also like to further test SceneWizard and add even more development tools that we did not came across during our design attempt, such as NDI protocol, for videosharing options. Moreover, we could try an alternative Depth Sensor, such as LeapMotion, for a more accurate hand-state detection, or even add up a marker-based tracking system for dynamic Projection Mapping. At last, conducting further testing in a wider audience would be beneficial to detect further flaws that require optimization.

References

[1] M. Dellepiane and R. Scopigno. Global refinement of image-to-geometry registration for color projection. In *2013 Digital Heritage International Congress (DigitalHeritage)*, volume 1, pages 39–46, Oct 2013.

[2] Paul Viola and William M. Wells III. Alignment by maximization of mutual information. *International Journal of Computer Vision*, 24(2):137–154, Sep 1997.

[3] Massimiliano Corsini, Matteo Dellepiane, Federico Ponchio, and Roberto Scopigno. Image-to-geometry registration: a mutual information method exploiting illumination-related geometric properties. *Computer Graphics Forum*, 28(7):1755–1764, 2009.

[4] W. Zhao, D. Nister, and S. Hsu. Alignment of continuous video onto 3d point clouds. In *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2004. CVPR 2004.*, volume 2, pages II–II, June 2004.

[5] Raskar R. Bimber O. Spatial augmented reality, 2005,New York: A K Peters/CRC Press.

[6] Teejo Renaud. Projection mapping, 2014, New York: Seminar January.

[7] ContraVision.

[8] Oliver Bimber, Daisuke Iwai, Gordon Wetzstein, and Anselm Grundhöfer. The visual computing of projector-camera systems. *Computer Graphics Forum*, 27(8):2219–2245, 2008.

[9] A. Grundhöfer and D. Iwai. Recent advances in projection mapping algorithms, hardware and applications. *Computer Graphics Forum*, 37(2):653–675, 2018.

[10] Brett Jones, Rajinder Sodhi, Michael Murdock, Ravish Mehra, Hrvoje Benko, Andrew Wilson, Eyal Ofek, Blair MacIntyre, Nikunj Raghuvanshi, and Lior Shapira. Roomalive: Magical experiences enabled by scalable, adaptive projector-camera units. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, UIST '14, pages 637–644, New York, NY, USA, 2014. ACM.

[11] Flexible camera calibration by viewing a plane from unknown orientations. In *Proceedings of the Seventh IEEE International Conference on Computer Vision*, volume 1, pages 666–673 vol.1, Sep. 1999.

[12] S. Yamazaki, M. Mochimaru, and T. Kanade. Simultaneous self-calibration of a projector and a camera using structured light. In *CVPR 2011 WORKSHOPS*, pages 60–67, June 2011.

[13] S. Garrido-Jurado, R. Muñoz-Salinas, F.J. Madrid-Cuevas, and M.J. Marín-Jiménez. Simultaneous reconstruction and calibration for multi-view structured light scanning. *Journal of Visual Communication and Image Representation*, 39:120 – 131, 2016.

[14] R. R. Garcia and A. Zakhor. Geometric calibration for a multi-camera-projector system. In *2013 IEEE Workshop on Applications of Computer Vision (WACV)*, pages 467–474, Jan 2013.

[15] S. Willi and A. Grundhöfer. Robust geometric self-calibration of generic multi-projector camera systems. In *2017 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*, pages 42–51, Oct 2017.

[16] Naoki Hashimoto. Robust geometric self-calibration of generic multi-projector camera systems, 2017.

[17] Y. Watanabe, T. Kato, and M. ishikawa. Extended dot cluster marker for high-speed 3d tracking in dynamic projection mapping. In *2017 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*, pages 52–61, Oct 2017.

[18] Parinya Punpongsanon, Daisuke Iwai, and Kosuke Sato. Projection-based visualization of tangential deformation of nonrigid surface by deformation estimation using infrared texture. *Virtual Reality*, 19(1):45–56, Mar 2015.

[19] Y. Fujimoto, R. T. Smith, T. Taketomi, G. Yamamoto, J. Miyazaki, H. Kato, and B. H. Thomas. Geometrically-correct projection-based texture mapping onto a deformable object. *IEEE Transactions on Visualization and Computer Graphics*, 20(4):540–549, April 2014.

[20] Amit H. Bermano, Markus Billeter, Daisuke Iwai, and Anselm Grundhöfer. Makeup lamps: Live augmentation of human faces via projection. *Computer Graphics Forum*, 36(2):311–323, 2017.

[21] Xinli Chen, Xubo Yang, Shuangjiu Xiao, and Meng Li. Color mixing property of a projector-camera system. In *Proceedings of the 5th ACM/IEEE International Workshop on Projector Camera Systems*, PROCAMS '08, pages 14:1–14:6, New York, NY, USA, 2008. ACM.

[22] C. Menk and R. Koch. Physically-based augmentation of real objects with virtual content under the influence of ambient light. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Workshops*, pages 25–32, June 2010.

[23] A. Grundhöfer. Practical non-linear photometric projector compensation. In *2013 IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 924–929, June 2013.

[24] A. Grundhöfer and D. Iwai. Robust, error-tolerant photometric projector compensation. *IEEE Transactions on Image Processing*, 24(12):5086–5099, Dec 2015.

[25] A. J. Law, D. G. Aliaga, and A. Majumder. Projector placement planning for high quality visualizations on real-world colored objects. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1633–1641, Nov 2010.

[26] O. Bimber, A. Grundhofer, T. Zeidler, D. Danch, and P. Kapakos. Compensating indirect scattering for immersive and semi-immersive projection displays. In *IEEE Virtual Reality Conference (VR 2006)*, pages 151–158, March 2006.

[27] Christian Siegl, Matteo Colaianni, Marc Stamminger, and Frank Bauer. Adaptive stray-light compensation in dynamic multi-projection mapping. *Computational Visual Media*, 3(3):263–271, Sep 2017.

[28] H. Habe, N. Saeki, and T. Matsuyama. Inter-reflection compensation for immersive projection display. In *2007 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–2, June 2007.

[29] G. Wetzstein and O. Bimber. Radiometric compensation through inverse light transport. In *15th Pacific Conference on Computer Graphics and Applications (PG'07)*, pages 391–399, Oct 2007.

[30] S. Takeda, D. Iwai, and K. Sato. Inter-reflection compensation of immersive projection display by spatio-temporal screen reflectance modulation. *IEEE Transactions on Visualization and Computer Graphics*, 22(4):1424–1431, April 2016.

[31] T. Amano and H. Kato. Appearance control using projection with model predictive control. In *2010 20th International Conference on Pattern Recognition*, pages 2832–2835, Aug 2010.

[32] T. Amano. Projection center calibration for a co-located projector camera system. In *2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 449–454, June 2014.

[33] T. Amano. Projection based real-time material appearance manipulation. In *2013 IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 918–923, June 2013.

[34] O. Bimber, D. Kloeck, T. Amano, A. Grundhoefer, and D. Kurz. Closed-loop feedback illumination for optical inverse tone-mapping in light microscopy. *IEEE Transactions on Visualization and Computer Graphics*, 17(6):857–870, June 2011.

[35] T. Amano. Shading illusion: A novel way for 3-d representation on paper media. In *2012 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pages 1–6, June 2012.

[36] O. Wang, M. Fuchs, C. Fuchs, H. P. A. Lensch, J. Davis, and H. Seidel. A context-aware light source. In *2010 IEEE International Conference on Computational Photography (ICCP)*, pages 1–8, March 2010.

[37] Gerwin Damberg, Anders Ballestad, Eric Kozak, Johannes Minor, Raveen Kumaran, and James Gregson. High brightness hdr projection using dynamic phase modulation. In *ACM SIGGRAPH 2015 Emerging Technologies*, SIGGRAPH '15, pages 13:1–13:1, New York, NY, USA, 2015. ACM.

[38] Reynald Hoskinson, Boris Stoeber, Wolfgang Heidrich, and Sidney Fels. Light reallocation for high contrast projection using an analog micromirror array. *ACM Trans. Graph.*, 29(6):165:1–165:10, December 2010.

[39] Oliver Bimber and Daisuke Iwai. Superimposing dynamic range. In *ACM SIGGRAPH Asia 2008 Papers*, SIGGRAPH Asia '08, pages 150:1–150:8, New York, NY, USA, 2008. ACM.

[40] S. Shimazu, D. Iwai, and K. Sato. 3d high dynamic range display system. In *2011 10th IEEE International Symposium on Mixed and Augmented Reality*, pages 235–236, Oct 2011.

[41] Brett R. Jones, Rajinder Sodhi, Pulkit Budhiraja, Kevin Karsch, Brian Bailey, and David Forsyth. Projectibles: Optimizing surface color for projection. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, UIST '15, pages 137–146, New York, NY, USA, 2015. ACM.

[42] S. Willi and A. Grundhöfer. Spatio-temporal point path analysis and optimization of a galvanoscopic scanning laser projector. *IEEE Transactions on Visualization and Computer Graphics*, 22(11):2377–2384, Nov 2016.

[43] Albert Ng, Julian Lepinski, Daniel Wigdor, Steven Sanders, and Paul Dietz. Designing for low-latency direct-touch input. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*, UIST '12, pages 453–464, New York, NY, USA, 2012. ACM.

[44] Sho Tatsuno Takeshi Yuasa Kiwamu Sumino Masatoshi Ishikawa Yoshihiro Watanabe, Gaku Narita. High-speed 8-bit image projector at 1,000 fps with 3 ms delay. *The International Display Workshops*, pages 1064–1065, March 2015/12.

[45] M. Regan and G. S. P. Miller. The problem of persistence with rotating displays. *IEEE Transactions on Visualization and Computer Graphics*, 23(4):1295–1301, April 2017.

[46] D. Iwai, K. Kodama, and K. Sato. Reducing motion blur artifact of foveal projection for a dynamic focus-plus-context display. *IEEE Transactions on Circuits and Systems for Video Technology*, 25(4):547–556, April 2015.

[47] T. Okatani, M. Wada, and K. Deguchi. Study of image quality of superimposed projection using multiple projectors. *IEEE Transactions on Image Processing*, 18(2):424–429, Feb 2009.

[48] Niranjan Damera-Venkata and Nelson L. Chang. Display supersampling. *ACM Trans. Graph.*, 28(1):9:1–9:19, February 2009.

[49] Will Allen and Robert Ulichney. 47.4: Invited paper: Wobulation: Doubling the addressed resolution of projection displays. *SID Symposium Digest of Technical Papers*, 36(1):1514–1517, 2005.

[50] Li Zhang and Shree Nayar. Projection defocus analysis for scene capture and image display. *ACM Trans. Graph.*, 25(3):907–915, July 2006.

[51] Kosuke Sato Daisuke Iwai, Shoichiro Mihara. Extended depth-of-field projector by fast focal sweep projection. *IEEE transactions on visualization and computer graphics.*, 21(4):462–470, 2015.

[52] Iwai D. & Sato K. Nagase, M. Projection defocus analysis for scene capture and image display. *Virtual Reality (2011)*, 15:119, July 2010.

[53] J. Jurik, A. Jones, M. Bolas, and P. Debevec. Prototyping a light field display involving direct observation of a video projector array. In *CVPR 2011 WORKSHOPS*, pages 15–20, June 2011.

[54] Stijn Roelandt Aykut Avci Herbert De Smet Hugo Thienpont Lawrence Bogaert, Youri Meuret. Single projector multiview displays: directional illumination compared to beam steering, 2010.

[55] Stijn Roelandt Jana Vanderheijden Aykut Avci Herbert De Smet Hugo Thienpont Youri Meuret, Lawrence Bogaert. Led projection architectures for stereoscopic and multiview 3d displays, 2010.

[56] Matthew Hirsch, Gordon Wetzstein, and Ramesh Raskar. A compressive light field projection system. *ACM Trans. Graph.*, 33(4):58:1–58:12, July 2014.

[57] Masahiro Yamaguchi Nagaaki Ohyama Takeyuki Ajito, Takashi Obi. Multiprimary color display for liquid crystal display projectors using diffraction grating. *Optical Engineering*, 38(11):1883 – 1888 – 6, 1999.

[58] Masahiro Yamaguchi Nagaaki Ohyama Takeyuki Ajito, Takashi Obi. Expanded color gamut reproduced by six-primary projection display, 2000.

[59] Isaac Kauvar, Samuel J. Yang, Liang Shi, Ian McDowall, and Gordon Wetzstein. Adaptive color display via perceptually-driven factored spectral projection. *ACM Trans. Graph.*, 34(6):165:1–165:10, October 2015.

[60] Yuqi Li, Aditi Majumder, Dongming Lu, and M. Gopi. Content-independent multi-spectral display using superimposed projections. *Computer Graphics Forum*, 34(2):337–348, 2015.

[61] Borderless TeamLad. Mori building digital art museum. https://borderless.teamlab.art/, 2018.

[62] Culturespaces. L'atelier des lumières. https://www.atelier-lumieres.com/en/home, 2016.

[63] B. R. Jones, R. Sodhi, R. H. Campbell, G. Garnett, and B. P. Bailey. Build your world and play in it: Interacting with surface particles on complex objects. In *2010 IEEE International Symposium on Mixed and Augmented Reality*, pages 165–174, Oct 2010.

[64] M. R. Mine, J. van Baar, A. Grundhofer, D. Rose, and B. Yang. Projection-based augmented reality in disney theme parks. *Computer*, 45(7):32–40, July 2012.

[65] Y. Sheng, T. C. Yapo, C. Young, and B. Cutler. A spatially augmented reality sketching interface for architectural daylighting design. *IEEE Transactions on Visualization and Computer Graphics*, 17(1):38–50, Jan 2011.

[66] J. Nasman and B. Cutler. Physical avatars in a projector-camera tangible user interface enhance quantitative simulation analysis and engagement. In *2013 IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 930–936, June 2013.

[67] A. Dolce, J. Nasman, and B. Cutler. Army: A study of multi-user interaction in spatially augmented games. In *2012 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pages 43–50, June 2012.

[68] Karl D.D. Willis, Ivan Poupyrev, Scott E. Hudson, and Moshe Mahler. Sidebyside: Ad-hoc multi-user interaction with handheld projectors. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, UIST '11, pages 431–440, New York, NY, USA, 2011. ACM.

[69] P. Punpongsanon, D. Iwai, and K. Sato. Softar: Visually manipulating haptic softness perception in spatial augmented reality. *IEEE Transactions on Visualization and Computer Graphics*, 21(11):1279–1288, Nov 2015.

[70] Iwai Daisuke Yoshikawa Yuki Watanabe Junji Nishida Shin'ya Ho, Hsin-Ni. Combining colour and temperature: A blue object is more likely to be judged as warm than a red object. *Scientific Reports*, 4, Nov 2014/07/03/online.

[71] Masahiro Nishizawa, Wanting Jiang, and Katsunori Okajima. Projective-ar system for customizing the appearance and taste of food. In *Proceedings of the 2016 Workshop on Multimodal Virtual and Augmented Reality*, MVAR '16, pages 6:1–6:6, New York, NY, USA, 2016. ACM.

[72] Brett R. Jones, Hrvoje Benko, Eyal Ofek, and Andrew D. Wilson. Illumiroom: Peripheral projected illusions for interactive experiences. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 869–878, New York, NY, USA, 2013. ACM.

[73] Brett R. Jones, Hrvoje Benko, Eyal Ofek, and Andrew D. Wilson. Illumiroom: Immersive experiences beyond the tv screen. *Commun. ACM*, 58(6):93–100, May 2015.

[74] G. YAMAMOTO. A palm interface with projector-camera system. *9th International Conference on Ubiquitous Computing, UbiComp 2007 Adjunct Proceedings, Innsbruck, Austria, September*, pages 276–279, 2007.

[75] Anselm Grundhöfer and Oliver Bimber. Virtualstudio2go: Digital video composition for real environments. In *ACM SIGGRAPH Asia 2008 Papers*, SIGGRAPH Asia '08, pages 151:1–151:8, New York, NY, USA, 2008. ACM.

[76] Daniel G. Aliaga, Alvin J. Law, and Yu Hong Yeung. A virtual restoration stage for real-world objects. In *ACM SIGGRAPH Asia 2008 Papers*, SIGGRAPH Asia '08, pages 149:1–149:10, New York, NY, USA, 2008. ACM.

[77] Daisuke Iwai and Kosuke Sato. Document search support by making physical documents transparent in projection-based mixed reality. *Virtual Reality*, 15(2):147–160, Jun 2011.

[78] M. Isogawa, D. Iwai, and K. Sato. Making graphical information visible in real shadows on interactive tabletops. *IEEE Transactions on Visualization and Computer Graphics*, 20(9):1293–1302, Sep. 2014.

[79] T. Amano and H. Kato. Appearance control by projector camera feedback for visually impaired. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Workshops*, pages 57–63, June 2010.

[80] Matthias Rauterberg. From gesture to action: Natural user interfaces. 1999.

[81] Microsoft. Kinect depth sensor(project natal), November 2010, https://developer.microsoft.com/en-us/windows/kinect.

[82] Andrew D. Wilson. Using a depth camera as a touch sensor. pages 69–72, 01 2010.

[83] J. Han, L. Shao, D. Xu, and J. Shotton. Enhanced computer vision with microsoft kinect sensor: A review. *IEEE Transactions on Cybernetics*, 43(5):1318–1334, Oct 2013.

[84] Z. Zhang. Microsoft kinect sensor and its effect. *IEEE MultiMedia*, 19(2):4–10, Feb 2012.

[85] [Uzun et al. An approach to 3d model application with kinect. volume 4, December 2017.

[86] M. Ballester Ripoll. "gesture recognition using a depth sensor and machine learning techniques. *FINAL BACHELOR THESIS*, 2016, http://hdl.handle.net/10251/77950.

[87] J. Appenrodt, S. Handrich, A. Al-Hamadi, and B. Michaelis. Multi stereo camera data fusion for fingertip detection in gesture recognition systems. In *2010 International Conference of Soft Computing and Pattern Recognition*, pages 35–40, Dec 2010.

[88] Nguyen Dang Binh, Enokida Shuichi, and Toshiaki Ejima. Real-time hand tracking and gesture recognition system. 2005.

[89] Feng-Sheng Chen, Chih-Ming Fu, and Chung-Lin Huang. Hand gesture recognition using a real-time tracking method and hidden markov models q. 2003.

[90] Microsoft. Kinect api, microsoft, windows. *Microsoft, Windows*, https://docs.microsoft.com/en-us/previous-versions/windows/kinect/dn758774%28v%3dieb.10%29.

[91] Oliver Wasenmüller and Didier Stricker. Comparison of kinect v1 and v2 depth images in terms of accuracy and precision. 11 2016.

[92] A. Amini, K. Banitsas, and J. Cosmas. A comparison between heuristic and machine learning techniques in fall detection using kinect v2. In *2016 IEEE International Symposium on Medical Measurements and Applications (MeMeA)*, pages 1–6, May 2016.

[93] Pushmeet Kohli Zhengyou Zhang Ling Shao, Jungong Han. In *Computer Vision and Machine Learning with RGB-D Sensors*. Springer, Cham, Dec Switzerland 2014.

[94] Robert E. Schapire Yoav Freund. A short introduction to boosting. *Journal of Japanese Society for Artificial Intelligence*, pages 771–780, 1999.

[95] Jia Sheng. A study of adaboost in 3d gesture recognition. *Department of Computer Science*.

[96] Yi Yao and Chang-Tsun Li. Hand posture recognition using surf with adaptive boosting. *British Machine Vision Conference Workshop*, 2012.

[97] K. Lai, J. Konrad, and P. Ishwar. A gesture-driven computer interface using kinect. In *2012 IEEE Southwest Symposium on Image Analysis and Interpretation*, pages 185–188, April 2012.

[98] Y. Wang, C. Yang, X. Wu, S. Xu, and H. Li. Kinect based dynamic hand gesture recognition algorithm research. In *2012 4th International Conference on Intelligent Human-Machine Systems and Cybernetics*, volume 1, pages 274–279, Aug 2012.

[99] T. Le, M. Nguyen, and T. Nguyen. Human posture recognition using human skeleton provided by kinect. In *2013 International Conference on Computing, Management and Telecommunications (ComManTel)*, pages 340–345, Jan 2013.

[100] M. Oszust and M. Wysocki. Recognition of signed expressions observed by kinect sensor. In *2013 10th IEEE International Conference on Advanced Video and Signal Based Surveillance*, pages 220–225, Aug 2013.

[101] and and. A fast algorithm for vision-based hand gesture recognition for robot control. In *2006 IEEE 14th Signal Processing and Communications Applications*, pages 1–4, April 2006.

[102] A. K. Sadhu, S. Saha, A. Konar, and R. Janarthanan. Person identification using kinect sensor. In *Proceedings of The 2014 International Conference on Control, Instrumentation, Energy and Communication (CIEC)*, pages 214–218, Jan 2014.

[103] M. Parajuli, , , and D. Sharma. Senior health monitoring using kinect. In *2012 Fourth International Conference on Communications and Electronics (ICCE)*, pages 309–312, Aug 2012.

[104] Gilles Vandewalle, Pierre Maquet, and Derk-Jan Dijk. Light as a modulator of cognitive brain function. *Trends in Cognitive Sciences*, 13(10):429 – 438, 2009.

[105] Timo Partonen. The habitat. *Seasonal Affective Disorder: Practice and Research*, page 331, 2010.

[106] Jeffrey S. Bedwell Jay Pratt Greg L. West, Adam K. Anderson. Red diffuse light suppresses the accelerated perception of fea. In *Red Diffuse Light Suppresses the Accelerated Perception of Fear*, volume 21, pages 992–999, July 2010.

[107] Burkhart Kimberly and Phelps James R. Amber lenses to block blue light and improve sleep: A randomized trial. *Chronobiology International*, 26(8):1602–1612, 2009.

[108] Gilles V, Christina Schmidt, Geneviève Albouy, Virginie Sterpenich, Annabelle Darsaud, Géraldine Rauchs, and Pierre yves Berken. Brain responses to violet, blue, and green monochromatic light exposures in humans: Prominent role of blue light and the brainstem.

[109] Raymond W. Lam, Anthony J. Levitt, Robert D. Levitan, Erin E. Michalak, Amy H. Cheung, Rachel Morehouse, Rajamannar Ramasubbu, Lakshmi N. Yatham, and Edwin M. Tam. Efficacy of Bright Light Treatment, Fluoxetine, and the Combination in Patients With Nonseasonal Major Depressive Disorder: A Randomized Clinical TrialBright Light Treatment and Fluoxetine for Nonseasonal DepressionBright Light Treatment and Fluoxetine for Nonseasonal Depression. *JAMA Psychiatry*, 73(1):56–63, 01 2016.

[110] Jo Phipps-Nelson, Jennifer R. Redman, Derk-Jan Dijk, and Shantha M. W. Rajaratnam. Daytime Exposure to Bright Light, as Compared to Dim Light, Decreases Sleepiness and Improves Psychomotor Vigilance Performance. *Sleep*, 26(6):695–700, 09 2003.

[111] Julie Carmigniani, Borko Furht, Marco Anisetti, Paolo Ceravolo, Ernesto Damiani, and Misa Ivkovic. Augmented reality technologies, systems and applications. *Multimedia Tools and Applications*, 51(1):341–377, Jan 2011.

[112] John Montgomery. September 2004.

[113] Heavym. *HeavyM*, https://heavym.net/en/.

[114] Openpool: Digital billiard diy kit. *OpenPool: Digital Billiard DIY Kit*, https://www.kickstarter.com/projects/511536811/openpool-digital-billiard-diy-kit?ref=live.

[115] Dalzieland Pow. Conductive ink + projection mapping = magic storytelling. *Conductive Ink + Projection Mapping = Magic Storytelling*, http://projection-mapping.org/conductive-ink-projection-mapping-magic-storytelling/.

[116] Mash Studio. Projection mapping on apple keyboard. *Projection Mapping on apple keyboard*, http://projection-mapping.org/projection-mapping-on-apple-keyboard/.

[117] produced by Leo Seele developed by B-Reel London. Projection mapping multiplayer game. *Projection Mapping Multiplayer Game*, http://projection-mapping.org/projection-mapping-multiplayer-game/.

[118] Brent Watanabe. Push play. *Push Play*, http://projection-mapping.org/video-game-installation/.

[119] Craig Downes. Umbra. *Umbra*, http://projection-mapping.org/umbra-interactive-dance-projection/.

[120] Sofia Aronov. Awake. *Awake*, https://www.bareconductive.com/news/awake-electric-paint-projection-mapping/.

[121] StoryLab. Interactive dining experience. *Interactive Dining Experience*, http://projection-mapping.org/interactive-dining-experience/.

[122] Hyunwoo Bang. Cloud pink. *Cloud Pink*, http://projection-mapping.org/cloud-pink-installation/.

[123] teamLab. Bordeless by teamlab. *Bordeless by TeamLab*, https://borderless.teamlab.art/.

[124] Jesse James. The enlightened one. *The Enlightened One*, http://projection-mapping.org/the-enlightened-one/.

[125] TD. Td. *TouchDesigner*, https://www.derivative.ca/.