PHYSICS DRIVEN 3D DYNAMIC GEOMETRY SOFTWARE FOR ELEMENTARY EDUCATION

DAVID ZELLER

Vienna University of Technology

January 2012 – version 0.1

ABSTRACT

Spatial abilities are an essential part of human recognition and general intelligence. They represent our ability to perceive spatial patterns, mentally manipulate two- or three-dimensional objects and orient ourselves within our environment. They are also a key factor when it comes to creative thinking and problem solving, and they determine our efficiency in information retrieval. The latter, in turn, is increasingly relevant in modern times as it is closely linked to our ability to navigate large websites or handle complex user interfaces. In consideration of the many ways spatial ability influences our everyday life, the work at hand presents a software intended to support its development in children. In order to achieve this goal, a virtual environment is combined with elements of geometry education, both of which have proven to be useful in improving spatial skills. Furthermore, a physics component is added to the mix, which leaves users with a virtual playground, in which they can construct complex scenes out of basic geometric objects.

The software has been developed in Unity 3D and uses a novel motion controller as input device, in order to increase the users' immersion into the virtual world. Embedded into an educational context, with specific instructions and work tasks, the application is believed to have beneficial effects on the development of spatial skills.

ZUSAMMENFASSUNG

Räumliche Fähigkeiten sind ein essentieller Bestandteil menschlicher Wahrnehmung und Intelligenz. Sie stellen unsere Fähigkeit dar räumliche Muster zu erkennen, zwei- oder dreidimensionale Objekte im Geiste zu manipulieren und uns innerhalb unserer Umwelt zu orientieren. Zudem spielen sie eine Schlüsselrolle in kreativen Denkprozessen, bei der Lösung von Problemen und beim Auffinden von Informationen in größeren Datenmengen. Letzterer Faktor gewinnt vor allem in der heutigen Zeit zunehmend an Bedeutung, da er in enger Verbindung zu unserer Fähigkeit steht, durch umfangreiche Websites zu navigieren bzw. uns in komplexen User Interfaces zurechtzufinden.

In Anbetracht der vielfältigen Arten und Weisen in denen räumliche Fähigkeiten unser tägliches Leben beeinflussen, wird in dieser Arbeit eine Software vorgestellt, deren Ziel es ist, die Entwicklung jener Fähigkeit bereits im Kindesalter zu unterstützen. Hierzu wird eine virtuelle Umgebung mit Elementen des Geometrie-Unterrichts kombiniert, wobei sich beide Faktoren als nützlich bei der Verbesserung von räumlichen Fähigkeiten herausgestellt haben. Zudem wird diesem

Mix eine Physik-Komponente hinzugefügt, wodurch die Applikation einem virtuellen Spielplatz gleicht, in dem Benutzer komplexe Szenen aus simplen geometrischen Objekten aufbauen können.

Die Software wurde in Unity 3D entwickelt und benutzt einen neuartigen Motion Controller als Eingabegerät, um den Benutzern zu erlauben, leichter und vor allem tiefer in die virtuelle Welt einzutauchen. Eingebettet in den schulischen Unterricht, mit spezifischen Anweisungen und Arbeitsaufgaben, wird angenommen, dass das Programm einen durchwegs positiven Einfluss auf die Entwicklung räumlicher Fähigkeiten haben wird.

We know only what we do, what we make, what we construct;
and all that we make, all that we construct, are realities.

— Naum Gabo [48]

ACKNOWLEDGMENTS

I'd like to thank the academy...

CONTENTS

I THEORY 1

1 INTRODUCTION 2

2 RELATED WORK 5
- 2.1 Dynamic Geometry Software 5
- 2.2 Two Dimensions 7
  - 2.2.1 Cabri Geometry 7
  - 2.2.2 GeoGebra 8
- 2.3 Three Dimensions 10
  - 2.3.1 Cabri 3D 10
  - 2.3.2 Construct3D 11
- 2.4 Physics Driven 13
  - 2.4.1 Cinderella 13
  - 2.4.2 Algodoo 16

3 TECHNOLOGICAL FOUNDATION 19
- 3.1 Razer Hydra 19
- 3.2 OpenTracker 21
- 3.3 Unity3D 23

4 DIDACTICAL BACKGROUND 27
- 4.1 Spatial Abilities 27
  - 4.1.1 What is Spatial Ability... 28
  - 4.1.2 ... and how can it be improved? 30
- 4.2 Constructivist Learning Theory 33

II PRACTICE 36

5 DESIGN 37
- 5.1 General Appearance and 3D Interaction 37
  - 5.1.1 Crosshair Control 38
  - 5.1.2 Camera Control 40
- 5.2 Interacting with Objects 42
  - 5.2.1 Object Creation 42
  - 5.2.2 Object Manipulation 43
- 5.3 Physics 45
- 5.4 Graphical User Interface 47

6 IMPLEMENTATION 51
- 6.1 Tracking 51
  - 6.1.1 HydraEvent 52
  - 6.1.2 Event Handling 53
  - 6.1.3 Calibration 54
- 6.2 Primitives 55
  - 6.2.1 Basics 56
  - 6.2.2 Wireframes 62

[August 1, 2012 at 14:50 – classicthesis version 0.1]

6.2.3 Colliders 63
6.2.4 View-Dependent Scaling 65
6.3 Control Classes 66
6.3.1 Main 66
6.3.2 Navigation 68
6.4 Serialization 71
6.4.1 Object Data 72
6.4.2 Scene Data 74
6.5 GUI 76
7 CONCLUSION & FUTURE WORK 78

III APPENDIX 81
A APPENDIX A 82

BIBLIOGRAPHY 84

LIST OF FIGURES

Figure 1 Constrained and unconstrained objects in DGS 6
Figure 2 Cabri Geometry 8
Figure 3 GeoGebra 9
Figure 4 Cabri 3D 10
Figure 5 Construct3D 12
Figure 6 Cinderella - Multiple Viewports 14
Figure 7 Cinderella - Physics Simulation 15
Figure 8 Algodoo 17
Figure 9 Razer Hydra 20
Figure 10 OpenTracker Sample Configuration 22
Figure 11 OpenTracker Project Configuration 23
Figure 12 Unity: Game Objects, Components Variables 24
Figure 13 General Spatial Ability 28
Figure 14 Spatial Ability Tests 31
Figure 15 Initial State 38
Figure 16 Calibration Routine 38
Figure 17 View-Angle-Invariant Crosshair 40
Figure 18 Object Creation Menu 42
Figure 19 Wireframe Overlay 44
Figure 20 Selecting Multiple Objects via Drag and Drop 45
Figure 21 Colored Objects 47
Figure 22 Main GUI Menus 49
Figure 23 File Chooser 50
Figure 24 Event Handling 53
Figure 25 Primitive Object Classes 55
Figure 26 Torus radii 58
Figure 27 Multiple Vertices 59
Figure 28 Compound Collider 64
Figure 29 The Application in Use 79

LIST OF TABLES

Table 1 Crosshair & Camera Interaction 82
Table 2 GUI Interaction 82

Table 3 Object Interaction 83

LISTINGS

Listing 1 `POTorus.createMesh` 57
Listing 2 `POTorus.getTorusVertex` 58
Listing 3 `POTorus.getTorusUV` 58
Listing 4 `WireframeRenderer.OnRenderObject` 62
Listing 5 `PrimitiveObject.rotatePivot` 65
Listing 6 `Navigation.adjustCrosshair` 68
Listing 7 `Navigation.moveCamera` 70
Listing 8 Parts of a file containing raw mesh data 73
Listing 9 Contents of a scene file 74

Part I

THEORY

1 INTRODUCTION

When Sir Francis Galton experimentally inquired mental imagery in 1880, a new area of research was brought into being which has thrived and prospered into recent days. It was around said time that the ability to perceive spatial patterns, to orient oneself in the world and to mentally picture it was first recognized as a separate factor independent from general intelligence. Henceforward, many different aspects of what has been coined *spatial ability* have been at the core of researcher projects worldwide. A plethora of studies have been conducted on which individual factors actually constitute it, how they can be measured, how they develop in the course of childhood and adolescence and how they are differently pronounced in different sexes.

One aspect receiving increasing attention in recent days is whether or not human spatial abilities can explicitly be improved. Considering the fact that they affect many things in our everyday lives, a vital interest in this area is hardly surprising. For one thing, they are important for rather obvious tasks such as the mental manipulation of objects, environmental orientation and navigation. High levels of spatial ability are, however, also closely linked to creativity, which is not only important from an artistic point of view but also for problem solving in such abstract disciplines as mathematics. Moreover, performance in terms of information search and retrieval is heavily influenced by spatial ability, which, in turn, is a major concern in research areas such as Human Computer Interaction.

A way to specifically improve spatial skills would, thus, be more than desirable. In the scientific community, there is no common understanding, though, on whether such a targeted training is possible. A variety of studies does, however, suggest that even seemingly unrelated activities, such as playing an ordinary video game, can positively influence spatial ability. More obvious approaches, including traditional geometry education or exercise with dedicated applications in virtual environments also proved to have beneficial effects.

It was research results like these which basically inspired the work this thesis is based on. The combination of geometry education and information technology in the form of Dynamic Geometry Software has already proven successful. Thus, the idea was to build upon this approach and create an application for use in early stages of school in order to aid the development of spatial abilities in children. Considering the target audience, the application was, on the one hand, not

indeed required to provide complex mathematical functionality but, on the other hand, needed to be enjoyable and fun to use.

The concept of a strictly formal geometry software was, thus, traded in for a virtual geometry and physics playground with rather open interaction possibilities. Users should be able to create simple geometric objects, like cubes, spheres and cylinders and arrange these objects into more complex scenes by moving, rotating and scaling them. Different materials and colors should be available in order for children to design visually appealing scenes. It should, furthermore, be possible to let the objects in a scene interact with each other based on the laws of physics. Users should be able to pick up objects, toss them around or let them collide with others. While this physics component, certainly does have an educational effect as well, a major reason for its inclusion was that it potentially enhanced the software's playful character.

Another area of concern was to provide a means for easy navigation in 3D space and increasing the level of immersion into the virtual scene. With fully-fledged virtual reality setups being both too expensive and too complicated in terms of installation and configuration a choice was taken in favor of the Razer Hydra motion controller. The Hydra is a new motion sensing device providing six degrees of freedom for acquiring both location and orientation data at the same time. Due to its magnetic tracking approach and its advertised precision of one millimeter and degree, it supposedly provided sufficiently fine-grained input data for use with the proposed software.

The thesis at hand is going to give detailed insight into all steps that have been undertaken in the process of creating the envisioned application. First, to provide an overview of the current state-of-the-art in this domain, Chapter 2 is going to present a variety of Dynamic Geometry Software dealing with both two- and three-dimensional software. Moreover, applications including physical simulations are going to be presented as they are of special relevance to the work described herein.

Chapter 3, in turn, is going to give a brief introduction into the third-party technologies used during implementation. The hardware side, the Razer Hydra, is going to be discussed as well as the software side, the game engine Unity 3D and the virtual reality middleware OpenTracker.

Having covered its technological background, Chapter 4 will elaborate on the psychological and didactical theories this work is based on. There is going to be a detailed discussion of human spatial ability and how it can be improved as well as an introduction into the constructivist learning theory.

The actual software developed in the course of this thesis is going to be described in the following part. Chapter 5 will focus on design aspects, including a detailed description of the applications

appearance, its graphical user interface as well as all the available user interactions.

Finally, Chapter 6 is going to provide details on the software's actual implementation. There will be an in-depth discussion of all the relevant aspects from the event-handling system to the data-format used for storing scenes. Code-samples are going to be provided along the textual description in order to aid a better understanding of the respective classes' inner workings.

2 RELATED WORK

The work at hand deals with the development of a Dynamic Geometry Software for use in the early stages of education. In order to establish an appropriate context and put the work into perspective, the first section of this chapter is going to provide a brief introduction into Dynamic Geometry Software in general. Sections 2.2 and 2.3 are subsequently going to give an overview of the current state-of-the-art by presenting some of the more recent approaches in this domain dealing with both two- and three-dimensional geometry. Since the application created in the course of this thesis seeks to enhance the educational effect of traditional DGS by combining it with real-time physics, Section 2.4 is finally going to present recent takes on physics-driven geometry software.

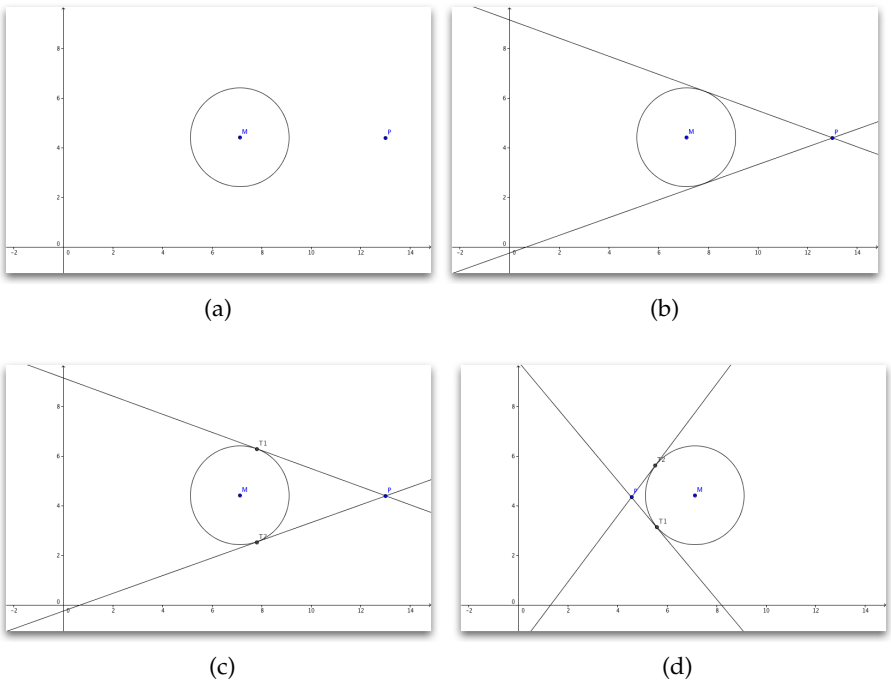
2.1 DYNAMIC GEOMETRY SOFTWARE

Along with the advent of computers, able to produce complex visual output in the 1980s, a new way of using these devices soon came into being. Applications falling into this new category of *Dynamic Geometry Software* (DGS) attempted to facilitate the creation of geometric drawings by basically offering digital versions of traditional tools such as rulers and compasses. With these tools at hand, accurate drawings could be produced with the greatest of ease even by people not profoundly skilled in manually doing so [32].

While not originally intended for this kind of use, these applications also quickly found their way into schools in order to support the teaching of geometry [32]. Said tendency towards an adoption in educational settings is also reflected in Sträßer's definition of Dynamic Geometry Software as [54]:

> *... software which is predominantly used for the construction and analysis of tasks and problems in elementary geometry.*

The popularity of Dynamic Geometry Software in educational contexts is mostly due to two reasons. On the one hand, as already mentioned above, the accuracy of the produced drawings is most often superior to that of drawings done with the aid of traditional tools. On the other hand, the characteristic ability of DGS solutions to record and redo construction steps is greatly beneficial in many teaching scenarios as it allows students to easily create and compare different configurations of the same figure [29].



(a) (b)

(c) (d)

Figure 1: A typical construction carried out with Dynamic Geometry Software. The unconstrained elements (i.e. the points M and P) are displayed blue while the constrained ones (i.e. the circle, tangents and intersection points) are displayed black.

Another important aspect of Dynamic Geometry Software, which is closely intertwined with said ability to recored construction histories, is the distinction between constrained and unconstrained parts of a figure [29]. As illustrated in Figure 1, a user might, for example, create a circle with a fixed radius and midpoint M and a single point P lying somewhere on the outside of it. He might then tell the system to automatically construct the two tangents to the circle running through P and subsequently find the intersection points $T1$ and $T2$ between the tangents and the circle. Features like these can be found in almost all recent DGS tools.

The described construction sequence yields two unconstrained elements, being the circle midpoint M and P. The circle and the tangents as well as the two intersection points, however, are considered constrained elements. The important difference between the two types of elements is that unconstrained ones can be moved freely around the workspace, which results in the entire construction being updated according to the established constraints. Thus, if a user desires to grab and move P to the opposite side of the circle, both the tangents and intersection points will instantly move along preserving their initial characteristics. Figure 1c and 1d provide an illustration of said behavior.

This inherent flexibility of Dynamic Geometry Software opens up wide application areas in educational and scientific contexts. For instance, it holds the possibility for students to not only be told and forced to believe certain theorems and definitions, but to actively explore and discover them. Conversely, it can be used as a classroom demonstration tool by teachers, in order to visualize the currently discussed content. With its possibilities to create constraints and complex relationships between elements, though, modern DGS tools are also increasingly used for simulating non-mathematical situations, such as moving robot arms or "a billiard ball bouncing off the sides of a square table" [29].

2.2 TWO DIMENSIONS

2.2.1 *Cabri Geometry*

With its first version being released in 1988, Cabri Géomètre (later Cabri Geometry) was one of the first applications falling into the domain of Dynamic Geometry Software [32]. The software has initially been developed by Jean-Marie Laborde and was, in its second version, published by Texas Instruments in the US and Canada. This collaboration with Texas Instruments also resulted in the tool being integrated into the TI-92, one of the company's graphing calculators, yielding a fairly wide spread of the software [3]. The most recent version of the software, Cabri Geometry II Plus[1], is currently available for Windows and Mac OS(X).

Laborde himself refers to the application as "a micro-world allowing multiple ways of exploring, experimenting and solving a problem" [35]. In order to enable the user to do so, he is provided with a canvas of one by one virtual meters, on which figures can freely be created (Figure 2 shows a screenshot of the application in Mac OSX). The main building blocks of these figures are all kinds of geometric objects ranging from the most simple one being a point to more complex ones such as polygons, curves or conic sections. Moreover the figures can be enhanced by adding descriptive elements like written text or formulae. [10].

Apart from the possibility to create complex geometric drawings, Cabri also offers tools that enable users to easily examine the dynamic aspects of a figure. These tools include pins to fix certain parts of a construction and make them immobile, a trace feature allowing users to comprehend the way an object changes when transformed and a simple animation feature for flipping elements around on the canvas [5].

Cabri also provides the possibility to define construction macros at any time within the creation of a figure, which allows users to auto-

1 http://www.cabri.com/cabri-2-plus.html

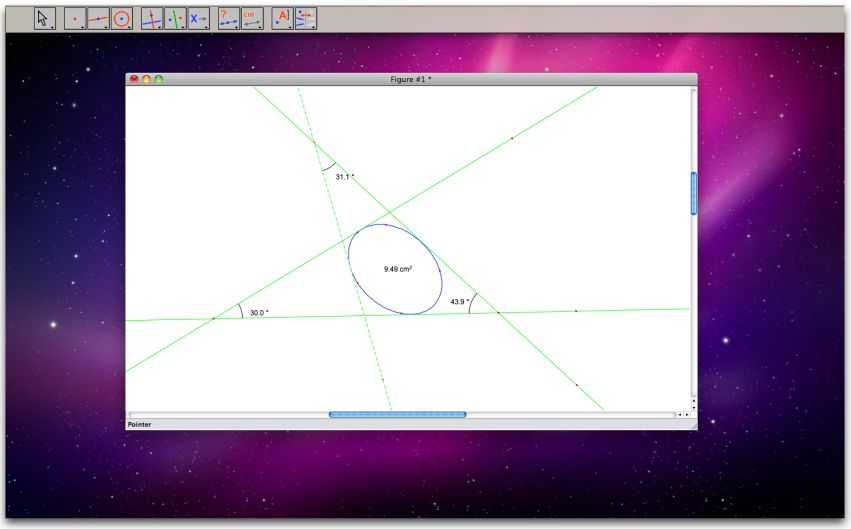


Figure 2: A screenshot of the latest version of Cabri Geometry in OSX, with the main toolbar at the top and the editable canvas in the center of the screen.

matically repeat this very creation process anytime later [10]. Furthermore, every work session with the application can be recorded and replayed, which, for instance, might be useful for teachers to gain insights into their students learning' progress [5].

2.2.2 *GeoGebra*

As the name implies, the open-source tool GeoGebra[2] aims to combine features of both geometry and algebra within a single software package [23]. It is intended by its creator Markus Hohenwarter as "a versatile tool for visualising mathematical ideas from elementary through to university level". GeoGebra is available in more than 35 languages and, being entirely written in Java, it is usable across all major platforms [25].

Reflecting the application's inherent duality, its main window is split up into an algebra view and a geometry view, as can be seen in Figure 3. Every element in the former view directly corresponds to an element in the latter, which enables users to not only directly interact with geometrical objects via drag and drop but also to indirectly do so via numeric input. In order to facilitate an understanding of the specific interrelations, changes made in one view are reflected in the other view in real-time.

GeoGebra provides a multitude of basic objects, such as "points, vectors, segments, polygons, straight lines, all conic sections" and many more. Additionally, users are provided with an input field,

2 http://www.geogebra.org/

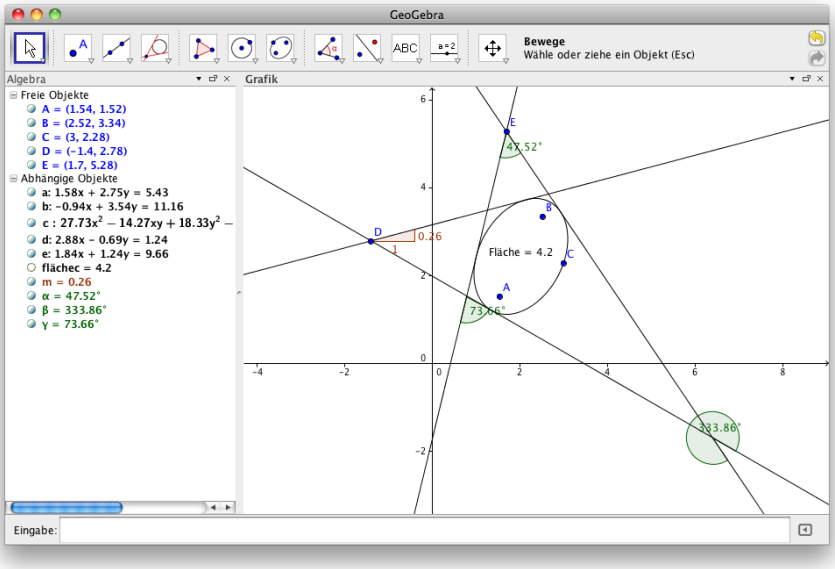


Figure 3: GeoGebra's main window. The right view displays the algebraic representations of the geometry in the left view.

which enables the creation of geometric objets by directly entering the respective equations. Within this field it is also possible to access existing elements and execute calculations that are not by default available in the application.

Apart from geometric objects, it is also possible to place a limited number of user interface elements in the geometry view [25]. These include input fields and sliders, the values of which can be bound to existing objects, buttons triggering custom functions as well as toggle switches to show or hide individual elements [6].

With GeoGebra being intended as an educational tool, the software also features e-learning functionality in that it allows users to create dynamic worksheets. Said worksheets are interactive HTML documents and can be viewed in every modern browser supporting Java without the need to have the application installed locally [24].

Initially conceived in the course of Hohenwarter's master thesis in 2002, the software quickly gained popularity amongst teachers worldwide after it was published on the internet. The positive feedback was furthermore reflected "by several educational software awards, including the European Academic Software Award 2002". Eventually, the software's unexpected success even culminated in the foundation of an *International GeoGebra Institute*, an organization dedicated to the promotion of software-supported mathematics education. The institute's main objectives are to foster the further development of the software and research projects affiliated with it as well as the active support of teachers and students in order to encourage enhanced classroom integration [26].

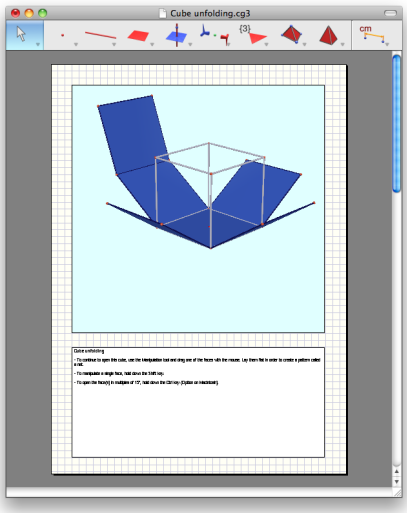


Figure 4: A Cabri 3D project illustrating the constituting polygons of a cube by providing an unfolded view of it.

2.3 THREE DIMENSIONS

2.3.1 *Cabri 3D*

Cabri 3D[3] was, as the name implies, developed by the same company as the previously described Cabri Geometry and was first released in 2004. The software claimed to be as much of a revolution to geometry education in three-dimensional as its counterpart was in two-dimensional space [20]. As such, it borrows many of the features, procedures and workflows common to 2D Dynamic Geometry Software [53].

The software runs in a single window, which is, by default, divided into a narrow toolbar at the top and an arbitrary number of interactive worksheets covering up the rest of the user interface. As depicted in Figure 4 these worksheets are intended to resemble pages in a paper notepad with a description of the current assignment at the bottom. Above this space for explanatory text and instructions, though, the pages are augmented with a view into a three-dimensional scene, which is where the actual work is carried out.

Within said scene, users are able to create a wide variety of three-dimensional objects of different complexity. Planes, spheres, polyhedra and many other geometric primitives can be constructed by either defining new points in the scene or utilizing already existent elements. Once created, these objects can be viewed from freely selectable angles and certain objects (all variations of polyhedra essentially) can even be opened up. When doing so the object effectively gets unfolded, allowing the user to gain a better overview of the polygons

3 http://www.cabri.com/cabri-3d.html

it is made up of (see Figure 4 for illustration). In addition to the geometric shapes that are available by default, users can create entirely new ones by simply intersecting existing objects with planes [20].

Being intended for actual geometry education, there are also tools for executing a variety of calculations in order to obtain detailed information about the current scene. These include simple computations such as the distance between two points in space up to more complex ones such as the volume of a complex polyhedron. Furthermore, the intersection lines and curves between arbitrary objects and planes can be calculated and displayed with the greatest of ease.

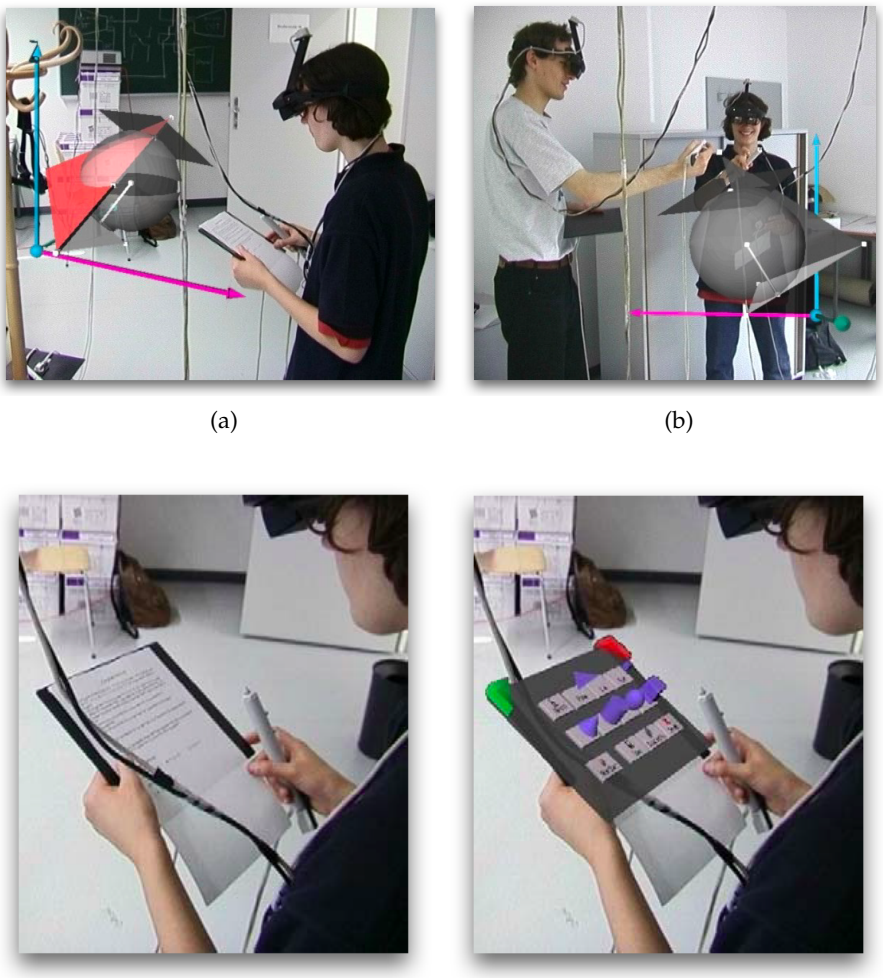
2.3.2 *Construct3D*

Developed by Hannes Kaufmann in the course of his PhD-thesis [27], Construct3D is a Dynamic Geometry Software intended for educational use from high-school to university. It deals with three-dimensional geometry and uses an augmented reality approach in order to create a greater level of immersion. The desired result of this increased immersion is a more intuitive comprehension of the subject matter on the part of the users as they are now working with actual three-dimensional objects instead of two-dimensional representations of the same.

The system is based on the Austrian Studierstube project[4], which strives to provide a framework for the development of collaborative augmented reality applications. Being an application of this kind, a special hardware setup is required to actually work with Construct3D. This setup includes a head-mounted display to overlay virtual content over the user's natural view, a pen for interacting with the virtual elements and a pad serving as the physical base for the virtual user interface. All of these components' positions and orientations need to be tracked in oder to provide for accurate visual output on the one hand and to enable interaction in the first place on the other hand. The actual tracking mechanism used is irrelevant to the system and can easily be switched, which, in fact, also happened during the creation of Construct3D itself.

Interaction in Construct3D is carried out via a clickable hand-held stylus, which can be moved around freely in space and basically acts as a three-dimensional mouse pointer. It is used to create, select and modify objects or individual elements of an object as well as to interact with the application's custom user interface. Said interface is realized as a set of standard GUI elements and controls, such as buttons and sliders which are mapped onto a portable pad. This way the user literally carries the user interface in the palm of his hand and interacts with it much in the same way as he would with a scratchpad, as can be seen in Figure 5c and 5d.

4 http://www.icg.tugraz.at/project/studierstube



(a) (b)

(c) (d)

Figure 5: Construct3D being used by a single user (a) and multiple users at the same time (b). The system's user interface (d) is mapped onto a physical pad (c) carried around by the user. Source: *Geometry Education with Augmented Reality* [27]

Apart from the entirely different interaction approach, the application provides all of the functionality that can be expected from Dynamic Geometry Software. A multitude of geometric primitives is available including both two-dimensional shapes, such as planes, circles and ellipses as well as three-dimensional objects, such as cuboids, spheres and cylinders. Moreover, Construct3D includes support for B-Splines, NURBS surfaces and surfaces created by rotational sweep operations.

Once one or more objects are created, the user is provided with a variety of geometric operations that can be carried out on them. These reach from the calculation of surface normals and tangential curves or surfaces to more sophisticated operations such as the computation of intersection points and curves. Moreover, entirely new objects can be

created by either using planar slicing or combining several objects via the three boolean operations Union, Difference and Intersection. Construct3D also supports the exact measurement of distances between any two points in the scene.

Being an augmented reality application users can easily view any geometric object or construction from all possible angles by simply moving around it in space. In order to establish a link to traditional geometry education, though, Construct3D provides orthographic top, front and side views, which can be displayed in addition to the actual three-dimensional object. This feature is intended to facilitate students' comprehension of the relation between spatial objects and their planar projections, which they usually deal with in geometry classes.

Construct3D also offers three different collaboration modes and thus allows multiple people to use the system simultaneously (see Figure 5b). For one thing, users can work in a collaborative mode, meaning that all objects and constructions are visible to and can also be manipulated by everyone. In independent mode, users are restricted to their very own constructions and cannot see or interact with objects created by others. Finally, in teacher mode, a single super user can freely set visibility for all other participating users.

2.4 PHYSICS DRIVEN

2.4.1 *Cinderella*

The Cinderella project[5], as conceived by Jürgen Richter-Gerbert and Henry Crapo in 1992, was originally intended as a pure Dynamic Geometry Software. The researchers' aim was to create a software, by the means of which users could easily create geometric figures "and then ask the computer about properties of these configurations." Due to its - by then - superior architecture, the application was initially targeted at the NeXT platform and a first prototype was finished after a few weeks into development. By and by, though, NeXT increasingly lost ground, which ultimately lead to a rewrite of the entire software in Java by Richter-Gerbert and Ulrich Kortenkamp [52].

Being a Dynamic Geometry Software at its core, Cinderella provides all the functionality common to this kind of application. On startup, the user is presented with a single window, featuring toolbars at the top and bottom and an empty canvas in the center part of the user interface. This canvas is where the user can freely place points and subsequently use them as anchors for the construction of lines, circles, polygons, conic sections, and many more geometric shapes. The system also provides means for the assisted construction of a variety of objects, such as orthogonal and parallel lines, angular

5 http://www.cinderella.de/

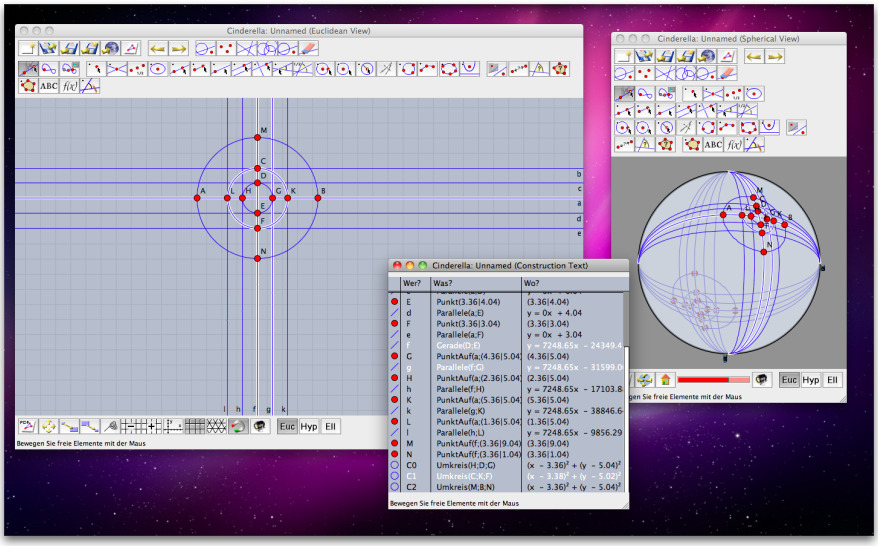


Figure 6: Cinderella simultaneously displaying the same construction in a euclidian and a spherical viewport along with its textual representation.

bisectors or objects mirrored along an arbitrary axis. Moreover, it is possible to add text labels and highlight certain aspects of a construction such as the angle between two lines.

In order to allow for detailed exploration of figures, Cinderella also provides users with the ability to measure distances between points or angles between lines. As the application also includes an animation feature and the possibility to draw an object's locus, the behavior of a construction under movement can be examined as well.

A feature distinguishing Cinderella from many other tools in the DGS domain is its ability to provide multiple views on the same construction, which is illustrated in Figure 6. Not only does this allow the user to get an overview of the entire scene side by side with a detailed view of certain aspects, but also to view the scene in an entirely different context. This is because Cinderella offers separate viewports displaying either a spherical projection of the current construction or its projection onto the hyperbolic plane. Furthermore a polar euclidian and a polar spherical viewport are supported as well as a viewport showing a textual description of the figure's evolution up to the current point.

Benefiting from the fact of being written entirely in Java, Cinderella offers an easy way to export constructions as applets embedded within webpages. Provided that the freely available Java runtime libraries are installed, these interactive applets can be viewed in any modern browser and do not require an actual copy of the software itself. Using this export functionality, teachers are able to effortlessly create e-learning material for students to study and explore geometric principles. As Cinderella features a built-in theorem checking engine, it

is also possible to create websites with fully interactive exercises, the solutions of which are automatically checked for correctness by the computer [32].

In 2006, the second version of Cinderella was released introducing a variety of new features. For instance, the software now offers users the possibility to manually enter functions and plot their respective graphs onto the canvas. Moreover a new sketching mode was introduced, allowing users to simply create hand-drawn scribbles of a construction, which are subsequently parsed and translated into an accurate version [33].

The two most important improvements, however, were the introduction of CindyScript and CindyLab. The former is a custom scripting environment, enabling users to write their own small programs, which are then executed from within the running application [34]. This way it is possible to create custom tools automating frequently re-occurring tasks [33] as well as to enhance constructions with special behavior or graphical elements [34].

CindyLab, in turn, provides means for simulating physical behavior within Cinderella. Users can build experimental setups much in the same way they would create geometric constructions and then

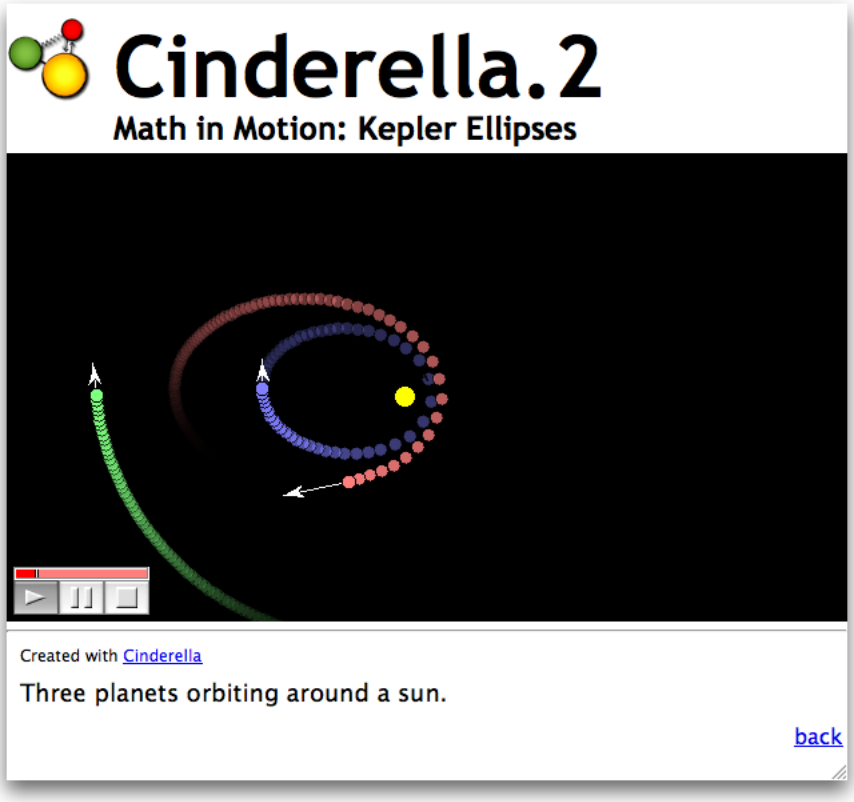


Figure 7: A physical simulation of three planets orbiting around a sun created with Cinderella and embedded into a website. Source: http://www.cinderella.de/files/HTMLDemos/6X01_Planets.html

run the simulation by hitting the start button. Abiding to the principles of Dynamic Geometry Software all elements can be grabbed and moved throughout the entire simulation yielding a physically correct reaction. The physics model employed in Cinderella is based on particles and forces. Particles are equipped with a certain mass and, under the influence of forces, move according to the laws of Newtonian mechanics. The user is provided with a variety of forces to use in an experiment including surface gravity, springs and rubber bands as well as simulations of the coulomb force, magnetic fields and planetary gravity fields [34]. Figure 7 depicts a simulation of planets orbiting around the sun, which has been exported to an applet and included into a website.

2.4.2 *Algodoo*

Although not actually falling into the domain of Dynamic Geometry Software, Algodoo[6] shall be presented herein as its playful approach to education corresponds well with the idea behind the software created in the course of this thesis. Algodoo is actively developed by the Swedish company Algoryx and is the successor of "the 2D physics sandbox Phun", which was the result of Emil Ernefeldt's Master Thesis at UmeåUniversity.

The basic principle behind the software is to convey the laws of physics by letting users experiment freely within an open environment. Particularly emphasizing playfulness, users are able to create even rather complex scenes with the greatest of ease by combining geometric objects with hinges, springs, motors, fluids, lasers and many other components. Algodoo currently enjoys great popularity, which is reflected in an extensive user community putting a lot of effort into the creation of a large database of re-usable scenes. While actually intended for use in more elementary levels of education, the software is, by now, also used in more sophisticated contexts [13].

As depicted in Figure 8, Algodoo runs in a single window but can also be switched into fullscreen mode if desired. After startup the window merely contains several toolbars, which are basically floating above a scene made up of nothing more than a solid ground plane. All buttons are equipped with tooltips popping up when hovering the mouse over them and providing a detailed explanation of how to use the respective tool. These extensive tooltips largely eliminate the need for going through a separate user manual before actually working with the application. When launching the application for the first time, though, a series of detailed tutorials is nevertheless provided guiding the novice user through the creation of several basic constructions and setups. Generally speaking, the entire look and feel of the application clearly targets a young audience and the develop-

6 http://www.algodoo.com/wiki/Home

Figure 8: The 2D physics sandbox Algodoo emphasizing playfulness and ease of use in educational software.

ers obviously invested a lot of effort into delivering a satisfying user experience.

The central element Algodoo scenes are made of are rigid bodies, which are physical objects that do not deform on collision or under pressure. They can, however, be assigned different materials which define how they react to collisions, whether they sink or swim in fluids and whether they fall to the ground or float in the air. In order to create a rigid body, the user is provided with a set of tools enabling the construction of boxes, circles and freehandedly drawn polygons. The system also offers the possibility to create gear wheels with random numbers of equally sized cogs so the individual wheels easily gear into each other [14].

Rigid bodies can be combined by merging them, which yields a single new object, or by interconnecting them with springs and hinges. Hinges can also act as motors, which can be configured by the user in terms of "target speed and maximum applied torque" [14]. Apart from motorized hinges, objects can also be driven by thrust jets, which can be applied anywhere on an object. The shear force of the thrust jet can be adjusted freely.

In addition to rigid body physics, Algodoo also supports fluid simulation. In order to work with fluids, every rigid body in the scene can be liquefied with a single mouse click, yielding bodies of water with correct density. Other rigid objects can then be dropped into the water and, according to their material, will either sink or swim in it [14].

Finally, the system also provides means for basic optics simulation, by allowing users to create lasers within the scene. The size of these lasers' emitted beams is as freely adjustable as the speed and intensity

of the light. The beam is correctly reflected and refracted by all the objects in the scene and can even be broken up into its color spectrum using prism-shaped bodies of translucent material [14].

Being a tool intended for educational use, Algodoo can also display a detailed description of the forces and impulses acting on the objects in the scene as well as their velocities. Moreover it provides access to a large database of lessons on physical concepts from within the application. These lessons are, amongst others, categorized in terms of age brackets and broader topic and each of them contains a detailed description along with a pre-fabricated scene for download. Apart from these educational scenes, the application also grants access to an archive of user-created content, most of which is of a playful character.

3 TECHNOLOGICAL FOUNDATION

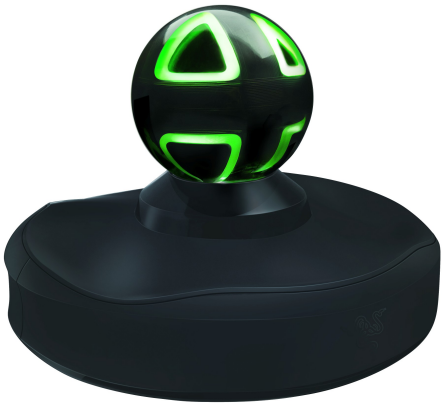
The main objective of this work was the development of a software for creating and manipulating geometric objects in three-dimensional space. While details about the actual implementation can be found in Chapter 6, the following part of this thesis is going to focus on the underlying technologies. First, the Razer Hydra motion controller, used as input device for the application, is going to be presented in detail. Subsequently, Section 3.2 will describe OpenTracker, a middleware technology enabling the communication between the controller and the system. Section 3.3 is finally going to provide introduction to Unity 3D, a development environment actually intended for the creation of 3D games.

3.1 RAZER HYDRA

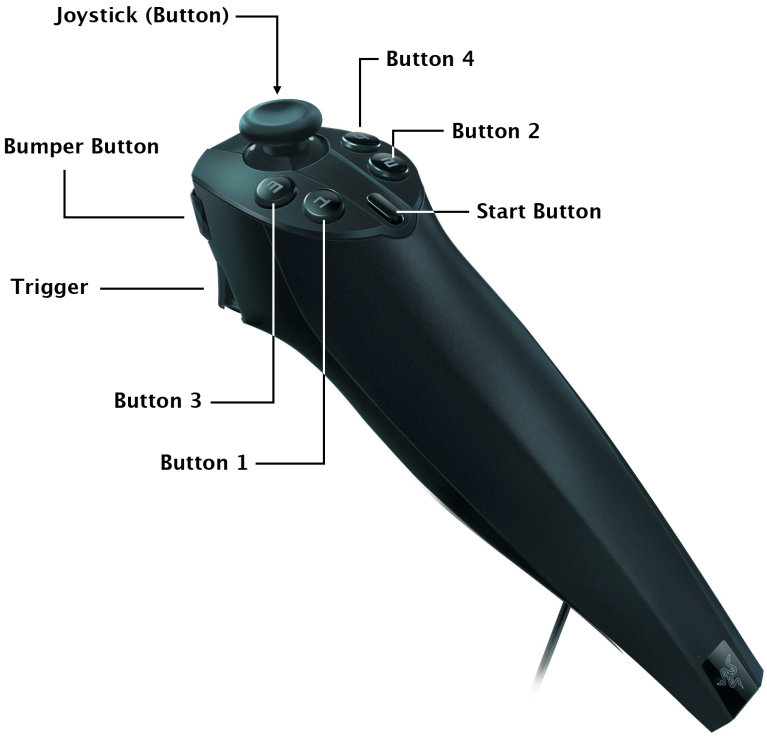
With the application being intended for use in elementary education, it was desirable to provide both an intuitive and engaging means of interaction. Traditional user interfaces and input devices such as mice and keyboards introduce a certain level of abstraction into the process of interacting with a software. This tends to render them inappropriate for young users though, since children are used to interact with real physical objects rather than abstract two-dimensional representations of the same [50]. In order to make up for this, a more tangible approach, using a new kind of motion controller, has been taken with the application at hand.

The Razer Hydra, officially announced on April 21st 2011 [38], "is the first mass-produced magnetic tracking system intended for the video game market" [11]. It consists of a base station and two handheld controllers, each yielding a total of eight buttons together with an analog joystick to be operated with the user's thumb (see Figure 9b for details). Both controllers are wired up to the base station shown in Figure 9a via a y-cable with a custom connector. The base station itself emits a weak magnetic field within which the position and orientation of the controllers are tracked with 3DOF each and an accuracy of up to one millimeter and degree [2]. Position and orientation computations are carried out on the device itself which yields an update frequency of 250Hz. Connected to the computer via USB, the data can be accessed through a special SDK provided by Sixense, the company which actually developed the Hydra [11].

Compared to other common systems based on optical tracking, like the Nintendo Wii[1], or the PlayStation Move[2], the Hydra does not only offer a significant increase in tracking precision, but it also eliminates the need for a line of sight between the controllers and the base station [2].



(a) The base station.



(b) One of two handheld controllers.

Figure 9: The Razer Hydra Motion Controller. Source: *golem.de* (edited by the author) [30]

1 http://www.nintendo.com/wii
2 http://us.playstation.com/ps3/playstation-move/

3.2 OPENTRACKER

OpenTracker, as presented by Reitmayr and Schmalstieg in [49], serves as a connection between the Razer Hydra and the actual application implemented with Unity 3D, which is going to be subject of the next section. The framework is mainly intended for use in the context of virtual and augmented reality projects, which it is supposed to facilitate the customization of the tracking process in. By providing an independent middleware layer it enables communication between the system and a variety of different input sources, thus allowing for flexible tracker configurations.

Independent of the concrete tracking mechanism, the generated data is usually processed in similar ways. Originating in the actual input device, it is read-out by appropriate drivers, transformed into a format suitable for the receiving application and then delivered via some sort of transport mechanism. OpenTracker aims to abstract away the device-specific details of this processing chain by allowing developers to define custom data flow graphs via standard XML.

Said graph is made up of individual nodes, each of which represents a single computational step on the way from the initial generation of the tracking data to the point where it is available for use by the actual application. Individual nodes are connected via directed edges originating at the child and ending in the parent node. In order to allow for complex graph structures and, thus, complex calculations, nodes can have multiple input ports, each of which expecting a certain type of data. Moreover individual nodes can be children to multiple parents, which means that a single output port can be connected to more than one input port. Constructs like these are achieved via multiple references to the same node within the graph.

The edges connecting the individual nodes can be of three different types, the most common and straight forward of which is *event*. Along edges of this type data is actively sent from child to parent node as soon as it is available. Conversely parent nodes have to explicitly poll child nodes for data when connected via edges of type *event queue* and *time dependent*. In this case multiple events are stored at the child node and can be retrieved either by their number in the queue or by the timestamp they are bearing. Functionality of this kind might, for example, be useful for nodes performing a smoothing operation and, thus, needing to work on a set of data instead of single events.

When considering the data flow graph's nodes themselves, just as with the edges, three different types can be distinguished:

SOURCE NODES : This is where all tracking data first enters the OpenTracker system. Typically these nodes are wrappers around specific device drivers; they might, however, also represent bridges to complex software-based tracking systems. Moreover, source

nodes may emulate a tracking device via keyboard input, receive data from external sources via a network or simply provide uniform data over and over again for development and debugging purposes.

FILTER NODES : These nodes are responsible for carrying out the actual work in the OpenTracker framework. Based upon all the input values they receive from their child nodes, they update their own state and place their output value at their parents' disposal. Throughout the whole framework, there is a wide variety of different filter nodes, each offering different functionality. Some of the most widely used ones are transformation filters which simply apply geometric transformations on the values received from their children. Amongst many others, there are also noise and smoothing filters, conversion filters for translating one data type into another and merge filters taking multiple inputs and combining them into new data values.

SINK NODES : As the the counterpart to source nodes, the sink nodes' mere responsibility is to distribute data instead of receiving it. They reside at the top-level of the data flow graph and might be used to generate debugging output for development or send processed data over a network. Most importantly, however, sink nodes act as the link between OpenTracker itself and the application integrating it as a library.

As previously mentioned, OpenTracker provides developers with the possibility to define complex tracking configurations via simple XML files complying to a special DTD. On start-up, said XML files are

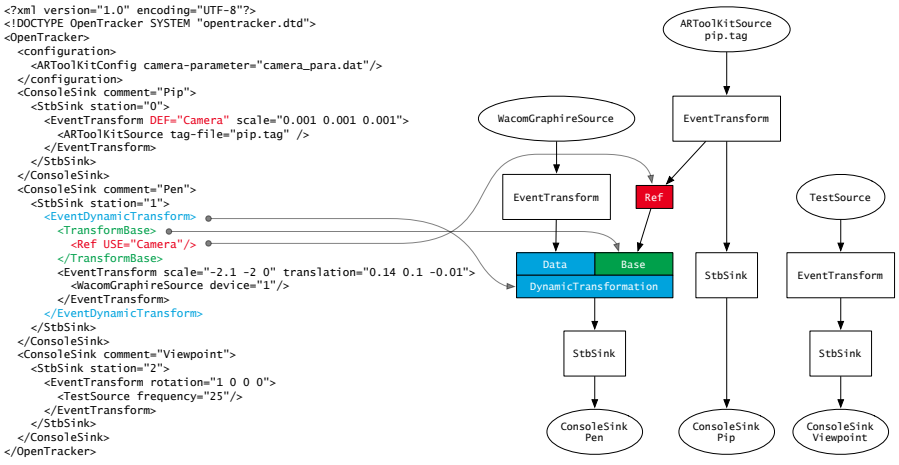


Figure 10: A sample tracking configuration using three different source and sink nodes. Some correspondences between XML elements and data flow graph nodes are exemplary highlighted. Source: *An Open Software Architecture for Virtual Reality Interaction* (recreated by the author) [49]

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE OpenTracker SYSTEM "opentracker.dtd">
<OpenTracker>
  <configuration>
    <HydraConfig base="0"/>
      <LogConfig active="on" stderr="on" file="on" filename="log.txt"
                 level="info" verbose="std"/>
  </configuration>
  <UnitySink name="controller0">
    <HydraSource controller="0"/>
  </UnitySink>
  <UnitySink name="controller1">
    <HydraSource controller="1"/>
  </UnitySink>
</OpenTracker>
```

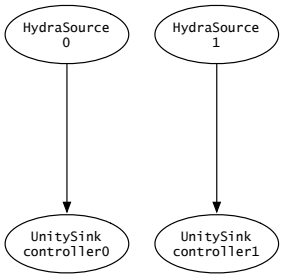


Figure 11: The OpenTracker configuration used in the project is extremely simple as it does not even require any filter nodes.

parsed, data flow graphs are created from it and the classes representing the individual nodes are dynamically instantiated. Developers do, thus, not have to care about the specifics of OpenTracker's implementation, but can conveniently construct tracking setups merely in standard XML. Figure 10 illustrates a sample configuration, opposing the contents of the XML file to the resulting data flow graph.

The aforementioned convenience is, of course, only given to the fullest degree, if all involved components, both soft- and hardware, are already fully integrated into OpenTracker. Otherwise, new source or sink nodes have to be implemented, depending on whether a new input device is used or the application is implemented by means of an unsupported application framework. In order to integrate OpenTracker into the application presented within this thesis a special sink node for communication with Unity 3D was required as well as a custom source node for the Razer Hydra. The actual tracking configuration used, though, is rather simple, as can be seen in Figure 11. In the context at hand OpenTracker was used as a mere adapter between input device and application framework to all intents and purposes.

3.3 UNITY3D

Most effort put into the software presented in this thesis was carried out in Unity 3D, an "integrated authoring tool for creation of 3D videogames" [8]. Over the past few years, Unity has gained popularity especially within the independent game developer community. This is likely due to the fact that the software is offered, amongst a variety of others, under a free license, which limitlessly supports all basic features. The free availability does, of course, not only render it an attractive tool for independent games, but also for research projects like the one at hand.

Apart from the financial benefits, the fact that projects developed in Unity can be deployed on multiple platforms without the need for any further adaptions represents an additional benefit [9]. These platforms include the three major desktop operating systems Windows,

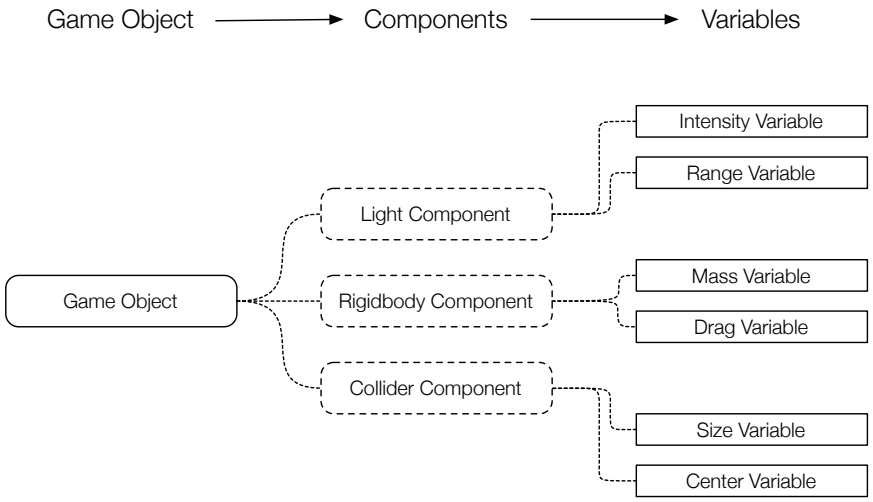


Figure 12: The relationships between `GameObjects`, `Components` and variables in Unity 3D. Source: *Unity 3.x Game Development Essentials* (recreated by the author) [19]

OSX and Linux, as well as the mobile platforms iOS and Android, various game consoles and a special web player for online deployment [8].

Unity provides a 3D rendering engine, which supports both DirectX and OpenGL and enables developers to create sophisticated graphics featuring "animated meshes, particle systems, advanced lighting and shadows". Moreover, the Nvidia (previously Ageia) PhysX engine is part of the package, providing support for realtime physics simulation including collisions, forces and joints between objects [57].

While both a graphics and physics engine are supplied, there are hardly any means for the creation of actual 3D content. Basically, Unity itself merely provides an empty three-dimensional space that can be populated with models created in dedicated third-party applications. Said models, just like all other things, are represented as `GameObjects` within Unity and can, in the matter of functionality and appearance, be enhanced by adding individual `Components` to them. A `Component` itself has variables, which can be used to control its behavior and, thus, the effect it has on the `GameObject` it is attached to (Figure 12 illustrates the concept). While Unity offers a multitude of different components out of the box, it is also possible to add special behavior to a `GameObject` via custom scripts.

In the following, the core concepts of Unity are going to be elaborated on in a little more detail [19].

ASSETS This is what Unity projects are essentially made of. Everything from 3D models over textures stored as image files to sound files for audio effects to scripts for defining behavior is referred to as an asset. Consequently all files created and used

throughout a unity project are stored within a folder called `Assets`.

SCENES In Unity, a scene can be thought of as the three-dimensional space that all objects exist in. The basic idea is that every level within a game gets its own scene, which positively affects load-times as contents can be incrementally loaded instead of all at once. It is, however, possible to create entire projects within a single scene, by dynamically loading and removing additional content via scripted code. This approach has also been taken when developing the application presented in the thesis at hand.

GAMEOBJECTS Any object, from lights over 3D models to particle systems, placed on the stage set up by the scene is automatically treated as a `GameObject`. As already mentioned above, a `GameObject` can be enhanced in terms of functionality and appearance by attaching `Components` to them. The one `Component` that every `GameObject` bears from the very beginning is the `Transform` component, which specifies an object's position, rotation and scale within the scene. These values are all given in x-, y- and z-coordinates and can be altered in the Unity editor or from within any given script.

`GameObjects` do not necessarily have to exist as separate entities but can also be grouped together in hierarchies. In such a hierarchy, there is always one parent object, the `Transform` component of which determines the position, rotation and scale of the entire group. Hence, the `Transforms` of all child components have to be considered as relative to the parent's `Transform`.

COMPONENTS `Components` can be thought of as building blocks that, step-by-step, lend functionality to a `GameObject`. They can be as simple as a `MeshFilter` storing the geometric data of a 3D model and as complex as a `Rigidbody` defining and controlling the physical behavior of an object. Generally speaking, everything that is part of a scene in Unity is inscribed into `Components`. Even a light or a camera is merely a `GameObject` with the appropriate `Component` attached to it.

By default, Unity comes with a broad range of prefabricated `Components`. If additional functionality is required, though, it can be achieved via custom scripts, which are also considered as such.

SCRIPTS While a lot of functionality is already made available via ready-made `Components` the actual logic of a game is created by writing custom scripts and attaching them to `GameObjects`. Unity supports three different scripting languages, namely C#, JavaScript and Boo (a Python based language), all of which are fully interoperable and can, thus, be used alongside in the same

project. The general performance deficit of scripting languages is made up for by compiling the scripts into native code, yielding an efficiency close to more low level languages such as C++ [57].

In order to write scripts in Unity, it is not necessary to have profound knowledge of the implementation and inner workings of the system itself. An extensive API for all of the three languages is provided allowing easy access to all required functionality. Most of the logic in a Unity project is coded in subclasses of `Monobehaviour`, which provides a set of functions automatically called by the system when the script is attached to a `GameObject`. The most important of these functions are `Start`, which is called when the object is first initialized in the current scene and `Update`, which is called once "in every frame that the game runs". Moreover there are functions for handling user input, reacting to in-game events such as collisions or triggers as well as drawing and managing a graphical user interface.

PREFABS Prefabs are Unity's way of reusing content both within and across project boundaries. In the game development process the need to have multiple instances of the same object is likely to occur frequently. In order to avoid having to recreate the same object over and over again, Unity simply allows developers to store an object with all of its components and its current configuration as a prefab. These prefabs can then easily be instantiated at any time, even in projects different from the one they have actually been created in. The newly created instances are, while initially equally structured, entirely independent and can be individually modified.

4 DIDACTICAL BACKGROUND

With the application developed in the course of this thesis intended primarily for elementary education, the question arises in which way pupils can benefit from such a tool. This question is especially pressing, as the software, in the version at hand, lacks many of the features found in most of the solutions described in Chapter 2. It is not particularly suited for accurate and formal constructions, does not distinguish between constrained and unconstrained objects and is not able to remember and redo any construction histories. While all of these features might very well be added in future versions of the software, it is still believed to be beneficial in its current state.

Various studies have shown that geometry education significantly benefits the development of spatial abilities, as do virtual reality applications [28] and even video games [17]. With its geometric objects, its 6DOF motion tracking input device and a general tendency towards experimentation and playfulness, the application presented herein unites aspects of all three of these domains. Thus, in its current form the software can be thought of much more as a means for children to develop and improve spatial abilities than as a fully fledged tool for geometry education.

In order to provide some background information, Section 4.1 is going to elaborate on what spatial abilities actually encompass and how they can be trained and improved. Section 4.2 will then briefly introduce the constructivist learning theory, a theory promoting the principles of learning-by-doing, which is the fundamental educational concept behind the developed software.

4.1 SPATIAL ABILITIES

From its early beginnings in the late 19th century up to now, research in the field of human spatial abilities has traversed many stages. Starting off with the recognition of spatial abilities as "separate from general intelligence", the research focus as well as the widely agreed upon central insights changed significantly. Different subcategories of spatial abilities have been identified in a multitude of publications and their temporal development in children and adolescents as well as gender-specific differences have been analyzed. Recent research especially deals with "the impact of technology on measurement, examination, and improvement of spatial ability" [40].

Providing a brief introduction into the subject matter, the following section is going to elaborate on the current understanding of

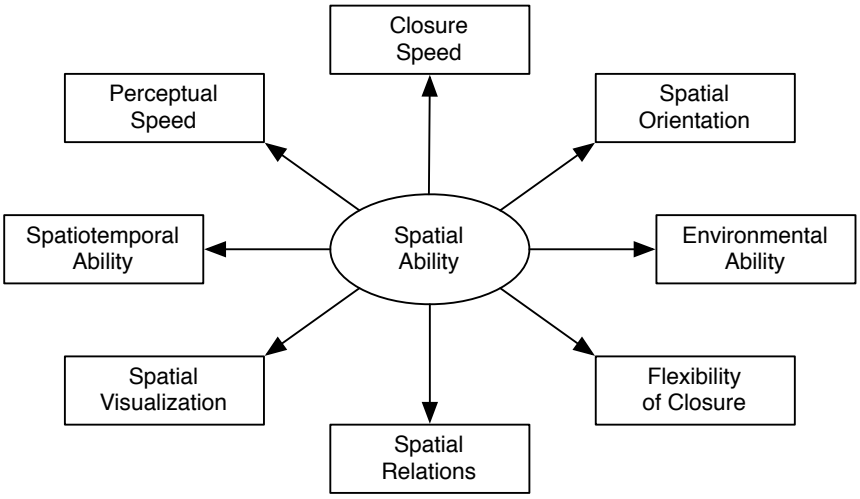


Figure 13: A model of general spatial ability including frequently overlooked factors such as Spatiotemporal and Environmental Ability. Source: *On the Development and Measurement of Spatial Ability* (recreated by the author) [59]

what spatial abilities are. Being relevant to the work carried out in the course of this thesis, Section 4.1.2 will, in turn, deal with ways to improve them.

4.1.1 *What is Spatial Ability...*

David F. Lohman, one of the most cited researchers on the topic, attempts to give a compendious definition of the term as follows [36].

> *Spatial ability may be defined as the ability to generate, retain, retrieve, and transform well-structured visual images.*

As already mentioned above, though, human spatial ability and especially the task of identifying the different factors constituting it has been dealt with by many researchers in the course of time. It is therefore hardly surprising that a plethora of different views and opinions have developed and full mutual consent is, at best, rare. While spatial ability is commonly acknowledged as an integral part of general intelligence, there is an active dispute about *how many* different factors it is comprised of and *how* these factors might best be described. Opinions range between two and ten concerning the number of factors and, while some of these factors are named differently but described in the same way, others are identically named but described differently depending on the individual studies [59].

Based on this variety of results produced by different researchers, Yilmaz tries to construct a "model of general spatial ability". The constituting factors of this model, as illustrated in Figure 13, shall briefly be described in the following [59].

SPATIAL ORIENTATION The ability to imagine an object's appearance from a different point of view [59]. Strong and Smith emphasize the *orientation* part of the definition by comparing the skill to that of a diver, who, while twisting and turning, always has an accurate mental image of the surrounding scenery and knows where the water is [55].

SPATIAL RELATIONS Closely related to Spatial Orientation, this is considered the ability to mentally rotate a pictured object in order to tell whether it is equivalent to another object or not. According to the given definition, Spatial Relations are also referred to as *Speeded Mental Rotation* [59].

SPATIAL VISUALIZATION Being the very factor occurring in the majority of all publications on the topic, Spatial Visualization is also the most general one. It is considered "the ability to imagine manipulating, rotating, twisting, or inverting objects without reference to one's self" [59]. It is worth noting that the focus herein is on the imaginative nature of the manipulation. A practical example falling into the domain of Spatial Visualization would, thus, be the ability to draw a sketch of one's house when viewed from above without any further reference [55].

PERCEPTUAL SPEED The ability to quickly identify a particular visual stimulus within a variety of other, potentially distracting material [59]. Typical tasks, which Perceptual Speed is beneficial in, is the comparing of figures and symbols, as, for instance, when comparing two large numbers for equality [16].

CLOSURE SPEED The ability to relate a single or combine many seemingly independent visual stimuli into a mental image of a well-known object. This way an object can be identified within an image even if it is partially obscured, out of focus or otherwise distorted, because of a well established representation of it in long-term memory [16].

FLEXIBILITY OF CLOSURE This factor describes the ability to identify a given object or visual stimulus within a bigger and more complex pattern. The factor is also referred to as *Field Independence* or *Disembedding*, which is due to the fact that tasks falling into this category require one to mentally extract objects from their surroundings [59]. Contrary to Closure Speed this ability involves a known stimulus, which means people know what they are looking for in an image instead of having to associate the contents with visual memories [16].

SPATIOTEMPORAL ABILITY Although neglected in many works on spatial ability, several researchers reason that there is an inherent difference between mentally processing dynamic and static

displays. Therefore the factor of Spatiotemporal Ability was introduced, which "involves judgments about and responses to" moving visual stimuli. Tasks falling into this category include, for example, "time of arrival" estimations as well as judgements about the location of moving objects relative to static ones [21].

ENVIRONMENTAL ABILITY This ability "requires integrating spatial information about natural and artificial objects and surfaces in an individual's surroundings". Much like the aforementioned Spatiotemporal Ability, Environmental Ability is frequently ignored by a multitude of researchers, although it is considered essential for tasks such as "way-finding and navigation" by others [59].

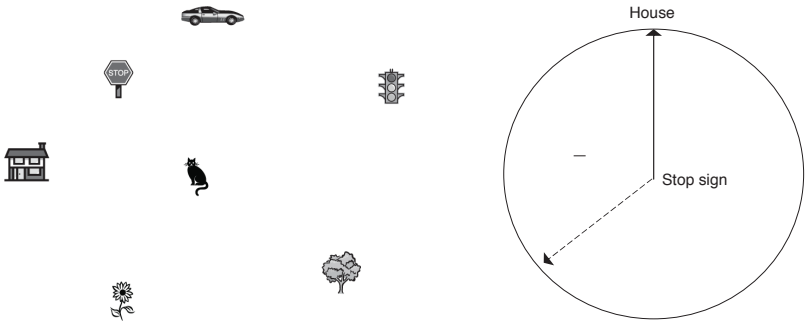
All of the factors described above are more or less pronounced in different human beings and there have been a variety of different methods developed to test people for them. Figure 14 provides an illustration of tests for different categories of spatial ability, which should also facilitate understanding of the sometimes rather abstract textual description of the same.

Apart from general differences between individuals, there is a common agreement on the fact that gender-specific disparities can be identified as well. Despite this understanding, though, researchers tend to disagree on the specifics of said disparities much as they do on the actual factors constituting spatial ability itself. Similarly, research on the topic of how and when spatial ability develops in children does not yield any uniform and commonly agreed upon results. In very general terms, it is somewhat "difficult to have a comprehensive understanding of the subject" anyway, considering the plethora of research undertaken in the field and the frequently contradictory results produced [59].

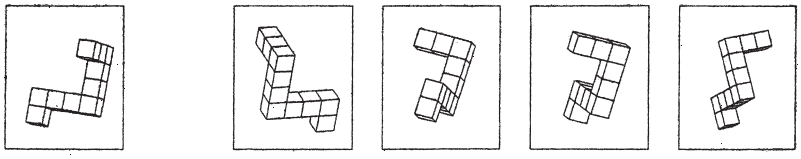
4.1.2 *... and how can it be improved?*

While there is general disagreement about its concrete definition, structure and development, the actual importance of spatial ability is hardly disputed in scientific research. People with well-developed skills in this area are frequently talented in creative thinking as well, which is not only beneficial when it comes to artistic endeavors but also in mathematics and science. Many well-known natural scientists and inventors, such as Albert Einstein, Michael Faraday and James Watts, have themselves reported that their high levels of spatial ability were of significant importance to their accomplishments [36].

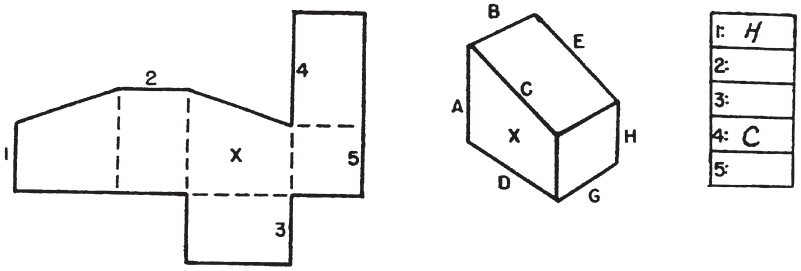
Apart from the benefits in terms of creative thinking, studies have shown that spatial abilities also play an important role in information seeking tasks, which is relevant nowadays more than ever. With the rapid technological advancements in recent years, information, to-



(a) Imagine you are at the stop sign and facing the house. Point to the traffic light.

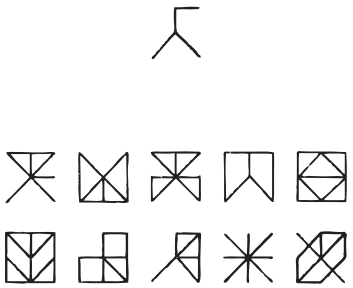


(b) Which of the righthand images portrays the same object as in the left-hand image?



(c) Mentally fold the blueprint into the object and match numbers to letters accordingly.

(d) Which commonly known object is shown in this picture? (Answer: Hammerhead)



(e) Which of the patterns at the bottom does the model at the top appear in?

Figure 14: Tests for Spatial Orientation (a), Spatial Relations (b), Spatial Visualization (c), Closure Speed (d) and Flexibility of Closure (e). Sources: *(a)*: [22], *(b, c, d)*: [58], *(e)*: [16] (all modified by the author)

day, is not only ubiquitously available but also presented in a wide variety of ways. Studies in the field of human computer interaction have shown that this can, in turn, be challenging for people with low levels of spatial ability. Information retrieval from a website or the completion of tasks within a complex user interface, for example, are usually accomplished significantly faster and with less effort by users with well-developed spatial ability [15].

Considering these apparent benefits, it is hardly surprising that a growing number of research efforts are nowadays put into how spatial ability can be improved or trained. While there is no overall agreement on whether a targeted improvement is possible, a variety of studies lead to believe that it *is* in fact, provided the appropriate materials and strategies are applied. Considering the definitions of spatial ability and its constituting factors given above, Olkun, for example, detects an "obvious connection between spatial thinking and transformational geometry". He furthermore hypothesizes that "work with the latter would improve skills in the former" [42].

Indeed, as noted by Kaufmann in [27], a surprisingly large number of studies on the improvement of spatial ability have been conducted in the context of geometry education and most of them yielded positive results. Thus, participants performed better on tests, such as the ones portrayed in Figure 14, after completing a training program on the topic of graphics or geometry than they did beforehand. While Kaufmann does point out that, considering the studies compared within his work, there might be a publication bias towards positive results, this can still be regarded as an indication for the beneficial effects of geometry education on spatial thinking.

It is not only traditional geometry education, though, that positively influences spatial abilities. A study conducted by Rafi et al. showed that also training programs utilizing web-based virtual environment setups significantly benefitted the test persons. Participants in the study worked with a software which allowed them to, for example, rotate and manipulate virtual objects so as to match them to given orthographic views of the same. Subsequent comparisons with a test group participating in conventional geometry classes showed that training with the virtual environment software was even more effective in terms of improving spatial ability. [47]. A positive influence of this type of software is also confirmed by Kaufmann, who included a similar study into his comparison [27].

Apart from software specifically designed as a training tool for spatial thinking, several studies concluded that even video games, actually targeted at mere entertainment, can have positive effects. Subrahmanyam and Greenfield, for example, showed that dynamic spatial skills in children could significantly be improved by simply playing the arcade video game *Marble Madness*[1] for only 2 hours and 15 min-

1 http://en.wikipedia.org/wiki/Marble_Madness

utes [56]. Feng et al., though, reached a somewhat different conclusion. Their study involved a group of students playing *Balance*[2], a 3D puzzle game similar to *Marble Madness* to some degree, for 10 hours, which did not yield any noticeable improvements. They did however detect significant improvement in a test group playing the game *Medal of Honor: Pacific Assault*[3], concluding that action games *do very well* bare the potential to positively affect a player's spatial abilities [17].

Considering the two studies' somewhat ambiguous results, it is worth pointing out that there were significant differences in their design. While Subrahmanyam and Greenfield examined children at the average age of 11, Feng et al. conducted their study with undergraduate students between 19 and 30 years of age. Moreover the tests applied for measuring the subjects' spatial skills prior to and after the training phase did not match and were targeted at different factors of spatial ability. It is, thus, not possible to fully compare the studies and it stands to believe that both have a valid point in claiming that video games positively influence spatial thinking.

One thing that is common to both specifically educational software and video games is that they all require active user participation. This *hands-on* experience might be particularly beneficial in terms of training efficiency, which has also been suspected by Kaufmann in his comparison of studies on the effects of geometry education. He concluded that "especially trainings involving hand-drawing or sketching and physical modeling" positively affected spatial abilities in test persons [27]. Said hands-on experience, active engagement, experimentation and exploration are also endorsed by the constructivist learning theory, which is going to be described in the following section.

4.2 CONSTRUCTIVIST LEARNING THEORY

Constructivism, next to Behaviorism and Cognitivism, is one of the three major learning theories that have developed throughout the course of history. Behaviorism, the earliest of these theories, promoted an idea of learning as a simple stimulus-response scheme. It, thus, regarded the human mind as a "black-box" merely reacting to external inputs and ignored the "possibility of thought processes occurring in the mind". Due to the many limitations this approach imposed on the understanding of learning, it was replaced by the new theory of Cognitivism in the 1920's. According to representatives of this theory, the learning process was much more complex and involved the constant creation, re-evaluation and manipulation of "internal knowledge structures". Researchers also acknowledged that information is

2 http://en.wikipedia.org/wiki/Ballance
3 http://en.wikipedia.org/wiki/Medal_of_Honor:_Pacific_Assault

processed and, in turn, remembered differently depending on many things, such as its meaningfulness, organization, or context [39].

Also in the 1920's, Jean Piaget, a Swiss developmental psychologist, laid the foundations of what was to become the third major learning theory in his doctoral thesis. Although the thesis was located in the field of biology and dealt with the way mollusks adapted themselves to new environments and, in the process, modified the environment itself, he later referred back to it when explaining the basic idea behind Constructivism. Much like the mollusks did with their environment, he claims, the human mind adapts to new knowledge and modifies it according to its already established mental structure and way of thinking [46]. Put in Piaget's own, frequently cited words [44]:

> *Intelligence organizes the world by organizing itself.*

The most important concept behind Constructivism is that learners are by no means passive consumers of information, as they are pictured in behavioral theory. On the contrary, they actively construct knowledge, as the name implies. They attempt to include new information into their existing view of the world so that everything still makes sense and they potentially alter both along the way. The actual act of learning is the central element of the theory. It is not only a deterministic stimulus-response mechanism but a process that requires the learner to actively get involved with the subject matter. Problems need to be thoroughly explored and compared to the established knowledge base in order to be able to construct new conceptual structures based upon the experiences made. Therefore, the term *learning* in Constructivism rather "emphasizes the process and not the product" [41].

In addition to the general idea described above, Mantovani elaborates on three key concepts of Constructivism and establishes a connection between them and the educational use of virtual environments. Regarding the nature of the developed software, these concepts and considerations are also relevant in the context of this work, which is why they shall briefly be presented in the following [37].

CONSTRUCTIONISM While some consider it an independent theory that merely shares "Constructivism's connotation of learning" [43], Mantovani treats it as one of the key concepts of the same. The basic proposition of Constructionism is that knowledge is constructed "from physical interaction with objects in the world". It is therefore integral that learners actively engage with the material, manipulate objects and build new artifacts in order to fully understand the subject matter.

Although real physical interaction is, of course, not possible in virtual environments, an increasing level of immersion can yield a close-to-real experience and significantly aid the learning process.

EXPLORATORY LEARNING It is believed that students acquire knowledge better when allowed to freely explore new contents and experiment without any given directives. By having to figure things out themselves, they invest more effort into the "construction of conceptual models that are both consistent with what they already understand and with the new content presented". The process of actively coming to a conclusion about something is, moreover, significantly more rewarding and motivating than simply being told other people's views.

Virtual environments bare the potential to provide students with exactly the playground they need for the currently covered material. When actual physical material can not be supplied, for whatever reason, simulation represents a means to create an appropriate context, resulting in more effective and meaningful learning.

COLLABORATION Another important constructivist statement is that working together in groups, engaging in discussions and reaching a consensus about a topic is greatly beneficial for the quality and efficiency of learning. Social interaction does not only bring up and confront learners with different views of the same subject or problem but it also requires reasoning and arguing for one's personal ideas. This, of course, incites profound engagement with the material and potentially leads to an improved learning outcome.

While collocated collaboration hardly benefits from the use of virtual reality setups, multi-user environments enable social interaction with remote participants. By the use of avatars, students can engage in collaborative activities, have verbal discussions and create a common understanding of a subject even if geographically separated.

Apart from the collaborative aspect, all of the above-mentioned concepts also support the educational value of the software developed in the course of this thesis. While collaboration functionality might very well be added in future versions, the following part of this work is going to provide an in-depth presentation of the application in its current state.

Part II

PRACTICE

5 DESIGN

An important aspect in the development process of the software presented in this thesis was the definition of how the users would interact with the system. While this should, of course, generally be a matter of great importance for every newly created software, it was especially important in the case at hand. For one part the application was to be controlled via a completely new interaction device, for which hardly any examples of usage existed before. For another part the target audience were children in elementary education. Thus interaction had to be as simple and easy to grasp as possible, while still granting full access to the system's rich functionality.

The following chapter is going to elaborate on the ideas and considerations that went into the interaction design. First of all the software's overall appearance and basic usage is described. Later sections are going to focus on the very essential part of object creation and manipulation. The chapter is going to be concluded by an in depth description of the graphical user interface.

5.1 GENERAL APPEARANCE AND 3D INTERACTION

The application deals with the creation and manipulation of geometric objects in 3D-space. As such it provides a virtual workspace that resembles those of most common 3D computer graphics tools. This is where objects are created and manipulated and it therefore is the software's main interaction area. In its initial state, as illustrated in Figure 15, the workspace only contains a non-editable two-dimensional grid aligned with the xz coordinate plane and six directional light sources. The sources itself are invisible but provide constant and uniform illumination throughout the whole scene, while the grid is supposed to facilitate orientation within the virtual space. There are also three red lines that are styled to create the impression of laser beams. They are perpendicular to and intersect each other in one single point, thus creating a three-dimensional crosshair used for interaction with the 3D scene. Finally the workspace contains a camera providing a view of the scene and actually rendering it. Both the camera and the crosshair are controlled by the user, which is going to be elaborated on in the following.

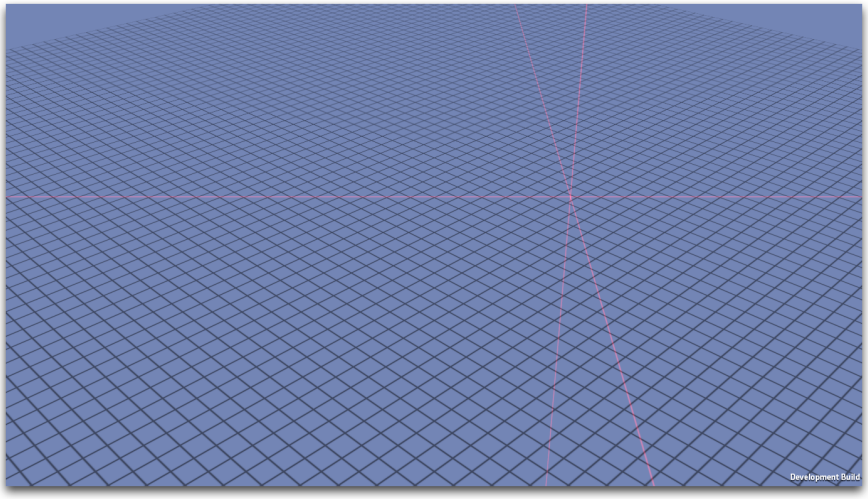


Figure 15: The main interaction area of the application as shown right after start up.

5.1.1 *Crosshair Control*

The crosshair is made up of three lines, arranged perpendicularly to each other and intersecting in its center. Its position in space (i.e. the positions of the intersection point) is bound to the position of the right-hand controller. Thus, the controller's motion is directly mirrored by the crosshair. Controller rotation does not affect the crosshair at all.

Calibration

In order to decouple crosshair position from the user's distance to the controller's base station a calibration mechanism is provided. At start-up the user is asked to pull the left- and right-hand controllers' triggers one after another. At these trigger events both the controllers' positions and orientations are recorded and subsequently used to normalize the respective values. The controllers can also be re-calibrated at any time by clicking and holding the joystick-button, which starts

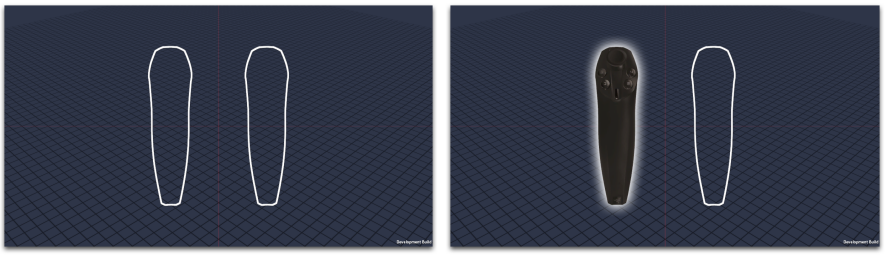


Figure 16: During calibration the user is prompted to subsequently press the left- and righthand controller's trigger.

the described routine over. Figure 16 depicts the calibration screen as seen by the user.

General Interaction

Apart from triggering the calibration routine, the joystick button serves an additional purpose. Simply clicking it results in the crosshair being reset to the center of the screen, regardless of current controller position. The possibility for resetting crosshair position in this manner is vital, since users are likely to, more or less slightly, move away from their initial position. This, in turn, also moves the crosshair in virtual space, which is most often unintended behavior; especially if, by the user's motion, the crosshair gets completely out of sight. The reset functionality allows the user to move freely in front of the screen and easily recover crosshair control afterwards.

Apart from their tendency to not take a fixed position, users naturally introduce another problem that has to be made up for. The controller's range of motion is limited to the user's arm length. Thus there are vast areas of virtual space that simply cannot be reached without assistance. This is a common problem in virtual reality applications and there are various ways to address it, like e.g. arm-extension techniques [45], ray-casting or combinations of both [12]. While these approaches might or might not work in a virtual reality setting they did not seem appropriate for the application at hand. Since, regarding user input, the level of abstraction is significantly higher and closer to standard input methods, an approach similar to mouse interaction was chosen. When using a computer mouse one frequently interrupts his motion to pick the mouse up, reposition it and continue the motion afterwards. By doing so one is able to increase the range of the mouse pointer, which would otherwise be limited by the range of motion of fingers and wrist. The same concept is applied in the presented software. Whenever the user holds down the start button on the right-hand controller, motion events for this controller are ignored by the system. This results in the crosshair's position being fixed until the start button is released. Thus, if a user reaches the limits of his natural range of motion he can temporarily freeze the crosshair, move the controller into a more comfortable position and step-by-step work his way towards the target. Compared to the aforementioned techniques, this potentially requires a higher amount of physical movement. It is however deemed to be a way of interaction that is more intuitive and easier to control.

Intuitivity in general is an important aspect in the design of the application. The user is supposed to feel like he is directly interacting with the virtual world as much as possible. It is therefore important that controller motion directly maps to crosshair motion. This is why the crosshair's axes are not aligned to the workspace's coordinate axes. Since the camera can be moved and rotated, which is going

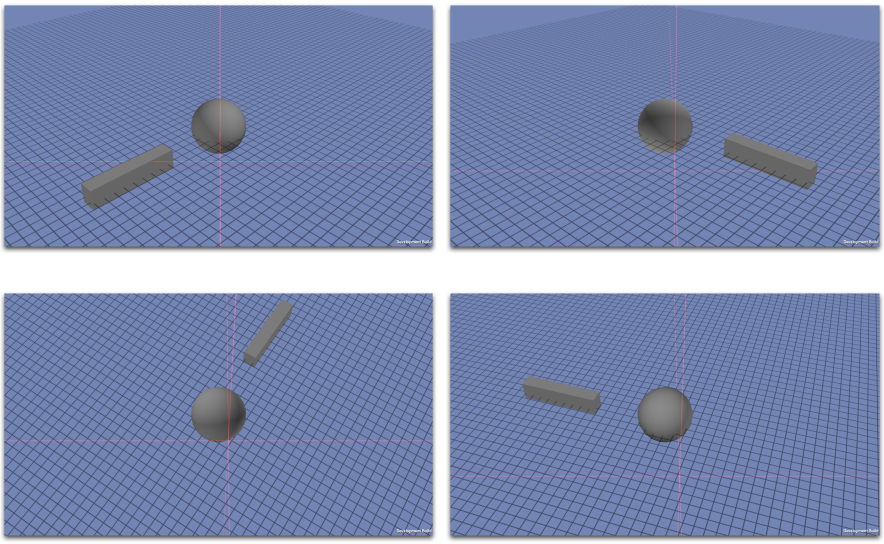


Figure 17: In order to provide for intuitive interaction, the crosshair's x-axis is always parallel to the viewport and thus independent from the viewing angle.

to be explained in the next section, the crosshair is instead always aligned to the camera. As the camera is rotated, the crosshair is always rotated along, so that its x-axis is parallel to the viewport (see Figure 17). This way a left-right motion of the controller is always directly reflected on the screen. The crosshair's y-axis is at all times aligned with the virtual world's y-axis, so that both xz-planes are parallel to each other. Up-down and forward-backward motions of the controller do therefore *not* move the crosshair independently from the viewing angle. Although this behavior somewhat loosens the direct mapping of controller to crosshair motion, intuitive navigation is maintained.

5.1.2 *Camera Control*

The camera represents the user's view into the virtual world. In order to view the scene from different angles and in different levels of detail, the camera's position and orientation has to be fully controllable. The presented software provides camera control via the left-hand controller. It supports the three camera movement techniques common to 3D computer graphics tools and also adds some useful features in order to facilitate navigation. Note that the naming convention for said techniques used herein is that of Autodesk's Maya [7]. The same functionality might thus be named differently in other applications.

TRACKING usually means left-right and up-down movements. In the case at hand this behavior is slightly modified. While the left-right movement is maintained, the up-down movement is re-

placed by a translation along the crosshair's z-axis. So the camera is moved across the world's xz-plane, much like in a first-person view video game. This behavior proved to be more intuitive for joystick navigation.

DOLLYING denotes a motion along the camera's z-axis, resulting in a zoom-like[1] effect. It is achieved by holding the bumper button pressed, while pushing the joystick forward to dolly-in and pulling it back to dolly-out.

TUMBLING changes the angle of view on the currently centered area of the scene. It is not a mere rotation but also a translation in order to keep the camera focused on the same area. Contrary to the previously described tracking behavior this is opposite to the way an angle-of-view change would be implemented in a video game. In the context of a dynamic geometry software however, the approach taken is more practicable.

Tumbling is done by rotating the left-hand controller, while holding down the trigger. The camera movement is inversely mapped to the controller rotation. That is, if the controller is rotated downwards, the camera rotates downwards but moves upwards. This way navigation gets a natural feel, much as if the scene was mounted on a free floating globe. One might argue about the practice of only allowing camera tumbling when holding down the trigger. A threshold for the action however has to be introduced, in order to avoid unintended tumbling, which would definitely occur otherwise. Tests have shown, that having to press an additional button outmatches range or speed of motion thresholds.

Apart from the three described camera movement techniques, the application features some convenience methods for easier navigation. One of them is the ability to easily define a focus point for the camera.

The camera's focus point specifies the point, which the camera rotates around when tumbling. It thus lies on the z-axis of the camera's local coordinate system. When camera tracking occurs the focus point is moved along. Hence it might be cumbersome to focus on a special point or object and center it on the screen. To make up for this handicap it is possible to select any object in the scene with the crosshair and focus the camera on it by pressing the joystick button. This results in the focus point being set to the center of the object and the camera dollying-in on the object, so that it is framed by the screen.

Pressing the joystick button and holding it for a while resets the camera to its start-up position. It also resets the camera's focus point.

1 Actual zoom is not a camera move, but a change in the lens' focal length. It is not implemented in the software.

This way the user is able to quickly get from the outer regions of the scene back to the center. The feature might also be useful if the user loses orientation.

5.2 INTERACTING WITH OBJECTS

Being a dynamic geometry software, the creation and manipulation of geometric objects is the application's core functionality. Therefore a lot of effort has been put into designing the workflow of handling objects. The following sections are going to explain the dynamics of object creation as well as the various ways existing objects can be modified.

5.2.1 *Object Creation*

There is a plethora of simple geometric shapes available to work with. Initially the user is presented with an empty workspace, so the objects have to be created first. Dealing with a dynamic geometry software it is a requirement to have a certain amount of control over the shapes' parameters. This direct influence on the parameters is only exerted during object creation. Though an existing object can very well be transformed by scale, translation and rotation operations, the initial parameters can no longer be changed.

Object creation is very straight forward. The user opens up the menu depicted in Figure 18, which contains all of the twelve available primitives, ranging from a simple cube or sphere to more complex shapes like torus and pipe, and chooses one of them. The crosshair is then positioned wherever the object is supposed to be created. The

Figure 18: Users can choose between twelve different objects. In the example shown, the torus object is currently highlighted.

actual construction is then started by pressing and holding the trigger on the right-hand controller and moving the crosshair. During this drag gesture the shape's parameters are constantly updated. Their values are based on the vector between the crosshair's position when the trigger was initially pressed and its position at update time. This way the newly created object is literally stretched from a single point to the desired shape and size. To finalize the construction process the trigger is released.

Most of the available objects can be created with a single drag gesture. This is because the three components of the distance vector described above can be translated into all the necessary parameters in a way that is intuitively intelligible by the user. Even rather complex objects can be created this way. When creating a torus, for example, the vector's x and z components are used to calculate the major radius. The minor radius is defined by the y component. This way the crosshair is situated directly on top of the object's surface throughout the whole creation process. So the dynamics of object creation reveal themselves to the user after only a few movements of the controller.

There are two objects though that require a slightly more complex construction process. Pipes and half-pipes are created taking two consecutive steps. In the first step the inner radius and height are defined, just like the torus' major and minor radii. Releasing the trigger does not complete the construction, however. It instead fixes the inner radius, while further controller movement results in further changes in height, but also in the definition of the outer radius. In order to finalize the creation the user needs to press the trigger a second time. Although this process is admittedly more complex than standard object creation, it is equally easy to grasp. The user is at all times presented with immediate feedback, providing sufficient guidance to complete the task.

5.2.2 *Object Manipulation*

Once a shape has been created it can be transformed and modified in a variety of ways and aspects. In order to work with an existing object, it has to be selected first. Doing so requires the user to move the crosshair into the shape, which results in its wireframe representation being laid over it (see Figure 19). Pressing the bumper button finally selects the object.

Once selected, different actions can be performed on an object. The most basic actions are the standard 3D object transformations. Note that translations and rotations can also be executed without explicitly selecting an object via the bumper button. They only require the crosshair to be within the shape. Performing transformation actions usually requires input from both controllers. The right-hand controller initiates and controls the transformation, whereas the left-

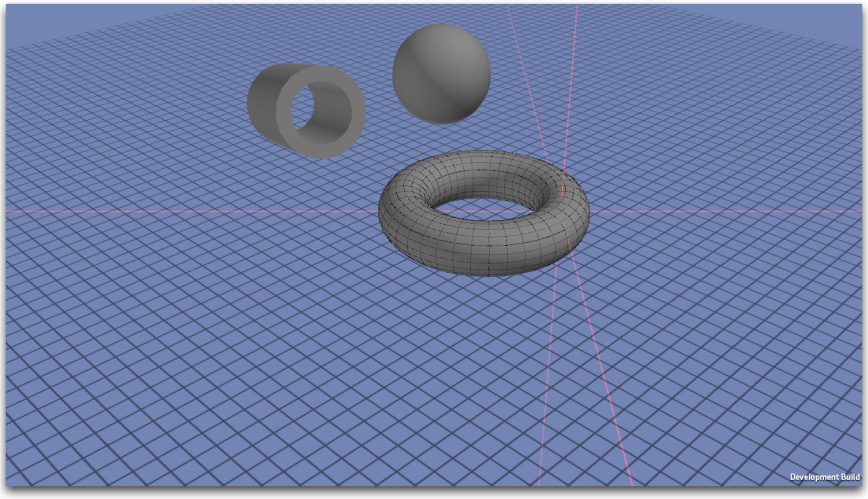


Figure 19: As the crosshair is moved into an object, its wireframe representation is overlaid in order to indicate that it can be selected.

hand controller, in a way, provides meta-input. By holding down its start button, or buttons one or two, for example, the transformation is limited to the *y*-, *x*- or *z*-axis[2] respectively. The left controller is also needed to define the transformation to be performed.

TRANSLATION: By holding down the right-hand controller's trigger and moving it, the object is moved along with the crosshair. The distance between crosshair position and object center is maintained in the process.

ROTATION: If the left-hand controller's bumper button is held down, pulling the trigger on the right-hand controller doesn't initiate a translation but a rotation. A change in orientation of the right-hand controller results in a rotation of the selected object.

Objects are by default rotated around their own center. If desired though, the user can press button three on the left-hand controller, which changes the rotation's pivot point from the object's center to the crosshair position. Pressing the button again resets rotation behavior to default.

SCALE: If both the left- and right-hand controllers' triggers are pulled, a scale transformation is set off. The scale value is determined by the change in distance between the two controllers. Thus, pulling the controllers apart results in increased scale along the respective axes. Moving them closer together, in turn, decreases the object's scale.

All of the described transformations can also be applied to multiple objects at the same time. In order to select more than one object the

2 Transformations always happen in the crosshair coordinate system. Doing them in the world coordinate system would potentially corrupt intuitive handling.

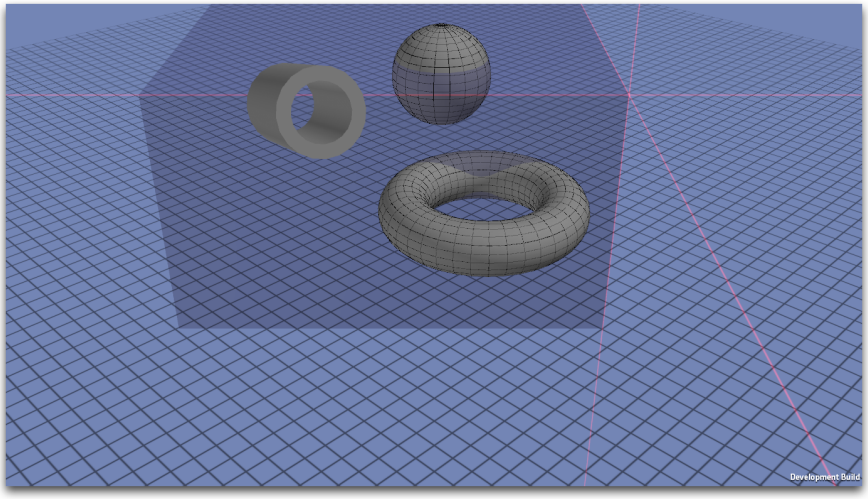


Figure 20: Multiple objects can be selected by spanning a selection box around them with the righthand controller.

user has two choices. One possibility is to consecutively select every object by positioning the crosshair within and pressing the bumper button. At times it might be more convenient though, to hold the bumper button and move the controller. This way a selection area (displayed as a transparent blue box) is spanned, adding every shape lying within to the selection.

Objects in a selection can not only be transformed together, they can also be grouped. In order to do so, one has to open a user interface menu (refer to Section 5.4 for details) and select the appropriate item. Once grouped, the objects are bound to each other. Whenever one of the group's objects is selected or transformed in any way, all of them are. Of course groups can be broken up again, removing all connections between objects.

If the user has no need for an existing object or group anymore, it can easily be deleted via the user interface. Likewise an object can be duplicated, resulting in an exact copy of the selected object. The duplicate's position is slightly offset from the original in order for the user to be able to separately select and move it.

Apart from the described actions there are still several more. Due to the fact, however, that most of the remaining actions impact an object's physical behavior, they are described in the following section.

5.3 PHYSICS

The software presented enhances its dynamic geometry aspect by adding physically correct behavior to the created shapes. By default, objects are static and do not interact with each other based on the laws of physics. They can be positioned freely, without falling down to the ground. They can also overlap or intersect without pushing

each other away. This gives the user a lot more flexibility and freedom in creating objects and arranging them in the scene.

Physics can be activated at any time via a user interface menu. As soon as physics are enabled, all the objects in the scene start interacting with each other and the environment. The are affected by gravity, collide with each other and bounce off the floor. The virtual floor is represented by the grid aligned with the world's *xz*-plane. Objects that have previously been grouped act as if they were one single object and do not collide with each other. When switching physics back off, the scene and all its objects are reset to the state they were in before activating physics.

With physics on, the user is no longer able to create any new objects. Existing ones, though, can still be moved and rotated. The crosshair, in this respect, acts like a virtual hand. By pulling the right-hand trigger, objects are literally picked up, which means a force is applied to them. Releasing the trigger lets loose of the object. It does not remove any given impulses, though. This again means that by releasing the trigger while moving the controller, the object can be thrown in the direction of the motion.

Every object is initially created with the same physical attributes. Some of these attributes can be changed via various user interface menus. The attributes are going to be described here, while the menus and user interface interaction in general are going to be elaborated on in Section 5.4.

An important property of objects is whether or not they are affected by gravity and collisions. Objects do so by default, but the user can define them to be static. Static objects do not move at all. They do not fall to the ground and do not react to collisions with other objects. This does not mean that other objects just pass right through them. They rather bounce off of them, while the static objects remain completely motionless.

Moreover, objects can have different materials assigned to them. Materials differ in bounciness, static and dynamic friction and in the way these values are combined when colliding with other objects. Providing a means of directly adjusting material properties has been avoided, in favor of well chosen presets. There is a total of five predefined materials, all of which have different physical characteristics. These materials can be assigned by one simple click, while manually tweaking properties requires a lot more user interaction. Considering the nature of user input, these actions are potentially hard to perform and would unnecessarily complicate matters.

Apart from internal physical properties the user is also able to assign a variety of colors to the shapes in the scene, as can be seen in Figure 21. Also, the objects' shading can be switched between diffuse and specular. They can be made transparent as well. The ability to alter an object's outside appearance does of course neither enhance

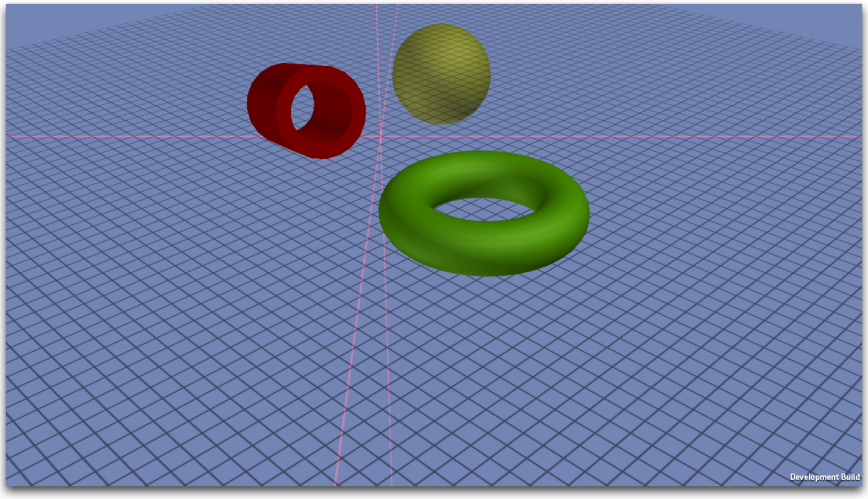


Figure 21: Users can assign different colors and shadings to objects. The depicted scene contains a dull red pipe, a shiny green torus and a transparent yellow sphere.

the geometry nor the physics aspect of the software. It is, nevertheless, deemed important regarding the application's target audience of children in elementary education.

5.4 GRAPHICAL USER INTERFACE

Developing a graphical user interface (GUI) for a system like the one at hand can be challenging. Despite its considerable versatility, the Hydra's main function is pointing at elements on a screen, when it comes to GUIs. This is why it also bears all the disadvantages inherent to remote pointing devices. For one thing the user's hand is prone to natural tremor, which occurs because the device is operated in mid-air without any additional stabilization. Also the limits of hand-eye coordination and motor precision are likely to cause problems. Especially with increasing distance to the screen, the user's pointing precision tends to become lower than pixel size. Precise navigation is therefore handicapped [31]. Gallo et al. also identified a lack in device precision as a source of inaccuracy in remote pointing [18]. In some respects this holds true for the Hydra as well. Although the controller is advertised to be able to deliver exact location and orientation data down to a millimeter and degree [38], practical tests yield different results to some extent. The controller does perform very well under optimum conditions. If it is operated in proximity to an electromagnetic source (e.g. TVs, displays, wireless and mobile phones), though, precision decreases significantly. Furthermore the distance from the hand-held controllers to the base station considerably influences signal accuracy.

With the described issues in mind, a user interface was designed that was explicitly targeted at dealing with potential pointing imprecisions. It is therefore almost exclusively made up of full-screen menus. These are bound to buttons on the right hand controller and can be faded in and out as required. Pressing the respective button once brings up the menu, pressing it again dismisses it.

Navigation within the GUI is done via the right-hand controller, with a circular cursor providing positional feedback to the user. The cursor is moved by moving and/or rotating the controller. A click is issued by pulling the controller's trigger.

A menu's items are laid out in a regular grid across the screen. They are, thus, large enough to be easily selected, even on smaller screens. As the cursor is hovered over them, visual feedback is provided by inverting their back- and foreground colors. Most items also show a textual description of the actions they trigger.

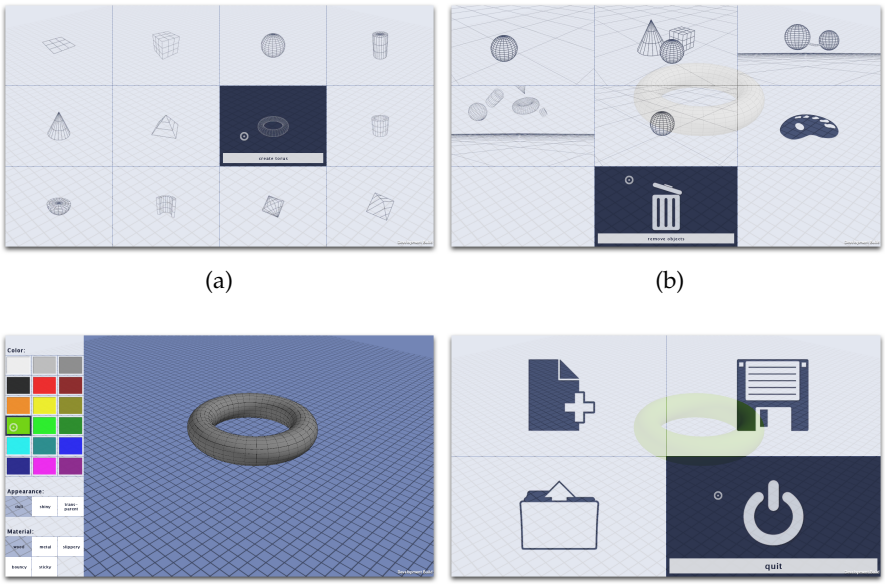
Concerning the items' actions in general, the icons they display are designed to be self-explanatory. Some functionalities, however, are rather hard to depict in a single image. Therefore, several items show an animation when hovered over. The menu item toggling physics, for example, shows a group of objects hovering over the ground plane in its passive state. As the cursor enters the item, the objects fall to the ground and collide with it. As a supplement to the textual description, this helps to quickly grasp the item's functionality. It is particularly useful when dealing with children, who might just be learning how to read.

Overall, there are five different menus, four of which can be opened via the right-hand controller's numbered buttons. The latter are depicted in Figure 22.

CREATING OBJECTS - BUTTON 1: The user is presented with all the primitives available to create. The respective items show wireframe representations of the shape.

MANIPULATING OBJECTS - BUTTON 2: This menu contains all the options available to manipulate objects, beyond simple transformations. In order to perform these actions on an object it has to be selected beforehand.

COLORS & MATERIALS - BUTTON 3: Of all available menus this is the only one that is *not* full-screen. Apart from assigning a physical material to an object it can also be used to change its outside appearance. It would hence be counterproductive to cover the entire screen with it. Instead, the menu is displayed at either the left or right side of the screen: which one, is decided upon every time the menu is opened. This way it is guaranteed that even selected objects lying at the edges of the screen are visible at all times.



(a) (b) (c) (d)

Figure 22: The menus for creating (a) and manipulating (b) objects, assigning colors and materials to them (c) and for managing files (d).

Note that this menu is also accessible via the previously described one. Providing a short-cut, however, seemed reasonable, especially because children are likely to make extensive use of the software's coloring functionality.

FILE - BUTTON 4: The usual file management options are contained herein. The user is presented with options to create a new scene, save the current one or load a scene previously saved. The application can also be quit via this menu.

Selecting the option to load an existing scene brings up the fifth menu.

The application allows the user to save and restore scenes as required. It does, however, *not* let the user influence where to put generated files in the local file system, or how to name them. This is mostly due to the specialized form of input with a two-handed motion controller. On the one hand, a virtual keyboard is not provided, since the handling with a remote pointing device would be cumbersome and error-prone. On the other hand, the use of a physical keyboard would be impractical, as it would involve putting away the controllers and picking them up again. For this reason text input has not been implemented at all.

Scenes are, therefore, automatically assigned a unique name when saved. They are furthermore all stored in one centralized location. Whenever trying to load a scene, the user is presented with a grid of tiles representing the respective scenes, as illustrated in Figure 23.

These tiles display both the time and date, the scene has last been saved, as well as a snapshot of the scene as a visual cue. If there are more scenes saved than fit on one screen, the user can browse through the multiple pages by flipping the controller to the left or right.

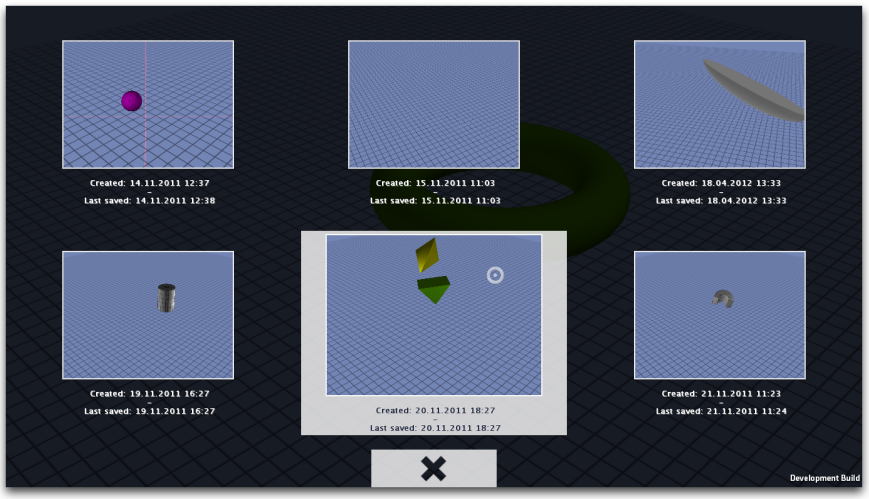


Figure 23: Previously stored scenes as presented to the user when trying to load one.

6 IMPLEMENTATION

After conceptualizing a first version of the design, the implementation process was started. Of course, as the implementation progressed, the design was revised, altered and adapted until it assumed the shape described in the previous chapter.

The application was developed for the Windows platform. Although Unity 3D, the main technology in use, would allow for deployment on various platforms, the special hardware setup is restricted to Windows only. While the middleware OpenTracker could have been compiled for other platforms, there are no such drivers available for the Razer Hydra.

All the implementation work described in this chapter was carried out solely in Unity 3D. This is due to the fact that there already is an OpenTracker node (see Section 3.2) for the Hydra and a corresponding interface to Unity readily available. Section 6.1 is going to have details on the required setup.

Although the application's target platform is Windows, it was simultaneously developed in Windows and Mac OS X. This way easy portability is guaranteed once adequate drivers are available. For the time being, motion controller input was simulated by accordingly mapping keyboard input on OS X. The main development environments were Visual Studio 2010 on Windows and MonoDevelop 2.4.2 on OS X. C# was used as scripting language.

The following sections are going to elaborate on the most important parts of the implementation. First, the tracking mechanism, i.e. the handling of input data provided by the controller, is going to be described. The next sections will cover the inner structure of primitive objects, as well as the way the whole application is controlled. Finally, serialization and the custom user interface system are going to be touched upon.

Note that there is *not* going to be an in-depth description of every single class in the application. Only the general functionality will be laid out. Some essential classes and methods will be presented in detail, while some classes will not be mentioned at all.

6.1 TRACKING

The Razer Hydra, as described in Section 3.1, provides information about its controllers' location and orientation, as well as their buttons' state (i.e. whether they are pressed or not) to the system. This information is made available to Unity via the aforementioned OpenTracker

node. To enable communication between Unity, OpenTracker and the Hydra some additional files are required within the Unity project's folder. Said files are `opentracker.xml` for configuring the middleware layer and `UnityInterface.dll` for communicating with it. Moreover `DeviceDLL.dll`, `sixense.dll` and `sixense_utils.dll` are needed to access the Hydra itself. A detailed description of the entire setup and how OpenTracker itself needs to be configured can be found in Section 3.2.

The application-side interface to OpenTracker and the Hydra is represented by the `HydraTracker` class. It is derived from Unity's `MonoBehavior` class and, as such, has to be attached to a game object in order to have an effect.

In it's `Awake` function the Tracker creates a new `Tracking` instance, a class provided by `UnityInterface.dll` as bridge between OpenTracker and Unity. Afterwards, references to both controllers are retrieved and they are randomly labeled right- and left-hand. The correct sides are going to be assigned during calibration, which is going to be described below. Moreover, two instances of `HydraEvent` are created, instantiated with default values and declared to be the controllers' previous events.

As illustrated in Figure 24, `HydraTracker` continually receives input events from its instance of the `Tracking` class and subsequently forwards them to third classes actually handling them. The order of delivery of new events to said classes is subject to the following priority scheme:

1. If the calibration routine is currently running, all events are forwarded *only* to the respective classes.
2. Without a calibration going on, events are *solely* delivered to the GUI components, should any be visible.
3. If none of the above scenarios occurs, events are finally dispatched to all other registered listeners.

6.1.1 *HydraEvent*

The events provided by the `Tracking` class are of type `TrackingEvent`. These events hold information about the respective controller's location and orientation as well as its buttons' states.

Since more functionality is desirable, the wrapper class `HydraEvent` is introduced. `HydraEvents` basically contain the same information as `TrackingEvents`, but they are further aware of their preceding event. This way they can provide some additional information. For one thing it is possible to recognize state changes on buttons and triggers. It is also possible to not only determine the current location and orientation but also the motion and rotation occurring between the last event

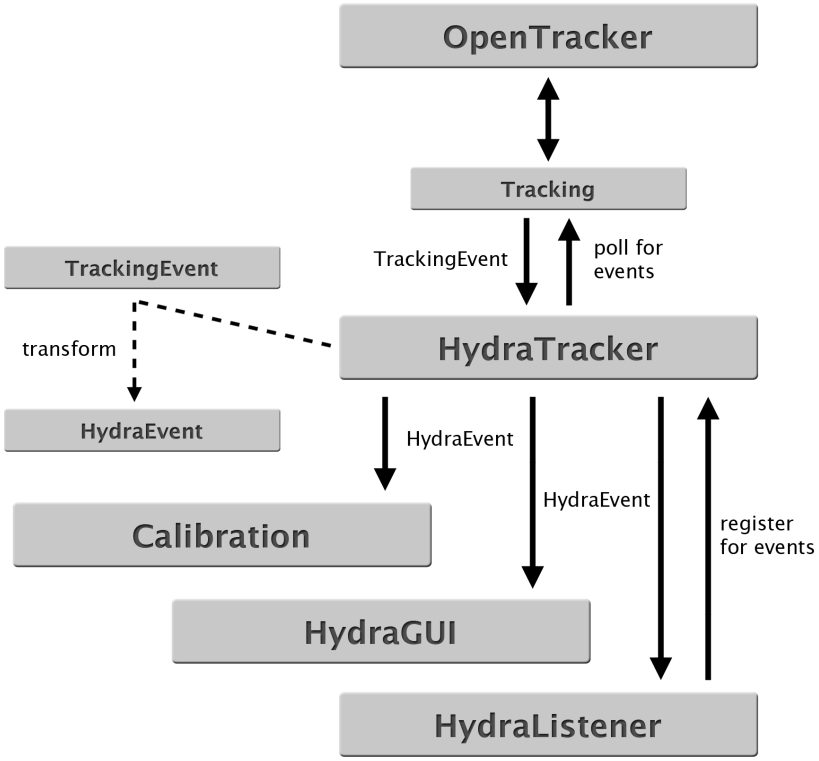


Figure 24: The event handling setup.

and the current one. `HydraEvents` also store position and orientation values, determined at calibration time. This allows for the computation of relative values that are decoupled from the controllers' initial positions and orientations. Lastly, in addition to its previous event, a `HydraEvent` also knows about the current event generated by the other controller. This way, the distance between the two controllers can easily be retrieved, which is, for example, needed for scaling objets (as described in Section 5.2).

6.1.2 *Event Handling*

In every update cycle of `HydraTracker`, OpenTracker is polled for both controllers' current events. The returned events, together with the controllers' last events, are then being used to create the respective `HydraEvents`.

In order to put this new information to use, any third class implementing the `HydraListener` interface can register with the tracker to receive notifications about new events. Listeners have to provide several functions, which are called within the tracker's `forwardTrackingEvent`

function. Which of these functions are actually called, depends on the values of the current `HydraEvent`.

Since the latter holds information about state changes, it is possible to explicitly call `buttonPressed` and `buttonReleased` functions, instead of simply forwarding all the button states all the time. The same holds true for the controllers' triggers, although it is slightly more complicated.

Unlike simple buttons, the values of which are either 0 or 1, a trigger can have values between 0 and 255, depending on how far it is pulled in. Thus, `HydraListeners` provide the function `triggerPressed`, which is called every time the trigger has a non-zero value. In order to use the trigger as a standard button, however, a threshold is defined. If the trigger produces a value below the threshold, it is considered an idle button. If its value is greater than the threshold, it is considered a pressed one. Whenever the trigger value crosses the threshold, either `triggerExceededThreshold` or `triggerDroppedBelowThreshold` is called on the listeners.

Just as all the listener-functions described so far, `joystickMoved` is only called when a controller's joystick is actually shifted. The only two functions that are called under all circumstances are `controllerMoved` and `controllerRotated`. This is because it can be safely assumed that a slight motion/rotation occurs in any case. Hence, checking for a difference between the previous and current event would mostly result in higher computational costs without any gain.

Every function declared in `HydraListener` gets at least the `HydraEvent` triggering it passed as parameter. As already mentioned above, event forwarding to registered listeners is suppressed under certain conditions. Both, active calibration processes, as well as visible GUI elements, consume all events before they are dispatched to the rest of the system.

6.1.3 *Calibration*

The software includes a simple calibration routine, which is automatically initiated on start-up. It is possible, however, to manually initiate the routine at any given time during execution. As described in previous chapters, calibration itself is fairly simple. The user is prompted to subsequently pull the trigger on the left- and the right-hand controller. Note that pulling the trigger on one controller twice will not yield any results, since the system requires single inputs from both controllers. As a trigger is pulled, visual feedback is provided. After pulling the right-hand trigger, changes are applied and the calibration screen is faded out.

Calibration serves two main purposes. One of them is to grant the user more flexibility concerning position in the room. The Hydra itself always calculates the controllers' positions relative to its base station,

with the latter representing the center of the coordinate system. From a usability perspective, however, it would be desirable to shift the center to where the user is located. This can be achieved by recording the controllers' position at a given point in time and then simply subtracting the recorded position from each newly delivered one. A newly created `HydraEvent` is, therefore, always passed the controller position recorded during the last calibration. So it is possible to query the event for both the position relative to the base station via `getPosition` and the position relative to the user via `getRelativePosition`. The same concept applies to controller orientation as well.

The second important role of calibration lies in assigning the correct sides to the controllers. By default, the system simply labels the controllers as numbers one and two. It is vital to the application, yet, to know which controller is held in which hand. By prescribing the sequence the triggers are supposed to be pulled in, this assignment can easily be done.

6.2 PRIMITIVES

Primitive geometric objects are the center piece of the software. Since Unity 3D only provides a very limited set of them by default, it was necessary to implement them specifically. In an early development stage, the intent was to only implement the shapes that were not available yet. As it turned out later, however, it was more practicable

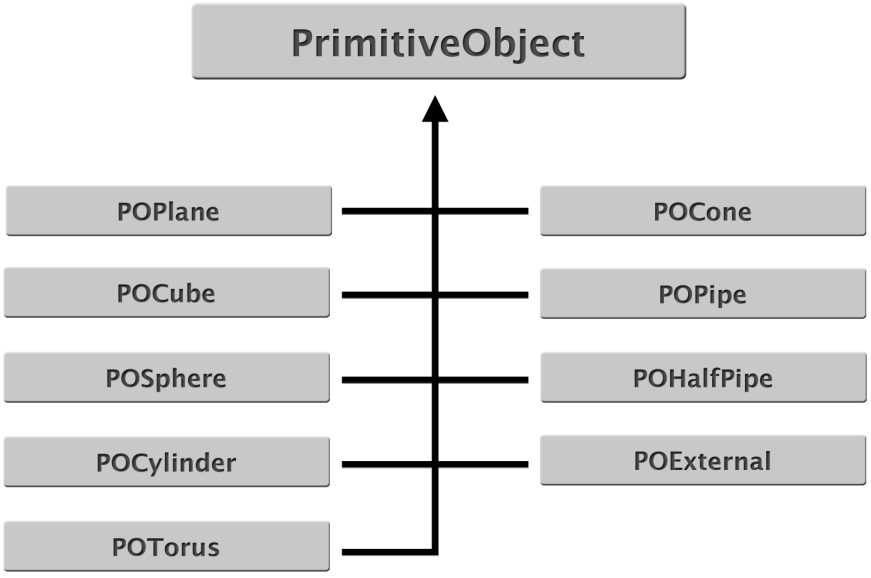


Figure 25: All availably objects are derived from `PrimitiveObject`. Note that some of the subclasses are capable of producing different objects depending on the parameters passed into their constructor (e.g. `POSphere` can also be used to create a half sphere).

to implement every primitive anew. Besides the obvious reasons of consistency throughout the whole application, there were other aspects in favor of this approach. In producing custom versions of all primitives, dynamic object creation was easier to implement. Moreover, this grants full control over object details and resolution.

As can be seen in Figure 25, the base class of all objects available in the application is `PrimitiveObject`. It is basically a wrapper around Unity's native `GameObject` class. The structure and composition of `PrimitiveObjects` is going to be laid out in detail below. Note that most of the functions mentioned herein are, while common to all primitives, actually implemented in subclasses of `PrimitiveObject`, such as `POCube`, `POSphere` or `POTorus`.

6.2.1 *Basics*

As mentioned above, `PrimitiveObject` is a wrapper class. It thus bears a `GameObject` - the base of all entities in Unity - at its core. At creation time, the `GameObject` is equipped with all the necessary components, to display a geometric shape.

```
obj.AddComponent<MeshFilter>();
obj.AddComponent<MeshRenderer>().material =
  new Material(Shader.Find("Diffuse"));
```

Both, a `MeshFilter` and a `MeshRenderer` are required to show an object. As can be seen, a pre-defined diffuse material is applied by default. With these components added to the object, all that is left to do, is to actually build the mesh to be displayed and assign it to the `MeshFilter`.

```
getMeshFilter().mesh =
  createMesh(resMajor, resMinor, 0, 0, out lines);
```

The `createMesh` function is common to every subclass of `PrimitiveObject`, but accepts different parameters for every type. The code in the listing above is taken from `POTorus`. In order to create a torus, the resolution as well as the value of both the major and minor radius have to be defined. Since the shape is dynamically created by a drag gesture, the values are initialized to zero and scaled up later in the creation process. The last parameter, `lines`, is used to store information about how to display the object's wireframe. The next section will further elaborate on wireframes.

The following listings are going to show parts of the torus' `createMesh` function along with a short explanation. Note that not the whole function is going to be depicted, as some parts are highly similar to each other.

Listing 1: POTorus.createMesh

```
private Mesh createMesh(int _resMajor, int _resMinor, float
    _rMajor, float _rMinor, out int[] _lines) {

  Vector3[] vertices = new Vector3[_resMajor*_resMinor*4];
  int[] triangles = new int[(int)(vertices.Length*1.5f)];
  Vector2[] uvs = new Vector2[vertices.Length];

  _lines = new int[vertices.Length];
```

As usual in computer graphics, a mesh is made up of connected triangles, which are in turn made up of vertices. Vertices are stored as three-dimensional vectors. The resolutions of the torus' two radii determines the amount of vertices needed to construct it. Triangles are simply stored as a list of indices into the vertex array.

In order to be able to apply a texture to an object, a set of UV coordinates has to be defined. For every vertex there is a UV coordinate, which basically maps the vertex onto a planar surface. Hence, the UVs are stored as an array of two-dimensional vectors.

Finally, there is one last array initialized at the beginning of the function. Similar to the triangle array, it will hold vertex indices, which are later used to draw the wireframe representation of the object.

```
  float dMajor, dMinor;

  dMajor = (2*Mathf.PI)/_resMajor;
  dMinor = (2*Mathf.PI)/_resMinor;

  int iV = 0;
  int iT = 0;
  int iL = 0;
  float t = 0.0f;
  float p = 0.0f;

  for (int i = 0; i < _resMajor; i++) {

    p = 0.0f;

    for (int j = 0; j < _resMinor; j++) {

      vertices[iV] = getTorusVertex(t, _rMajor, p, _rMinor);
      vertices[iV+1] = vertices[iV];
      vertices[iV+2] = vertices[iV];
      vertices[iV+3] = vertices[iV];

      uvs[iV] = getTorusUV(i, _resMajor, j, _resMinor);
```

```
      uvs[iV+1] = uvs[iV];
      uvs[iV+2] = uvs[iV];
      uvs[iV+3] = uvs[iV];
```

After initializing the necessary arrays, they have to be populated. Step sizes for each of the two radii, illustrated in Figure 26, are calculated (`dMajor` and `dMinor`) and, within a nested loop, vertices and UV coordinates are retrieved. The actual computation of the coordinates is handed off to `getTorusVertex` and `getTorusUV` respectively. Listing 2 and 3 provide details about the actual implementation of said functions.

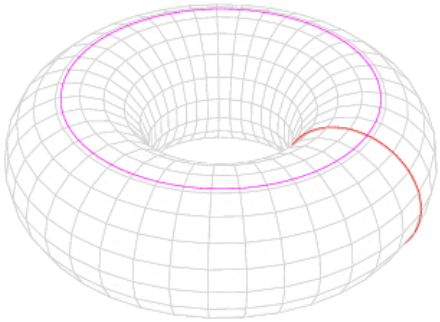


Figure 26: The two defining radii of a torus. The pink one is represented by `rMajor` in the code, while the red one is represented by `rMinor`. Source: *Wikipedia* [4]

Listing 2: `POTorus.getTorusVertex`

```
private Vector3 getTorusVertex(float t, float rMajor, float p,
    float rMinor) {

  Vector3 vertex = new Vector3();

  float cT = Mathf.Cos(t);
  float sT = Mathf.Sin(t);
  float cP = Mathf.Cos(p);
  float sP = Mathf.Sin(p);

  vertex.x = (rMajor + rMinor*cP)*cT;
  vertex.y = rMinor*sP;
  vertex.z = (rMajor + rMinor*cP)*sT;

  return vertex;
}
```

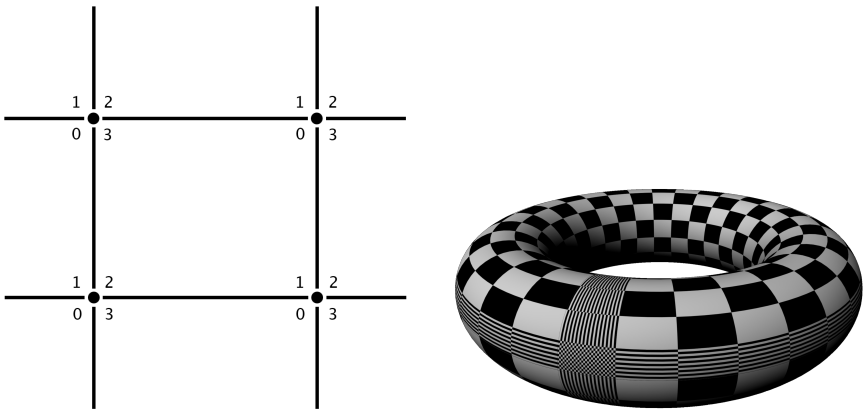
Listing 3: `POTorus.getTorusUV`

```
private Vector2 getTorusUV(float t, float resMajor, float p,
    float resMinor) {

  float u = 1.0f - ((1.0f/resMajor)*t);
  float v = ((1.0f/resMinor)*p);

  return new Vector2(u, v);
}
```

Attention should be paid to the fact that there are always sets of four identical vertices and UVs created at a time in order to avoid having to share vertices between quads. Vertices and UVs are consecutively stored within their respective arrays, and which of the four is effectively used for an individual polygon is determined by the fixed pattern displayed in Figure 27a.

The general reasoning behind using multiple vertices is twofold. First, this approach makes it possible to seamlessly apply a texture to the object. As every vertex can only have one UV coordinate assigned, problems in texture mapping would occur when using singular vertices. Concretely, the range between the last and first vertex on a radius span would show a reverse repetition of the texture between the first and last vertex (see Figure 27b). In using manifold vertices this shortcoming can be circumvented, by simply simulating single vertices with multiple UV coordinates.

The second advantage over singular vertices is that it is possible to either show an object smooth *or* flat shaded. This is due to the nature of the normals, which are calculated for each vertex later in the execution of `createMensh`. If two triangles share the same vertex, the vertex's normal is going to be averaged over the two triangles' normals,



(a) (b)

Figure 27: By using separate vertices for each quad, texturing issues can be resolved (b). Which concrete vertex is used at which point in the individual polygons is defined by the offsets in (a).

resulting in a smooth shading transition. With the approach taken, however, no triangles share any vertices. Thus the vertices' normals are always exactly those of the one triangle they are part of, which again yields no smooth transitions, hence flat shading.

To achieve smooth shading anyway, the function `smooth` can be called, in order to average out the normals of all identical vertices.

```
    if (i != 0) {

      if (j > 0) {

        triangles[iT]   = iV;
        triangles[iT+1] = iV - 3;
        triangles[iT+2] = iV - 4*(_resMinor + 1) + 2;
        triangles[iT+3] = iV - 4*(_resMinor + 1) + 2;
        triangles[iT+4] = iV - 4*_resMinor + 3;
        triangles[iT+5] = iV;

        iT += 6;

        _lines[iL]   = iV;
        _lines[iL+1] = iV - 4;
        _lines[iL+2] = iV - 4;
        _lines[iL+3] = iV - 4*(_resMinor + 1);

        iL += 4;
```

Next, the triangle strips are set up. In every cycle of the inner loop one quad, i.e. two triangles, are created. As already mentioned above, triangles are merely stored as a list of indices into the vertex array.

The same goes for the lines constituting the wireframe. It should be pointed out that the wireframe is made up of quads instead of triangles, which is due to aesthetic reasons alone. For every two triangles, there are two connected lines stored. This way, when all triangles are created, the lines form a coherent wireframe.

Note that neither triangle nor UV and line creation is always as straight forward as depicted in the above code listings. Special treatment has to be applied at the seams where the two ends of a radius sweep meet. These exceptions, however, are not going to be described any further, as they do not significantly differ from the aforementioned actions but simply require different vertex indices to be used.

```
      }
    }

    iV += 4;
    p += dMinor;
```

```
    }

    t+= dMajor;
  }

  Mesh m = new Mesh();

  m.vertices = vertices;
  m.triangles = triangles;
  m.uv = uvs;

  m.RecalculateNormals();
  m.RecalculateBounds();

  return m;
}
```

Once all the vertices, triangles, UVs and lines have been generated, a `Mesh` object is created and its major fields are set. The `Mesh` class, furthermore, provides means to automatically calculate normals and a bounding box around the object. As mentioned before, calling `Mesh.RecalculateNormals` results in a flat shaded object. Most subclasses of `PrimitiveObject`, however, provide a `smooth` function, which takes care of averaging normals in a way that yields a smooth shaded result.

When the creation of an object is done, i.e. the user released the right-hand trigger, `finishCreation` is called. Things happening in this function are rather straight forward.

First of all, the object is equipped with a `Rigidbody`, which is required for an object to physically interact with the scene. By default an object should not respond to gravity or collisions with other objects. This can be achieved by correctly setting the respective fields on the `Rigidbody`.

```
Rigidbody rBody = obj.AddComponent<Rigidbody>();
rBody.useGravity = false;
rBody.isKinematic = true;
```

Secondly, an identification is added. The `Identification` class is a helper component, the only purpose of which is to make informations about the `PrimitiveObject` available to the `GameObject`.

```
obj.AddComponent<Identification>();
obj.GetComponent<Identification>().setPrimitiveObject(this);
```

This is necessary due to the wrapper nature of `PrimitiveObject`. When collision detection is done, only the `GameObjects` involved in the collision can be retrieved by built-in Unity functions. If, however, a

`PrimitiveObject` needs to be informed about a collision, the wrapped `GameObject` needs to know about its wrapping instance.

Moreover, a special `WireFrameRenderer` is attached to the object and provided with the data calculated in `createMesh`.

```
WireFrameRenderer wfRenderer =
  obj.AddComponent<WireFrameRenderer>();
wfRenderer.setData(getMesh().vertices, lines);
wfRenderer.setActive(false);
```

6.2.2 *Wireframes*

As described in the previous chapter, any time an object is selected, a wireframe representation of it is overlaid as a visual cue. Since Unity does not provide a mechanism for rendering wireframes by default, it has to be built from scratch.

The class `WireframeRenderer` extends `MonoBehaviour` and makes use of Unity's low-level graphics functions. It is based on the `WireFrameLineRenderer` class posted in the Unity Forums by the user Rhapsodus [51]. Via `setData` the component is passed its parent object's vertex and line array right after it has been added. The actual rendering happens in `OnRenderObject`.

Listing 4: `WireframeRenderer.OnRenderObject`

```
public void OnRenderObject() {

  if (!rendering)
    return;

  LineMaterial.SetPass(0);

  GL.PushMatrix();
  GL.MultMatrix(transform.localToWorldMatrix);
  GL.Begin(GL.LINES);
  GL.Color(LineColor);

  if (builtFromMesh) {

    foreach( Line line in LinesArray ) {

      GL.Vertex( vertices[line.pointA] );
      GL.Vertex( vertices[line.pointB] );
    }
  }
  else {
```

```
    for (int i = 0; i < indices.Length; i+=2) {

      if (indices[i] != -1 && indices[i+1] != -1) {

        GL.Vertex(vertices[indices[i]]);
        GL.Vertex(vertices[indices[i+1]]);
      }
    }
  }

  GL.End();
  GL.PopMatrix();
}
```

The function uses immediate drawing commands to superimpose the wireframe over the object.

First the current model-view and projection matrix are stored by calling `GL.PushMatrix`. Since direct drawing happens in world coordinates, the world coordinate system has to be aligned with the object's local coordinate system. This is achieved by multiplying the model-view matrix with a matrix that transforms an object's local coordinates into world coordinates. Every `GameObject` in a Unity scene is equipped with a `Transform` component that, amongst other things, automatically provides said matrix. Finally the lines are drawn according to the input data and the previously saved matrices are restored.

Note that there are two different types of wireframes drawn, depending on the way the data has been computed. This is due to the software allowing the incorporation of external objects into a scene.

External objects are shapes that have been defined in any other 3D computer graphics software and imported into the Unity scene before starting the application. Since these objects have not been created by the software, they do not carry any wireframe data. In order to make up for this deficit, it is possible to build a wireframe from an existing mesh as well. These types of wireframes, however, are made up of triangles as opposed to quads, since it is simply not possible to find a quad representation without explicitly knowing about a mesh's vertex arrangement.

6.2.3 *Colliders*

The last thing that is added to a `GameObject` in `finishCreation` is a `Collider`. As the name indicates, this component is required for an object to actually detect collisions with other objects. Unity offers a variety of different `Colliders`, such as `BoxCollider`, `SphereCollider` or `CapsuleCollider`. The most versatile variation, however, is the `MeshCollider`. While all the aforementioned have a fixed, unalterable

shape, a `MeshCollider` can be assigned *any* given shape as `sharedMesh`. Since every object in a scene can be scaled freely and, above all, non-uniformly along any given axis, the use of `MeshColliders` is indispensable.

While using `MeshColliders` offers a lot of flexibility on the one hand, there are also a number of problems introduced, on the other. For one thing, `MeshColliders` can only collide with one another, if they are flagged as convex. Thus, as all objects need proper collision detection, all the colliders in the application have to be convex. Unity, though, limits the number of triangles of a convex collider's `sharedMesh` to 255. Most available primitives feature less than 255 triangles, hence there is no problem. Some, however, like e.g. a sphere, require significantly more triangles to be smoothly displayed. For these shapes, a separate mesh has to be generated at a lower resolution in order to be applied to a `MeshCollider`.

Another problem occurs due to the required convexity of colliders alone, because there are some non-convex primitives, like, for instance, `POTorus` or `POPipe`, available. In order to enable collisions for these kinds of shapes, compound colliders are used. Compound colliders are realized by recreating an object's mesh as a series of convex meshes and applying those to `MeshColliders`. The empty `GameObjects` carrying the colliders are attached to the base object and their transformation matrices are synchronized.

```
root.transform.position = obj.transform.position;
root.transform.rotation = obj.transform.rotation;
root.transform.parent = obj.transform;
```

Hereby, the child objects and with them their attached `MeshColliders` directly reflect every transformation that is applied to the base object.

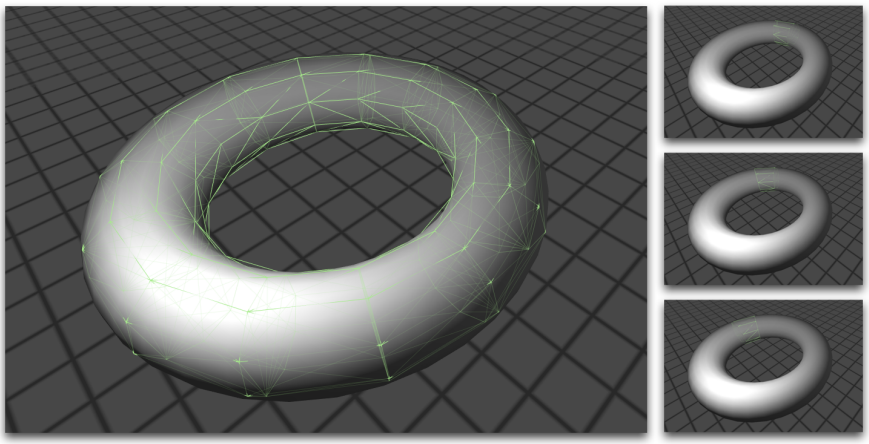


Figure 28: A `POTorus`' compound collider (left) and three of its constituting sub-colliders (right).

Figure 28 illustrates the concept of compound colliders by means of a `POTorus` objects.

6.2.4 *View-Dependent Scaling*

The last thing about primitive objects that shall briefly be touched upon is the way they are scaled. In order to provide the most intuitive handling experience, objects are always scaled exactly along the axis, along which the controllers are moved apart from each other. Since the camera rendering the scene can be rotated, the axes reflecting controller motion in virtual space are not necessarily aligned with the main coordinate axes. They are much rather dependent on the angle of view on the scene.

When changing the scale of a `GameObject` in Unity, the change is applied in local space. Initially the axes of an object's local coordinate system are aligned with those of the world coordinate system. Hence, in order to scale an object relative to the viewport and successively relative to the controllers, its coordinate system has to be aligned with the camera's coordinate system before the transformation.

To achieve this alignment, the object can simply be rotated accordingly. The natural rotation of the object along with the coordinate system, however, is an undesired side effect in this case. Thus, the object has to be reset to its previous state, without resetting the coordinate system. This is achieved by simply applying an inverse rotation to the vertices of the primitive's mesh, as can be seen in the following listing. A direct manipulation of vertex data, of course, requires some further action to be taken. For one thing, the `WireframeRenderer` has to be informed of the change. Moreover, any compound colliders that may be attached, or rather their meshes' vertices, have to be rotated as well.

Listing 5: `PrimitiveObject.rotatePivot`

```
public virtual void rotatePivot(Quaternion rot) {

  Vector3 scale = obj.transform.localScale;
  Quaternion rrot = obj.transform.rotation;
  obj.transform.localScale = Vector3.one;

  obj.transform.rotation = rot;

  rot = Quaternion.Inverse(rot);

  Mesh mesh = getMesh();
  Vector3[] vertices = mesh.vertices;

  for (int i = 0; i < vertices.Length; i++) {
```

```
    vertices[i].Scale(scale);
    vertices[i] = rrot*vertices[i];

    vertices[i] = rot*vertices[i];
  }

  mesh.vertices = vertices;
  mesh.RecalculateBounds();
  mesh.RecalculateNormals();

  smooth();

  getWireframeRenderer().setData(vertices, lines);

  /* Also rotate all attached collider meshes */
  Vector3[] v;

  foreach (MeshCollider mc in getNonDefaultColliders()) {

    v = mc.sharedMesh.vertices;

    for (int i = 0; i < v.Length; i++) {

      v[i].Scale(scale);
      v[i] = rrot*v[i];

      v[i] = rot*v[i];
    }

    mc.sharedMesh.vertices = v;
  }
}
```

6.3 CONTROL CLASSES

Within the whole system, there are two particularly important classes when it comes to controlling the application. They are, in a way, the user's interface into the virtual world. One of them is concerned with handling navigational elements, like the crosshair and camera. The other one coordinates all the other parts of the application, from communication with the graphical user interface to the actual handling of primitive objects. Both of the classes are being presented in the following two sections.

6.3.1 *Main*

The `Main` class is the application's central hub. This is where all the information is gathered and processed in order to, for example, create

and manipulate objects. It is also where most of the things, which are not by default part of the Unity scene, are initiated and set up. Moreover, a number of constants is defined, which are used all over the system.

All the setup work is carried out in the `Start` function, inherited from `MonoBehaviour`. After the class registers itself as a `HydraListener` at the tracker and the `Navigation` (see next section for details) is initiated, the user interface menus are created. The inner workings of the GUI will be further elaborated on in Section 6.5.

There are also three distinct lists created to manage all animators[1] within the application. This is due to the fact that animators can only be added or removed at the beginning of a new update cycle. Thus, three lists are maintained: one holds the currently active animators, while the others hold the animators scheduled for destruction or addition during the current cycle. `Animator` management is carried out in the `Update` function.

One more thing happening in the `Start` function is the import of external objects. As already mentioned in Section 6.2, the application allows the use of externally generated 3D objects. This functionality is deemed especially useful for easily building an environment for the scene, or providing more complex objects to the user. Classically these are tasks required for level design. Though levels or similar functionalities are not yet implemented, they are extensions that might be added later. In order to use external objects, they have to be added to the Unity scene and their name has to be prefixed by the characters "ToPo_". Within the `Start` function, all `GameObjects` in the scene are queried for the "ToPo_" prefix in the name. If a matching object is found, it is wrapped in a `PrimitiveObject`.

Most of the remaining functions in `Main` are either functions common to all `HydraListeners` (see Section 6.1 for details) or callback functions for the previously created user interface components. The `HydraListener` functions handle the actual processes of object creation, selection and manipulation, as they are described in Section 5.2. The UI callbacks contain functions such as `makePrimitiveObject`, which schedules a new primitive for creation the next time the right-hand trigger is pulled. Also, `deleteSelection`, `setColor` or `togglePhysics` are frequently occurring callbacks. The use of most of these functions, however, is self-explanatory and their inner workings are rather straight forward. This is why, in spite of their undoubted importance to the application, they shall not be explicitly discussed.

1 `Animator` is a helper class providing a variety of different animation curves to any class in need of animated values.

6.3.2 *Navigation*

One of the most important classes that is instantiated at start-up in `Main` is `Navigation`. It is where both the camera and the crosshair are controlled and thus implements the `HydraListener` interface to communicate with `HydraTracker`.

Crosshair

In the `Navigation` constructor the crosshair's visual representation is set up by creating the respective `GameObjects`. Note that an axis in the crosshair is not realized as a standard mesh object. Instead, there is a `LineRenderer` attached to an object. The laser-beam-like appearance is created by telling the `LineRenderer` to use a transparent material and a special gradient texture. The actual positioning of the crosshair happens in `drawLines`. This is where the axes' vertices are calculated according to the desired crosshair position and passed to the `LineRenderer`.

The constructor also takes care of initially positioning the camera and pointing it at the center of the crosshair, as well as registering with the tracker.

Moving the crosshair is fairly simple. Whenever the tracker invokes `controllerMoved`, the motion of the right-hand controller is retrieved and forwarded to `moveCrosshair`. There, the crosshair's center is translated relative to its local coordinate system and a re-draw of the lines is triggered. Should the point that the camera is rotating around lie behind the camera, the motion along the x- and z-axis has to be inverted first.

The more interesting function when it comes to crosshair control is `adjustCrosshair`. It is called in order to re-align the crosshair with the viewport whenever the camera is transformed in any way. In order to do so, the crosshair is first moved to the point the camera is focussing on, respectively rotating around. At this point it is rotated around the y-axis, so that its z-axis is aligned with the camera's z-axis. Unity's built in `LookAt` function is utilized to do the rotation, as it saves the overhead of calculating rotation angles.

Once the crosshair is re-aligned with the camera, all that is left to be done is to reset its position. Since the crosshair's position relative to the camera should be preserved, however, it is not possible to simply move the crosshair to the coordinates it was at before. Much rather, its position relative to the camera is constantly monitored and stored in a dedicated variable (`cameraRelativePosition`). This makes it possible to maintain the spatial relation between crosshair and camera at all times, even if the latter has been moved.

Listing 6: `Navigation.adjustCrosshair`

```
private void adjustCrosshair(bool fixedWorldPosition) {

  /* This option allows to maintain the crosshair's world
      position instead of re-aligning it with the camera. It is
      never actually used. */
  if (fixedWorldPosition)
    cameraRelativePosition = cam.transform.InverseTransformPoint(
        crosshair.transform.position);

  crosshair.transform.position = focusPoint;
  Vector3 lookAtPoint = cam.transform.position;
  lookAtPoint.y = focusPoint.y;
  crosshair.transform.LookAt(lookAtPoint);

  crosshair.transform.position =
    cam.transform.TransformPoint(cameraRelativePosition);

  drawLines(length);
}
```

Camera

All the functions that control the camera at application side have to call `adjustCrosshair` as described above. Controlling the camera is fairly straight forward. One thing that still has to be kept in mind is that two particular variables have to be updated any time the camera is manipulated. These variables are `cameraRelativePosition`, which has already been covered above, and `camToFocusPoint`. The latter stores the vector between the camera and its focus point. The focus point, in this sense, denotes the point the camera is rotating around. It is, thus, also the point that is in the center of the viewport, unless the camera is being moved beyond the point when zooming.

The following listing depicts the code used to move the camera. It can be seen that `cameraRelativePosition` has to be updated at the beginning of the function, while `camToFocusPoint` is set at the end. This is because the former is used in `adjustCrosshair` to re-establish the spatial crosshair-camera rotation prior to the camera transformation. It shall also be emphasized that the camera is always moved parallel to the world's xz-plane. Therefore, the actual camera motion is split up into a standard translation along the camera's x-axis and one along the camera's z-axis projected onto said plane. The projection is achieved by getting the camera's `forward` vector [2], setting its y component to zero and then normalizing it.

One more thing worth mentioning is that, when moving the camera, its focus point is always moved along.

2 Unity's name for the local z-axis in world space.

Listing 7: `Navigation.moveCamera`

```
private void moveCamera(float x, float z) {

  if (isAnimating)
    return;

  cameraRelativePosition = cam.transform.InverseTransformPoint(
      crosshair.transform.position);

  cam.transform.Translate(x/1000, 0.0f, 0.0f);
  Vector3 zm = cam.transform.forward;
  zm.y = 0;

  cam.transform.position += Vector3.Normalize(zm)*(z/1000);

  setFocusPoint(cam.transform.position + camToFocusPoint);

  camToFocusPoint = focusPoint - cam.transform.position;

  adjustCrosshair(false);
}
```

The functions `zoomCamera` and `rotateCamera` are analogically structured. They naturally only differ in the way the camera is transformed in the center part of the function.

While the zoom effect is achieved by simply translating the camera along its local z-axis, the rotation is carried out via Unity's `RotateAround` function. Said function allows to define a pivotal point and an axis passing through that point, which the rotation is going to be carried out around.

```
cam.transform.RotateAround(focusPoint, cam.transform.right,
  -rotation.x);
cam.transform.RotateAround(focusPoint, new Vector3(0, 1, 0),
  -rotation.y);
```

Note that vertical rotation happens around the camera's local x-axis, while horizontal rotation is performed around the global y-axis.

Another vital function concerning camera control is `pointCameraAtSelection`. It is called whenever the user wants to focus the camera on an object, putting it in the center of the screen. Re-focussing is done by first retrieving the center of the currently selected object. Then the object is centered by moving the camera with the aid of the previously described `camToFocusPoint` vector.

```
cam.transform.position = CustomSelection.center -
  Vector3.Normalize(camToFocusPoint)*50;
```

The listing shows that the `camToFocusPoint` vector is being normalized and scaled up to a fairly large size. This is the first step in finding the optimum distance from the camera to the object, so the latter is being displayed as big as possible while not overlapping the screen bounds too much. Since it can safely be assumed that after moving the camera far away from the object, is is now entirely visible, and thus its on-screen bounds can be calculated. Together with the size of the screen itself and the distance between object and camera, an estimate can be made as to how close the camera can be moved to achieve the desired results. This target distance is stored in the vector `td` in the following listing.

```
float cH = CustomSelection.getScreenBounds().height;
float tH = Screen.height;

Vector3 tD =
  (CustomSelection.center - cam.transform.position)/(tH/cH);
```

For reasons of simplicity, the current implementation only takes an object's height into consideration. It would, of course, be more accurate to use the maximum of both width *and* height as a reference.

6.4 SERIALIZATION

As already mentioned in Chapter 5, it is possible to save any scene to the local file system and restore it later. In order to provide such functionality, the `Serializer` class, which most of the logic involved in writing and reading files is located in, has been implemented. The following section is going to dicuss the file format used and how the scene data is actually serialized.

When a scene is first saved, a new folder is created to hold all the generated data. Since the user cannot choose where to save a scene on the file system, a default location for all scenes is provided. They are automatically stored in a folder named "projects" inside the application's root folder. The newly created folder is named according to the current date and time, using the format *ddMMyyyyHHmmss*. Within this folder, one more folder is created and named "metadata". This is where geometric information about the objects in the scene is going to be saved.

Before anything happens with the data, a screenshot of the application is captured. It is stored along with the scene and is going to serve as a visual cue for the user, when browsing through all the saved scenes. Next, the objects are serialized.

6.4.1 *Object Data*

An instance of `POData` is created for every `PrimitiveObject` in the scene. All information required to fully restore an object is written into this instance. The respective fields of `POData` are listed below.

- `id`: Every object in the scene gets a unique ID at creation time.
- `type`: The concrete subclass of `PrimitiveObject` (e.g. POCube, POSphere, POTorus, etc.).
- `position`: The object's position as a 3D vector in world coordinates.
- `rotation`: The object's rotation as a 3D vector in world coordinates.
- `color`: An array of floating point numbers storing the object's color as red, green, blue and alpha values.
- `shader`: Since only predefined shaders are provided, they can simply be referenced by a name.
- `material`: The above holds true for materials as well.
- `isStatic`: Whether the objects is affected by gravity and collision or not.
- `meshFile`: The URL of the file, containing the object's actual mesh data.

As already mentioned above, an object's geometric data is separately stored into a mesh file. The structure of this file is based on the `.obj` file format for geometry definition [1]. It can contain different parts, depending on the object's composition. The parts are separated by a simple heading string.

The first part is the object's basic mesh, as indicated by the preceding text string `MainMesh`. This part contains separate lists of vertices, vertex normals, UV coordinates and triangles. Vertices and vertex normals are stored as a triple of decimal numbers separated by blank space characters and prefixed by the characters `v` and `vn` respectively. UV coordinates are represented by pairs of decimal numbers prefixed by `vt`, while triangles are again triples of integers with a preceding `f`. Note that each list is prefixed with a line containing the number of entries in the list. This is so as to facilitate the creation of a new mesh when loading a scene. The following listing shows excerpts of a mesh file's `MainMesh` part.

Listing 8: Parts of a file containing raw mesh data

```
MainMesh
numV 3200
v -2.205358 0 2.205357
v -2.185292 0.179168 2.185292
v -2.127059 0.3407978 2.127058
v -2.036358 0.469068 2.036357

...

numVN 3200
vn -0.6963012 -5.960464E-08 0.6963004
vn -0.6623973 0.3044211 0.6623967
vn -0.5638606 0.5794467 0.5638601
vn -0.4100215 0.7982264 0.410021

...

numVT 3200
vt 0 1
vt 0 0.05
vt 0 0.1
vt 0 0.15

...

numF 4800
f 84 81 2
f 2 7 84
f 88 85 6
f 6 11 88

...
```

If the object has any special colliders attached, be it a lower resolution one or a compound collider, the collider meshes are added next. The beginning of a new collider mesh is indicated by a `ColliderMesh` heading. Apart from the heading, the representation of a collider mesh does not differ from the base mesh in principle. There are, of course, no UV coordinates to be stored. Thus the respective list is empty and the preceding string reads `numVT 0`.

The last part of the mesh file contains the wireframe information for the object. Its beginning is marked with the string `Wireframe` and it contains pairs of integers, representing vertex indices. The pairs are prefixed by the character `l`, as can be seen below.

```
...
```

```
Wireframe
numL 3200
l 84 80
l 80 0
l 88 84
l 84 4

...
```

6.4.2 *Scene Data*

After all meshes have been serialized, the remaining information about the scene has to be stored as well. The respective data is written into an instance of `SceneData`, the fields of which are listed below.

- `created`: The date and time the scene has been first saved.
- `lastSaved`: The current date and time.
- `cameraPosition`: The camera's position as a 3D vector in world coordinates.
- `cameraFocus`: The camera's focus point as a 3D vector in world coordinates.
- `cameraRotation`: The camera's rotation as a 3D vector in world coordinates.
- `poData`: The array of previously serialized `PrimitiveObject` data.
- `groupData`: An array holding information about which objects are grouped together.

When all the data has been assembled into the described format, it can be written into a file. Contrary to the mesh files, which are written in plain text, the general scene and object data is written in XML format. The creation of XML files itself is fairly easy, as C# provides its own `XMLSerializer` class. Below, the contents of a file generated from a simple scene containing only one single object are listed.

Listing 9: Contents of a scene file

```
<?xml version="1.0" encoding="utf-8"?>
<SceneData xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <created>2012-03-01T15:42:42.589204+01:00</created>
  <lastSaved>2012-03-01T15:42:42.589204+01:00</lastSaved>
  <cameraPosition>
```

```
    <x>24.4949</x>
    <y>20</y>
    <z>-24.4948978</z>
  </cameraPosition>
  <cameraFocus>
    <x>0</x>
    <y>0</y>
    <z>0</z>
  </cameraFocus>
  <cameraRotation>
    <x>30.0000038</x>
    <y>315</y>
    <z>9.858529E-07</z>
  </cameraRotation>
  <poData>
    <POData>
      <id>0</id>
      <type>POTorus</type>
      <position>
        <x>2.51840544</x>
        <y>3.64464784</y>
        <z>0.0582609177</z>
      </position>
      <rotation>
        <x>0</x>
        <y>135</y>
        <z>0</z>
      </rotation>
      <color>
        <float>0.5</float>
        <float>1</float>
        <float>0</float>
        <float>1</float>
      </color>
      <shader>Specular</shader>
      <material>bouncy</material>
      <isStatic>false</isStatic>
      <meshFile>projects/01032012154242/metadata/0</meshFile>
    </POData>
  </poData>
  <groupData />
</SceneData>
```

In conclusion, it shall be noted that, when loading a scene, the steps described in this section are mostly just run through in reverse order. It is therefore not deemed necessary to go into any further detail on the topic of restoring saved scenes.

6.5 GUI

In order to make all the functionality of the application available to the user, a graphical user interface is required. The use of the Hyrda as input device, however, requires the implementation of a custom user interface system. The system needs to map controller movement to the motion of a cursor on the screen. Moreover, the trigger assumes the functionality of a mouse button and is the main way to interact with user interface elements. Unity's built-in GUI components cannot be used, since they only respond to events generated by standard input devices.

The core part of the system is the class `HydraGUI`, which is the only GUI class actually communicating with the tracker. It does not implement the `HydraListener` interface though. In fact, the HydraGUI prevents events from being forwarded to listeners, should any GUI component be visible. This is checked for in `HydraTracker.forwardTrackingEvent`. If the `HydraGUI` reports a visible component, there are three different functions called on it. The first, `firePointerMovedEvent`, is called under all circumstances, since it can be safely assumed that the controller moves during every update cycle. The other two functions, `fireButtonPressedEvent` and `fireButtonReleasedEvent`, are called only if a trigger is pulled or released.

In `HydraGUI` the responsibility of the mentioned functions is to generate `HydraGUIEvents`. These events bear information about the current state of the trigger (i.e. whether it is pulled or not) and the position of the cursor on the screen. The latter is calculated in `mapControllerToScreen` each time `firePointerMovedEvent` is called.

Both, the controller's position and orientation contribute to the computation of screen coordinates.

```
position = new Vector2(Screen.width/2 - Screen.width/40 * r.y,
  Screen.height/2 - Screen.height/20 * r.x);

position.x += m.x;
position.y += m.y;
```

Note that the horizontal resolution is twice the vertical resolution. While this is actually incorrect, as any screen's aspect ratio is most likely different, it proved to be a good overall measure, yielding good usability.

The newly generated events are subsequently forwarded to registered `HydraGUIComponents`, subclasses of which represent the actual user interface elements. Every component registers itself at the `HydraGUI` during creation and provides the following methods to be notified about input events.

- `buttonPressed`: Called when the trigger is pulled. Note that the trigger is referred to as a button, since these are GUI related functions.
- `buttonReleased`: Called, when the trigger is released.
- `pointerMoved`: This function is called to notify the component that the cursor has been moved.
- `swipeToRight`: Whenever the cursor is horizontally rotated more than forty degrees in either direction a swipe event is triggered. In this case the controller was rotated to right.
- `swipeToLeft`: This indicates a rotation to the far left.

Certain `HydraGUIComponents`, such as `HydraGUIButton` are able to trigger external actions. In order to do so they get a callback function passed in the constructor. The function is invoked within `buttonPressed`, if the pointer is inside the components bounds.

Other components, such as `HydraGUIComponentGrid`, can be used to lay out user interface elements on the screen. When instantiated, they get a random number of `HydraGUIComponents` passed and set their size and bounds according to the layout they are to produce.

The function that actually brings user interface elements to the screen is `draw`. It uses Unity's built in `GUI` class, or rather its `drawTexture` function, to render the visual contents of a `HydraGUIComponent` to the screen. The rendering is coordinated by `HydraGUI`. It calls the `draw` function on every visible component and simultaneously suppresses rendering of the crosshair. Also, `HydraGUI` takes care of drawing the cursor. It shall be mentioned that the crosshair is hidden in order to avoid distracting interferences between cursor motion and crosshair motion.

`HydraGUI` itself is instructed to initiate a new render cycle from within the `OnGUI` function in `Main`. This function is common to all subclasses of `MonoBehaviour` and is where Unity renders user interface elements and handles respective events.

7 CONCLUSION & FUTURE WORK

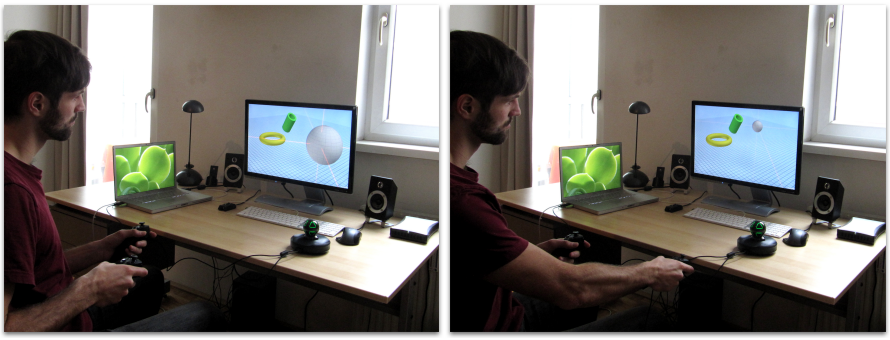
Spatial ability is an integral part of human intelligence. It influences how we perceive the world around us, orient ourselves within it and create a mental model of it in order to assess both real and hypothetical situations. High levels of spatial ability are commonly linked to creative thinking in any domain, from liberal arts to natural sciences, which emphasizes its significance even further.

Considering these facts, the improvement and training of spatial abilities is a desirable goal in all levels of education. As shown in a variety of studies, different approaches can be taken in order to achieve this goal. Traditional geometry education, for example, has been found to have positive effects on spatial thinking as well as engagement with dedicated applications in virtual environments or even playing video games developed for entertainment purposes.

The application developed in the course of this thesis, in a way, borrows from all three of these. It provides basic features of 3D Dynamic Geometry Software and combines them with a motion sensing input device in order to increase the level of immersion for the user. Moreover, the integration of a physics engine adds a sense of playfulness to the application, which is especially important considering its target audience are children in elementary education.

In its current version, users are able to create and arrange a multitude of geometric objects within a three-dimensional scene. Objects can be moved, rotated and scaled at will and the scene can be viewed from any desired angle just as it is possible in every common 3D modeling software. Furthermore, objects can be duplicated, they can be arranged in groups and assigned different materials and colors, which, again, is deemed important in consideration of the target audience. If desired, physics can be activated, which results in all the scene's objects correctly reacting to gravity and collisions. With physics on, users are also able to pick up objects, toss them around or nudge others with them.

It is worth noting that the application, in its state at the end of this work, is to be considered a working prototype rather than a fully functional and ready-to-deploy software. The basic functionality described above is available and working (Figure 29 shows the application in use), but the entire software lacks thorough testing for both implementation errors and usability issues. Tests for the latter would be especially interesting, given the fact that most of the application's functionality is encoded in controller buttons and combinations of the same, rather than in user interface elements. One the one hand, this is



(a) (b)

Figure 29: The developed application in use. Note the direct correlation between the right hand's position and that of the gray sphere in the scene.

deemed a justifiable design decision, since the handling of graphical user interfaces with devices such as the Razer Hydra is inherently difficult due to low pointing precision. On the other hand, though, this means an overhead effort on the user to memorize the individual controller buttons' functionalities and their potential combinations (Appendix A provides a detailed overview of the mappings). Therefore, one of the most important things to do, when continuing work on this project are extensive user tests. It shall be pointed out here that tests of this capacity would have certainly been desirable in the context of this thesis already. The amount of necessary implementation work, though, which included an entirely new event and user interface system to integrate the Hydra into the Unity context, unfortunately left no more time for user testing.

Apart from extensive test sessions, one can surely think of many features that might be added to future versions of the software. For one thing, the possibility to interconnect objects with joints, hinges, chains or ropes would be highly desirable. An early version of a rope connection has already been implemented in the course of the thesis but remains error-prone and has not been further developed due to differing priorities and lack of time. It has, therefore, been dropped from documentation within this work, although a reference to it can still be found in Figure 22b (top-right icon).

Another major area for improvement are the application's geometry functionalities. Many features of Dynamic Geometry Software could be added here, such as constrained and unconstrained objects, the ability to record and replay constructions as well as boolean operations and intersections. An addition of these features would, of course, broaden the application's target audience from children in elementary to students in all levels of education.

Especially considering the use in elementary education, however, an interesting direction of future work would be the development of

concrete lecture material, including prepared scenes with work tasks. Subsequent tests of these lectures in the classroom would, of course, be particularly desirable.

Part III

APPENDIX

APPENDIX A

LEFT	RIGHT	ACTION
-	Move	Move Crosshair
-	Start	Freeze Crosshair
-	Joystick Button	Reset Crosshair
-	Hold Joystick Button	Start Calibration
Rotate	-	Rotate View
Joystick	-	Move View
Bumper + Joystick	-	Zoom View
Hold Joystick Button	-	Reset View
Joystick Button	-	Focus on Selected Object

Table 1: Crosshair & Camera Interaction

LEFT	RIGHT	ACTION
-	Trigger	Click
-	Button 1	Creation Menu
-	Button 2	Manipulation Menu
-	Button 3	Colors & Material Menu
-	Button 4	File Menu
-	Rotate Far Left/Right	Switch Pages in File-Chooser

Table 2: GUI Interaction

LEFT	RIGHT	ACTION
-	Trigger + Move	Create Object
-	Bumper	(Un-)Select Object
-	Bumper + Move	(Un-)Select Multiple Object
-	Trigger + Move	Move Object
Bumper	Trigger + Rotate	Rotate Object
Button 3	-	Rotate Around Crosshair
Trigger	Trigger + Move	Scale Object
Button 1	-	Constrain to x-Axis
Start	-	Constrain to y-Axis
Button 2	-	Constrain to z-Axis

Table 3: Object Interaction

BIBLIOGRAPHY

[1] Object Files (.obj). URL http://paulbourke.net/dataformats/obj/.

[2] Razer Hydra - Technical Specifications.

[3] *Getting Started with Cabri Geometry II for Macintosh, Windows and MS-DOS*, 1999.

[4] Torus Cycles. Wikipedia, the free encyclopedia, May 2005.

[5] Reference of Cabri II Plus. CABRILOG SAS, 2007.

[6] *GeoGebra Handbuch*, September 2011.

[7] Autodesk Maya 2013 Help, 2012. URL http://download.autodesk.com/global/docs/maya2013/en_us/.

[8] G. Agugiaro, F. Remondino, G. Girardi, J. Von Schwerin, H. Richards-Rissetto, and R. De Amicis. A Web-Based Interactive Tool for Multi-Resolution 3D Models of a Maya Archaeological Site. *Int. Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, 38, 2011.

[9] VM. Anastassiou, P. Diamantopoulos, and S. Vassos. iThink: A Library for Classical Planning in Video-Games. *stavros.lostre.org*.

[10] E. Bainville. Manual of Cabri II Plus. CABRILOG SAS, March 2007.

[11] A. Basu, C. Saupe, E. Refour, and A. Raij. Immersive 3DUI on One Dollar a Day. *2012 IEEE Symposium on 3D User Interfaces*, pages 97–100, March 2012.

[12] D.A. Bowman and L.F. Hodges. An Evaluation of Techniques for Grabbing and Manipulating Remote Objects in Immersive Virtual Environments. *Proceedings of the 1997 symposium on Interactive 3D graphics*, pages 35–ff., 1997.

[13] J. Carlquist. Evaluation of Teaching Evolution Using Evolutionary Algorithms in a 2D Physics Sandbox. Master's thesis, Umeå University, Umeå.

[14] E. Dahlberg. Electricity in a 2D Mechanics Simulator for Education. Master's thesis, Umeå University, Umeå.

[15] R.E. Downing, J.L. Moore, and S.W. Brown. The Effects and Interaction of Spatial Visualization and Domain Expertise on Information Seeking. *Computers in Human Behavior*, 21(2):195–209, 2005.

[16] R.B. Ekstrom, J.W. French, H.H. Harman, and D. Demen. Manual for Kit of Factor-Referenced Cognitive Tests, 1976, 1976.

[17] J. Feng, I. Spence, and J. Pratt. Playing an Action Video Game Reduces Gender Differences in Spatial Cognition. *Psychological Science*, 18(10):850–855, 2007.

[18] L. Gallo, M. Ciampi, and A. Minutolo. Smoothed Pointing: a User-Friendly Technique for Precision Enhanced Remote Pointing. *Complex, Intelligent and Software Intensive Systems (CISIS), 2010 International Conference on*, pages 712–717, 2010.

[19] W. Goldstone. Unity 3.x Game Development Essentials. Packt Publishing Ltd, 2011.

[20] B. Güven and T. Kosa. The Effect of Dynamic Geometry Software on Student Mathematics Teachers' Spatial Visualization Skills. *The Turkish Online Journal of Educational Technology*, 7(4), October 2008.

[21] D.F. Halpern and M.L. LaMay. The Smarter Sex: a Critical Review of Sex Differences in Intelligence. *Educational Psychology Review*, 12(2):229–246, 2000.

[22] M. Hegarty and D. Waller. A Dissociation Between Mental Rotation and Perspective-Taking Spatial Abilities. *Intelligence*, 32(2): 175–191, 2004.

[23] M. Hohenwarter. GeoGebra - Ein Softwaresystem für dynamische Geometrie und Algebra der Ebene. *Unpublished Master Thesis, Universität Salzburg, Austria*, 2002.

[24] M. Hohenwarter and K. Fuchs. Combination of Dynamic Geometry, Algebra and Calculus in the Software System GeoGebra. *Computer Algebra Systems and Dynamic Geometry Systems in Mathematics Teaching Conference. Pecs, Hungary*, 2004.

[25] M. Hohenwarter and K. Jones. Ways of Linking Geometry and Algebra: The Case of GeoGebra. *Proceedings of the British Society for Research into Learning Mathematics*, 27(3):126–131, 2007.

[26] M. Hohenwarter and Z. Lavicza. Mathematics Teacher Development with ICT: Towards an International GeoGebra Institute. *Proceedings of the British Society for Research into Learning Mathematics*, 27(3):49–54, 2007.

[27] H. Kaufmann. *Geometry Education with Augmented Reality*. PhD thesis, Vienna University of Technology, Vienna.

[28] H. Kaufmann and D. Schmalstieg. Mathematics and Geometry Education with Collaborative Augmented Reality. *Computers & Graphics*, 27(3):339–345, 2003.

[29] J.R. King and D. Schattschneider. *Geometry Turned On! Dynamic Software in Learning, Teaching, and Research*. The Mathematical Association of America, October 1997.

[30] C. Klaß. Portal 2 mit Bewegungssteuerung spielen, April 2011. URL http://www.golem.de/1104/83001.html#gg1_anchor.

[31] W. König, J. Gerken, S. Dierdorf, and H. Reiterer. Adaptive Pointing – Design and Evaluation of a Precision Enhancing Technique for Absolute Pointing Devices. *Human-Computer Interaction–INTERACT 2009*, pages 658–671, 2009.

[32] U. Kortenkamp. *Foundations of Dynamic Geometry (PhD. Thesis)*. PhD thesis, Inst. Tech. Zurich, 1999.

[33] U. Kortenkamp and J. Richter-Gebert. Making the Move: The Next Version of Cinderella. *Proceedings of the First International Congress of Mathematical Software*, pages 208–216, 2002.

[34] U. Kortenkamp and J. ürgen Richter-Gebert. Cinderella. 2–Geometrie und Physik im Dialog. *Computeralgebra-Rundbrief, Sonderheft zum Jahr der Mathematik*, pages 12–14, 2008.

[35] C. Laborde. Integration of Technology in the Design of Geometry Tasks with Cabri-Geometry. *International Journal of Computers for Mathematical Learning*, 6(3):283–317, 2002.

[36] D.F. Lohman. Spatial Ability and G. *Paper presented at the first Spearman Seminar, University of Plymouth*, July 1993.

[37] F. Mantovani. VR Learning: Potential and Challenges for the Use of 3D Environments in Education and Training. *Towards CyberPsychology: Mind, Cognitions and Society in the Internet Age*, 2001.

[38] Melonie McElhannon, Timo Helmke, and Terng Shing Chen. Thanks to the Razer Hydra, Now You're Thinking with Motion Control . Razer USA, Ltd., April 2011.

[39] B. Mergel. Instructional Design & Learning Theory. May 1998.

[40] J.L. Mohler. A Review of Spatial Ability Research. *Engineering Design Graphics Journal*, 72(2), 2009.

[41] E. Murphy. Constructivism: From Philosophy to Practice. 1997.

[42] S. Olkun. Making Connections: Improving Spatial Abilities with Engineering Drawing Activities. *International Journal for Mathematics Teach- ing and Learning*, pages 1–10, April 2003.

[43] S. Papert. Situating Constructionism. In S. Papert and I. Harel, editors, *Constructionism*, pages 1–11. Ablex Publishing Corporation, Norwood, NJ, 1991.

[44] J. Piaget. *La Construction du Réel Chez l'Enfant*. Neuchâtel: Delachaux et Niestlé, 1937.

[45] I. Poupyrev, M. Billinghurst, S. Weghorst, and T. Ichikawa. The Go-Go Interaction Technique: Non-Linear Mapping for Direct Manipulation in VR. *Proceedings of the 9th annual ACM symposium on User interface software and technology*, pages 79–80, 1996.

[46] J. Proulx. Constructivism: A Re-Equilibration and Clarification of the Concepts, and Some Potential Implications for Teaching and Pedagogy. *Radical Pedagogy*, 8(1):64–89, 2006.

[47] A. Rafi, K. Anuar, A. Samad, M. Hayati, and M. Mahadzir. Improving Spatial Ability Using a Web-Based Virtual Environment (WbVE). *Automation in Construction*, 14(6):707–715, December 2005.

[48] Herbert E. Read. *The Philosophy of Modern Art*. Faber & Faber, London, UK, 1952.

[49] G. Reitmayr and D. Schmalstieg. An Open Software Architecture for Virtual Reality Interaction. *Proceedings of the ACM symposium on Virtual reality software and technology*, pages 47–54, 2001.

[50] G. Revelle, O. Zuckerman, A. Druin, and M. Bolas. Tangible User Interfaces for Children. *CHI '05: CHI '05 extended abstracts on Human factors in computing systems*, April 2005.

[51] Rhapsodus. Wireframe 3D, March 2011. URL `http://forum.unity3d.com/threads/8814-Wireframe-3D/page3`.

[52] J. Richter-Gebert and U. Kortenkamp. *The Cinderella.2 Manual*. Working with The Interactive Geometry Software. Springer Verlag, March 2012.

[53] H. Schumann. Interaktives Modellieren im virtuellen Raum mit Cabri 3D. *LOG IN*, (133 S 55):61, 2005.

[54] R. Sträßer. Research on Dynamic Geometry Software (DGS) - An Introduction. *ZDM*, 34(3):65–65, 2002.

[55] S. Strong and R. Smith. Spatial Visualization: Fundamentals and Trends in Engineering Graphics. *Journal of Industrial Technology*, 18(1):1–6, 2001.

[56] K. Subrahmanyam and P.M. Greenfield. Effect of Video Game Practice on Spatial Skills in Girls and Boys. *Journal of applied developmental psychology*, 15(1):13–32, 1994.

[57] S. Wang, Z. Mao, C. Zeng, H. Gong, S. Li, and B. Chen. A New Method of Virtual Reality Based on Unity3D. *Geoinformatics, 2010 18th International Conference on*, pages 1–5, 2010.

[58] K.R. Wanzel, S.J. Hamstra, M.F. Caminiti, D.J. Anastakis, E.D. Grober, and R.K. Reznick. Visual-Spatial Ability Correlates with Efficiency of Hand Motion and Successful Surgical Performance. *Surgery*, 134(5):750–757, 2003.

[59] H.B. Yilmaz. On the Development and Measurement of Spatial Ability. *International Electronic Journal of Elementary Education*, 1 (2), 2009.