OULUN SEUDUN
AMMATTIKORKEAKOULU

Olli Koskenranta

MANIPULATING 3D OBJECTS WITH GAZE AND HAND GESTURES

MANIPULATING 3D OBJECTS WITH GAZE AND HAND GESTURES

Olli Koskenranta
Bachelor's thesis
April 2012
Information Technology and
Telecommunications
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Information Technology and Telecommunications, Software Development

Author: Olli Koskenranta
Title of thesis: Manipulating 3D Objects with Gaze and Hand Gestures
Supervisor: Timo Vainio (OUAS), Jarkko Vatjus-Anttila (CIE)
Term and year when the thesis was submitted: Spring 2012
Pages + appendices: 31 + 2

Gesture-based interaction in consumer electronics is becoming more popular these days, for example, when playing games with Microsoft Kinect, PlayStation 3 Move and Nintendo Wii. The objective of this thesis was to find out how to use gaze and hand gestures for manipulating objects in a 3D space for the best user experience possible.

This thesis was made at the University of Oulu, Center for Internet Excellence and was a part of the research project “Chiru”. The goal was to research and produce user interface techniques for handling 3D objects, and create a user interface for testing and calibrating the gestures.

The physical tools used were a hand held accelerometer-gyroscope and a camera for the gaze tracking. The work was done on Linux-based Ubuntu 11.04 and the gestures were implemented on an open source 3D platform RealXtend.

The main result was a customizable UI made in JavaScript for testing the gestures and finding problems in their usability as well as solutions for improvement.

Keywords: 3D, Gaze, Gesture, Manipulation, Object, Interface, Eye, Tracking, RealXtend

CONTENTS

TERMS AND ABBREVIATIONS

3D	Three-dimensional
UI	User Interface
EC	Entity-Component
LED	Light Emitting Diode
C++	General purpose programming language
Qt	Cross-platform application and UI framework
JavaScript	Scripting language
Python	Scripting language
QAction	Class in Qt for abstract user interface action
Frustum	Bottom part of a solid cone or pyramid formed by cutting off the top
RealXtend	Open source platform for 3D Internet

1 INTRODUCTION

Gesture-based interaction in consumer electronics is becoming more popular these days, for example, in playing games with Microsoft Kinect, PlayStation 3 Move and Nintendo Wii. Microsoft Kinect uses a camera to track the movement of the user whereas PlayStation 3 Move uses a camera and a hand held controller to track the movement of the controller. Nintendo Wii uses only a hand held controller. If implemented properly gestures are found to be intuitive and natural interaction methods for controlling programs and devices with a low learning curve (8). Once users learn how to use gestures to access different services they have found them to be enjoyable (11).

The goal of this thesis was to examine gesture handling in object manipulations, and implement a customizable user interface (UI) into an open source 3D platform called RealXtend. This thesis was done as a part of the research project "Chiru", which studies future UI techniques. Combining the gaze with hand gestures was one of the research questions which lead to the topic of this thesis. The UI was used to test and calibrate the gestures. The devices used in the implementation were an accelerometer-gyroscope for the hand gestures, a camera for tracking the gaze on the screen and a laptop. The work was done on Ubuntu 11.04.

The required interactions were handling the camera for navigating in a 3D scene, focusing on objects, grabbing them, manipulating their position and rotation and releasing them after the manipulation. These interactions were considered primitive, which, after a successful implementation, could be used as the basis of building more complex user interfaces.

2 DEVICES AND APPLICATIONS

This chapter describes the devices, applications and programming languages used in this thesis. The physical setup consisted of a camera, four infrared LEDs (Light Emitting Diodes) for improving the lightning conditions for the camera, a handheld sensor and a laptop. The camera and the LEDs were integrated into the laptop. Figure 1 shows the laptop with the camera, the LEDs and the program RealXtend running on it.

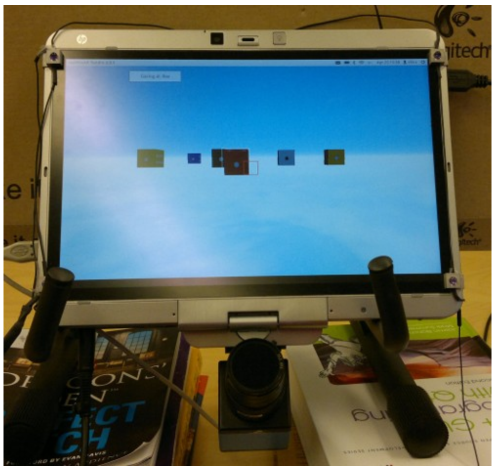


FIGURE 1. The camera (on the bottom) and the LEDs integrated into the laptop with the program RealXtend running on it

2.1 Camera

The selected camera used for the gaze tracking was Imaging Source DMK 31AU03. Its video format is 1024x768 pixels with Y800 color format (3). The Y800 color format is an 8-bit monochrome format where every pixel is represented by one byte. Monochrome means the frames captured consist of one color or shades of one color (2). The frame rate of the camera was 30 frames per second. These specifications were considered adequate to achieve an accurate enough implementation for the gaze tracking in the start of the project.

2.2 Accelerometer-gyroscope

The sensor used for the hand gestures was ATR-Promotions WAA-010. It is a small wireless sensor with a 3-axis acceleration sensor, a 3-axis gyro sensor and a 3-axis geomagnetic sensor. It contains a Bluetooth transmitter for sending data (14). Only the acceleration and gyro sensor data was used in this thesis. The geomagnetic sensor works as a compass and its functionality was not needed. Figures 2 and 3 show the hand device and the way it is attached to the hand.

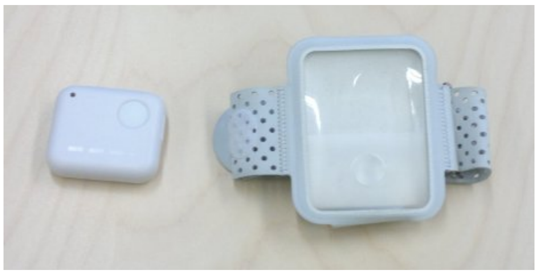


FIGURE 2. Picture of the accelerometer-gyroscope and its container

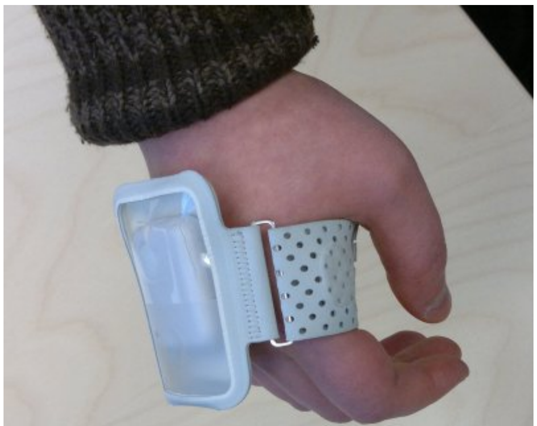


FIGURE 3. Picture of the accelometer-gyroscope in its container attached to the hand

2.3 Laptop

The laptop used for running the setup was Hewlett-Packard EliteBook 2760p. It has a 12.5” HD+ LED screen with 1366x786 resolution, Intel i5-2410M dual-core processor, Intel HD Graphics 3000, a multi-touch screen and the option to fold it into a tablet-like shape. The calculation power of the Core i5 processor was considered sufficient for this project and the laptop resembled a tablet device with a touch screen, which made it the most interesting device type for the scope of the project Chiru, since the project focuses on mobile solutions in its research.

2.4 Operating System

All the work and testing was done on Ubuntu 11.04. Ubuntu is an open source Linux-based operating system (13). Its development tools and the portability of RealXtend to Ubuntu was a good base for this project.

2.5 RealXtend

RealXtend is an open source 3D platform founded in Oulu, Finland. It has been in development since 2007. RealXtend began as a collaboration between several small companies aiming to develop a common technology base that they can use in different application fields, such as virtual worlds, video games and educational applications (1). RealXtend is programmed with C++/Qt.

2.5.1 Entity-Components

RealXtend is built entirely using the entity-component (EC) model. The entity-component model is about creating components for (game) entities instead of relying on a deep class hierarchy. Using deep class hierarchies, even simple objects can contain a large amount of useless functionality, which can affect the performance of a program in a negative manner. Instead of having these class hierarchies, the entity-component model is about separating functionality into individual components, which are independent on one another (4).

In the entity-component model, if one wants to create an entity, for example a rock, one does not use a pre-created rock class, which inherits all the required classes. Instead, one creates an entity and adds the required components for a rock, such as EC_Placeable (entity has a position), EC_Mesh (entity has a mesh for visual presentation) and EC_Rigid (entity has a solid body). Figure 4 shows example code of creating an entity with components. The EC_Script component adds support for the use of scripts in the entity and, in this case, the script reference is set to a JavaScript file "simpleavatar.js". The scripts can be used for controlling the entity and adding more functionality to it. EC_Placeable contains position data of the entity in the scene. EC_AnimationController allows animations for the entity.

```
function serverHandleUserConnected(connectionID, userconnection)
{
    var avatarEntity = scene.CreateEntity(scene.NextFreeId(),
                       ["EC_Script", "EC_Placeable", "EC_AnimationController"]);
    avatarEntity.Name = "Avatar" + connectionID;
    avatarEntity.Description = userconnection.GetProperty("username");
    avatarEntity.script.ref = "simpleavatar.js";

    // Set random starting position for avatar
    var transform = avatarEntity.placeable.transform;
    transform.pos.x = (Math.random() - 0.5) * avatar_area_size + avatar_area_x;
    transform.pos.y = (Math.random() - 0.5) * avatar_area_size + avatar_area_y;
    transform.pos.z = avatar_area_z;
    avatarEntity.placeable.transform = transform;
}
```

FIGURE 4. An example of JavaScript source code, which creates a new avatar entity when a user connects to a server and attaches several components to it (1)

2.5.2 Script support

RealXtend supports scripts created with JavaScript and Python. Using scripts allow the developers to add functionality into a program without modifying the core application source code. Recompiling the program is not needed when the core source code is not modified, which allows the developers to quickly test and change the scripts, if they are not working as intended. Also, if a user does not want use a script, s/he can simply disable it before running the program.

JavaScript is one of the most common scripting languages for adding functionality into programs in the game industry (6). This was the reason for adding the JavaScript support in RealXtend. JavaScript was also used in this thesis as a fast development and an implementation method.

2.5.3 RealXtend architecture

The architecture of RealXtend is based on a core, which is the kernel of the program. Modules extend the functionality of the core without altering it. The modules can be independent on other modules or they can have dependencies on other modules or third party libraries. The modules can also be disabled if their functionality is not needed.

2.6 OGRE

For rendering 3D scenes, RealXtend uses OGRE (Object-Oriented Graphics Rendering Engine), which is an open source 3D graphics engine (7). OGRE is implemented in RealXtend as an extension module, called OgreRenderingModule, and some of its functionality is encapsulated in entity-components, such as EC_Camera and EC_Mesh.

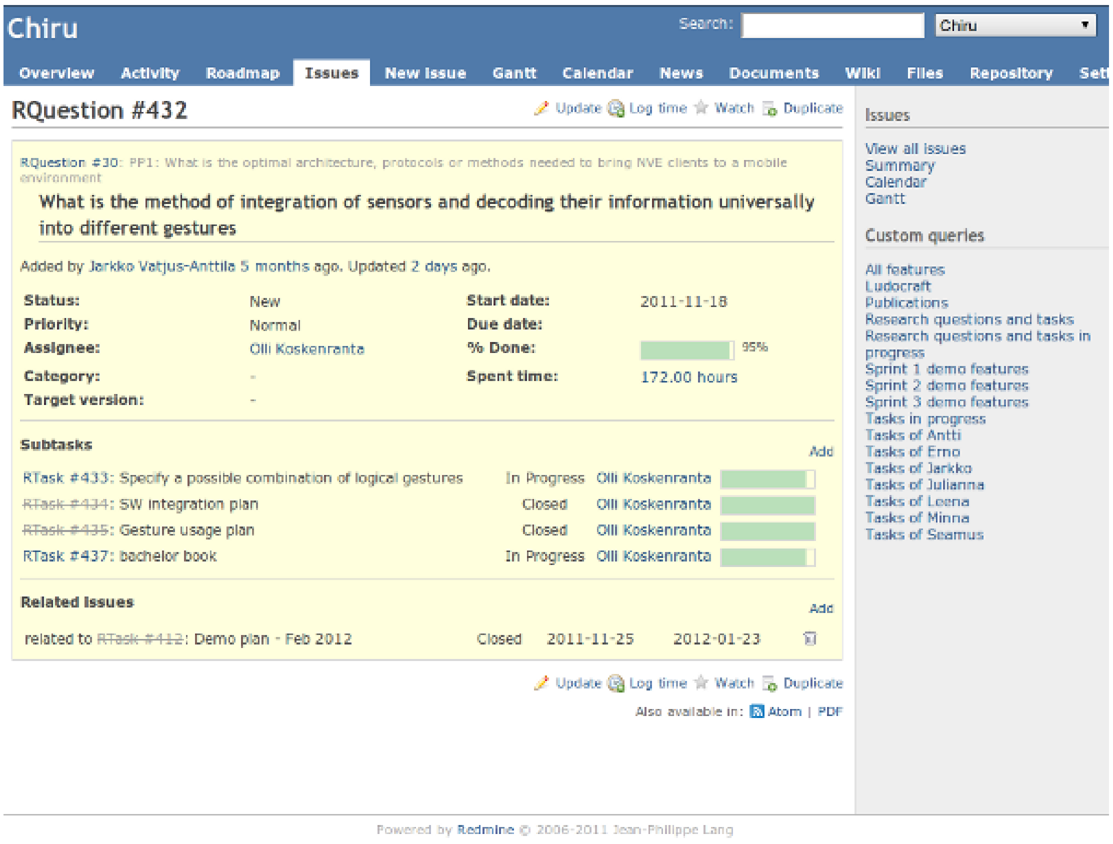
2.7 Git

Git was the version control program used in this project. Git is an open source version control system for managing the repository of the files of a project (5).

2.8 Redmine

Redmine is a project management web application (12) and it was used in this project. Relevant documents, files and templates were stored into Redmine. It was also used to store instructions on how to use the other tools in the project.

The main purpose of Redmine was to issue tasks, also called tickets, to the personnel in the project. The tickets contained the task description, deadlines and other relevant information. The person, who the ticket was assigned to, updated the task with notes and used hours. Figure 5 includes a picture of a ticket in Redmine.


Chiru
Search:
Chiru
Overview Activity Roadmap Issues New Issue Gantt Calendar News Documents Wiki Files Repository Set
Update Log time Watch Duplicate

RQuestion #432

RQuestion #30: PP1: What is the optimal architecture, protocols or methods needed to bring NVE clients to a mobile environment

What is the method of integration of sensors and decoding their information universally into different gestures

Added by Jarkko Vatjus-Anttila 5 months ago. Updated 2 days ago.

Status:	New	**Start date:**	2011-11-18
Priority:	Normal	**Due date:**	
Assignee:	Olli Koskenranta	**% Done:**	95%
Category:	-	**Spent time:**	172.00 hours
Target version:	-		

Subtasks Add

RTask #433: Specify a possible combination of logical gestures	In Progress	Olli Koskenranta
~~RTask #434~~: SW integration plan	Closed	Olli Koskenranta
~~RTask #435~~: Gesture usage plan	Closed	Olli Koskenranta
RTask #437: bachelor book	In Progress	Olli Koskenranta

Related issues Add

related to ~~RTask #412~~: Demo plan - Feb 2012	Closed	2011-11-25	2012-01-23

Also available in: Atom | PDF

Powered by Redmine © 2006-2011 Jean-Philippe Lang

FIGURE 5. An example of a ticket in Redmine

3 REQUIREMENTS FOR INTERACTION

This chapter covers the required user interface interactions and their explanations. These requirements cover the mechanisms for object selection, manipulation and controlling the camera.

3.1 Object selection

Selecting an object from the virtual 3D space is the first step in object manipulation. Selecting an object means that it is ready for an actual manipulation. Once selected, the object has to be released to end the manipulation.

3.1.1 Focusing on an object

A user has to know what object s/he is about to select. Providing a visual cue as to which object is being focused on is considered a standard, such as a label or a tool tip box that appears or changes color (2). The focusing was to be done with the gaze tracking, i.e. the object which the user was looking at was selected and then highlighted.

3.1.2 Grabbing an object

After focusing on the target object, the user needs to be able to grab it to start the manipulation. The focusing and grabbing worked as a two phase validation method to start the actual manipulation. This was predicted to be a good method to prevent accidental manipulations. The grabbing was to be done with a hand gesture.

3.1.3 Releasing an object

To end the manipulation, the user has to have a way to release an object. Releasing was to be done with a hand gesture.

3.2 Object manipulation

After an object has been selected, the user has to be able to rotate and move it. Rotating and moving was to be done with hand gestures.

3.2.1 Rotating an object

An object has to be rotated around the X- and Y-axes to cover all the possible rotation positions. The Z-axis rotation was not part of the implementation, because the implementation of the hand device only recognized two axes.

3.2.2 Moving an object

Object has to be moved along the X- and Y-axes. This was to be done with hand gestures. The Z-axis rotation was not part of the implementation, because the implementation of the hand device only recognized two axes. The X- and Y-axes were sufficient for testing the gestures' usability.

3.3 Camera handling

A camera is used to navigate in the 3D scene. For navigation, moving and rotating the camera is required. This was to be done with both the gaze tracking and hand gestures.

3.3.1 Rotating the camera

Rotating the camera means turning it without changing its actual position. This was to be done using the gaze tracking.

3.3.2 Moving the camera

The camera has to be able to move forward, backward and sideways. This was to be done with hand gestures.

4 IMPLEMENTATION OF THE INTERACTIONS

This chapter covers the implementation of the handling of the gestures. Sending of the gaze and hand gesture data and their implementation was outside of the scope of this thesis, and therefore their implementation is explained only briefly. The scope of this thesis was to handle object manipulations with gestures received from external peripherals.

4.1 JavaScript

RealXtend supports JavaScript for adding functionality into the program, and the advantages of using a script are explained in section 3.2.2. Using JavaScript was chosen for this implementation because it does not require modifying of the core source code and because the functionality of the manipulations was for specific use instead of general functionality.

4.2 Gaze tracking and hand gestures

The data flow between RealXtend, the gaze and the hand gesture device is demonstrated in Figure 6.

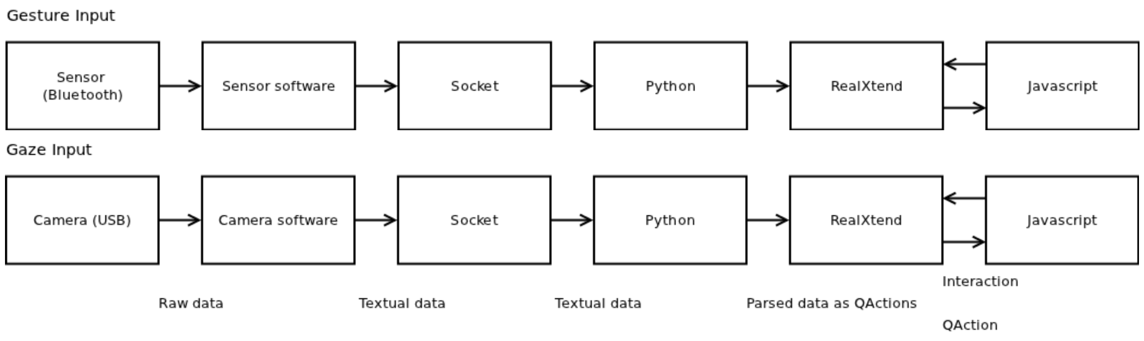


FIGURE 6. Data flow of gaze and gesture input. The bottom row describes the contents of the data in the corresponding arrows.

Both input methods used the same data flow. Their software opened and sent data into a socket, which was parsed by a python script, which sent the parsed information as QActions(10) to an entity in a RealXtend scene. The parameters of the QAction included the execution type (server or client), the name of the action and its parameters, the name of the scene and the name of the entity, which the action was sent to. These actions were then caught in JavaScript and their parameters were used to handle the required tasks.

4.2.1 Gesture parameters

The gaze input sent out the X- and Y-coordinates of the gaze on the screen. The gesture input sent out two kinds of parameters: unique gesture types (grasp, release, switch) and continuous information about the rotation of the hand (pitch and roll) in values approximately between -90 and 90. Table 1 contains the available gestures and their uses.

TABLE 1. Gestures and their uses

Gesture	**Use**	**Related parameters**
Gaze (Eyes)	Rotate the camera and move the cursor to focus on objects.	XY-coordinates between 0,0 and the screen width, screen height.
Pitch (Hand)	Rotate or move the object forward or backward. Move the camera forward or backward.	Floating-point number between -90 and 90.
Roll (Hand)	Rotate or move the object left or right. Strafe the camera left or right.	Floating-point number between -90 and 90.
Grasp gesture (Hand)	Select the object, if no other object is selected.	None
Release gesture (Hand)	Release an object, if an object is selected.	None
Switch gesture (Hand)	If in manipulation mode, switch the mode between rotation and movement mode.	None

4.3 Object selection

The object selection consisted of three phases: focusing, grabbing and releasing, and in the program code they were implemented as a simple state machine. Figure 7 demonstrates the states. See Appendix 1 for the JavaScript function for the conditions for allowing to select an entity.

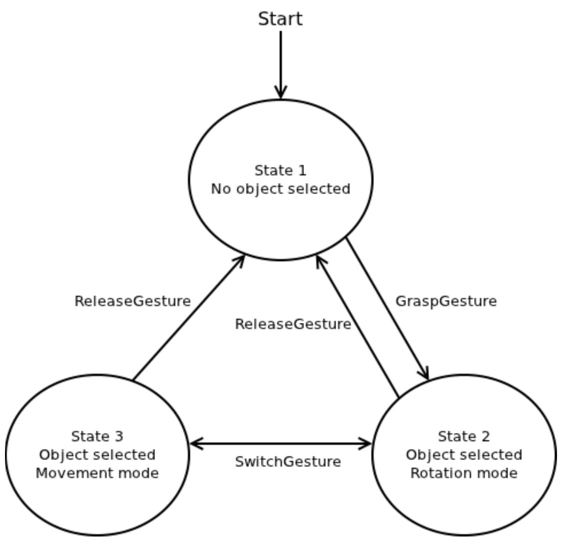


FIGURE 7. State machine of the object manipulation program

The default state (State 1) handled the camera manipulation algorithms (camera was movable and turnable) and the object selection algorithm. If the conditions for selecting an object were met and a grasp gesture was detected, the state would change to State 2.

State 2 handled the object manipulation algorithm for rotation. If a switch gesture was detected, the state would change to State 3. State 3 handled the object manipulation algorithm for movement and the algorithm for rotating the camera. The switch gesture would change the state back to State 2. Going back to State 1 from either of State 2 or State 3 was done with a release gesture.

Going straight to State 3 (moving the object) was not allowed, because it might had caused an accidental movement of the object without the user realizing what was happening. Rotating an object by accident was not as major of a manipulation as movement to cause a big difference in the state of the object.

4.3.1 Focusing on an object

Focusing was done using the gaze parameters. RealXtend has implemented two methods from OGRE for selecting entities in a 3D scene: Raycast and FrustumQuery. Raycast means shooting a ray from the camera through the viewport to the given coordinates on the screen (x, y), and it returns an entity if it hits one. FrustumQuery works in a similar fashion, but instead of a ray it shoots a customsized rectangle and returns a list of the hit entities. A depiction of a raycast is shown in Figure 8.

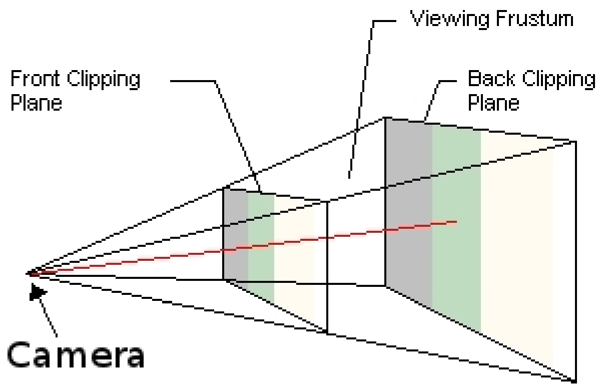


FIGURE 8. A depiction of a single raycast. Red line is the cast ray. Picture modified from (9)

FrustumQuery was chosen for the implementation because the gaze coordinates were not as precise as using a mouse would have been. A red rectangle was drawn on the viewport for the user to see where the gaze was, and it acted as a cursor. The color of the cursor changed depending on the current state of the object manipulation. When the gaze hit an object, it drew the bounding box on the object to show the user which object was being focused on.

4.3.2 Grabbing an object

Grabbing of an object was done with a hand gesture. If an object was being focused on and the user made a quick forward rotation with the hand, an object would be grabbed and ready for manipulation. The cursor (the rectangle to show the gaze) would turn green when an object was grabbed.

4.3.3 Releasing an object

Releasing an object was done with doing a shake-like move with the hand. In other words, it required to rotate one's hand left and right very fast.

4.4 Object manipulation

Once the object is grabbed, the manipulation is possible. The start of the manipulation was indicated to the user with the color of the cursor. The color of the cursor turned green (manipulation has started) from red (not manipulating anything).

4.4.1 Rotating an object

Rotating the object was done in two ways. One way was rotating the object in the direct relation to the angle of the hand. This limited the possible positions the object could be rotated into, because of the limitations of the hand. Hence, this method was found inadequate. Another way was using a toggled rotation. In other words, the object started rotating in the direction the hand was rotated, and the speed of the rotation depended on the steepness of the angle of the hand.

4.4.2 Moving an object

Moving the object was a separate mode from rotating the object. The rotation and movement could not happen simultaneously because of the limited number of available gestures. This was probably also more convenient for the user, because only one action was meant to be handled simultaneously. The object would move left and right and forward and backward along the world axes depending on the rotation of the hand.

4.4.3 Changing manipulation mode

The changing between the rotation and the movement modes was done with a switch gesture. The switch gesture was the same gesture as the grab gesture, since the grab gesture was available for use after the object was grabbed.

4.5 Camera handling

When no object was selected, the camera was movable. The camera movement was locked during the rotation of an object to prevent the object from getting lost from the view of the user. When an object was being moved, the camera was turnable to allow the object to be moved to a position outside the view.

4.5.1 Rotating the camera

The camera was rotated when the gaze entered on the edges of the screen. The size of the edge was an adjustable variable in the JavaScript. The speed of the rotation depended on how near the edge was from the gaze. When gazed on the very edge of the screen, the speed of the rotation would be at the maximum, and slower when the gaze was closer to the center. 25% of the size of the screen from each edge of the screen was used as the area to cause the camera to start rotating. Hence, the area size was independent on the used display resolution and was tied more tightly to the physical dimensions, which are, in the end, more relevant for the gaze tracking. A depiction of the use of the screen can be seen in Figure 9.

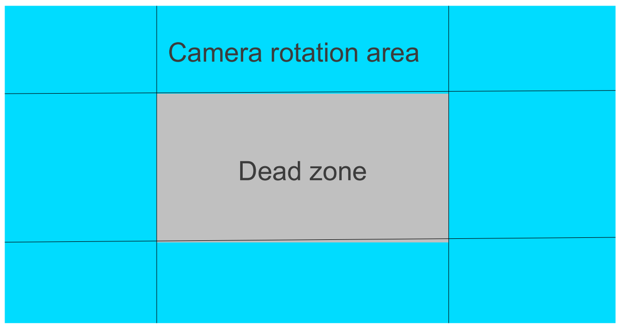


FIGURE 9. A depiction of the use of the screen for turning the camera

4.5.2 Moving the camera

The camera was moved forward and backward by rotating the hand forward and backward. When the hand was rotated left and right, the camera would strafe left and right accordingly.

5 TESTING

This chapter covers the testing done with the UI and the changes made to the UI if a change was needed. A user test was also done to compare the gaze and hand gestures to the use of a touch screen.

5.1 Test setup

The test setup can be seen in Figure 10. The camera is attached to the bottom of the laptop. There are four infrared LEDs near each corner of the laptop for improving the lightning conditions for the camera. In this case, the hand gesture device is attached to the right hand of the user. The hand device could be used with either hand.

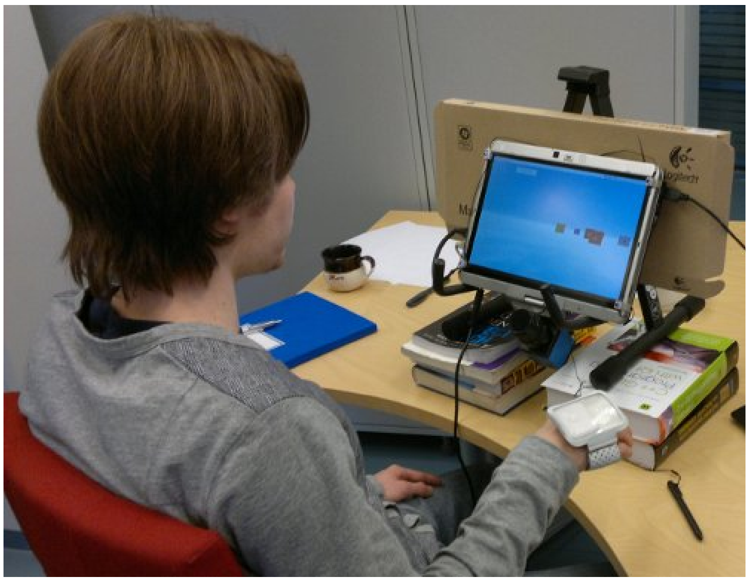


FIGURE 10. Test setup

5.2 Gaze

The initial testing of the gaze was done by one person to see how the first implementation worked.

5.2.1 Accuracy problems and solutions

The gaze coordinates were sent 30 times per second. The camera was not able to predict the gaze coordinates very accurately, and therefore the cursor movement was not very stable. This was caused by the lack of precision on the camera, the algorithm which calculated the coordinates and the disturbances caused by moving the head, since the camera was attached on the laptop and not on the head of the user.

To improve the performance, a setting was added to adjust the amount of points, which the UI would use to calculate the average coordinates of the gaze. Increasing the amount of points for calculating the coordinates improved the performance and caused the cursor to become more stable, which made the focusing on the objects easier. The downside was that it also caused the reaction time to the gaze increase. See Appendix 2 for the JavaScript function for handling the received gaze coordinates.

The average of 30 points was found to be rather accurate, and since the camera sent 30 points per second, it caused the gaze to lag behind a maximum of one second. This was not a problem when focusing on the objects but it made turning the camera more difficult as the camera kept on turning even if the user switched his gaze to the center of the screen.

To solve the camera rotation problem, a change was made. The turning of the camera was done with the current coordinates but leaving the focusing to be done with the average coordinates.

5.2.2 Turning the camera

The display area for turning the camera (25% of the screen size from each edge of the screen) was found to work well and no complaints were received.

5.2.3 Using glasses with gaze tracking

If a person was using glasses, it interfered with the camera tracking the eye. The gaze tracking was found to be unusable if a person was using glasses. This was caused by the reflection of the glasses, which resulted in the algorithm to miscalculate the correct eye positions.

5.3 Hand gestures

The first implementation of the hand gestures was tested by two persons.

5.3.1 Manipulating with gestures

The first implementation of the manipulations consisted of using three steps for adjusting the speed of movement and rotation. This implementation proved to be insufficient for accurate manipulations. The manipulation algorithms were changed to use a stepless manipulation to rotate and move the objects using the angle of the hand with a multiplier as the default speed of the movement or the rotation. Figure 11 shows the JavaScript calculation code of the manipulation speed, and it was found to be reasonable.

```
movement_speed = Math.pow((roll_angle/100),7)
```

FIGURE 11. JavaScript code for calculating the speed of the manipulation

5.3.2 Manipulation modes

An addition to the cursor color was made to show the manipulation mode. When the mode was changed to the movement, the cursor turned from green to purple and vice versa. Figure 12 shows the different colors of the cursor.

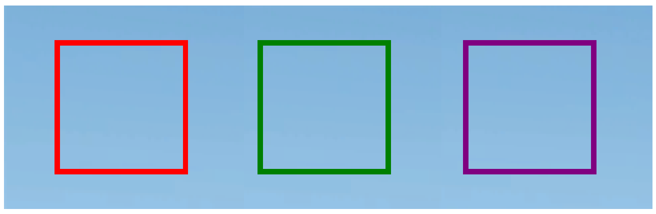


FIGURE 12. The cursor of the gaze and its different colors

5.3.3 Movement change

A change to the working of movement was needed. Moving the object originally along the world axes made it difficult to control the objects in a natural way.

Thus, the movement was changed from using the world axes to the local axes of the camera.

5.4 Comparison to using a touch screen with a user test

The possibility to use the touch screen to perform the same manipulations as with the gaze and hand gestures was needed as a comparison for the user tests.

5.4.1 Manipulations with a touch screen

A finger replaced the gaze and hand gestures when using a touch screen. Grabbing and releasing an object was done with a tap and hold gesture. Switching the manipulation mode was done with a double tap. Moving the camera was done with a swipe to the wanted direction. Moving the finger to the wanted direction in the rotation and movement modes caused the object to move and rotate into that direction. Table 2 shows the used events, their uses and their related parameters.

TABLE 2. Touch events and their uses

Touch event	**Use**	**Related parameters**
Move	Move the cursor or adjust the rotation or the position of an object.	XY-coordinates between 0,0 and the screen width, screen height.
Release	Stop moving the camera, unless sweep was detected.	None
Tap and hold	Grab or release an object.	XY-coordinates between 0,0 and the screen width, screen height.
Double tap	Switch the manipulation mode from rotation to movement and vice versa.	None
Swipe	Start moving the camera in the swipe direction.	Direction

5.4.2 User test

The user test consisted of one scene with six dice-objects and a goal area. The camera was stationary, because it was noticed that if the camera was movable, it made it difficult to perform the manipulations because the camera movement gestures got mixed with the manipulation gestures. The goal for the test was to move the six dices into the goal area and rotate them into the positions which displayed their values from one to six. Figure 13 shows an overall view of the test scene.

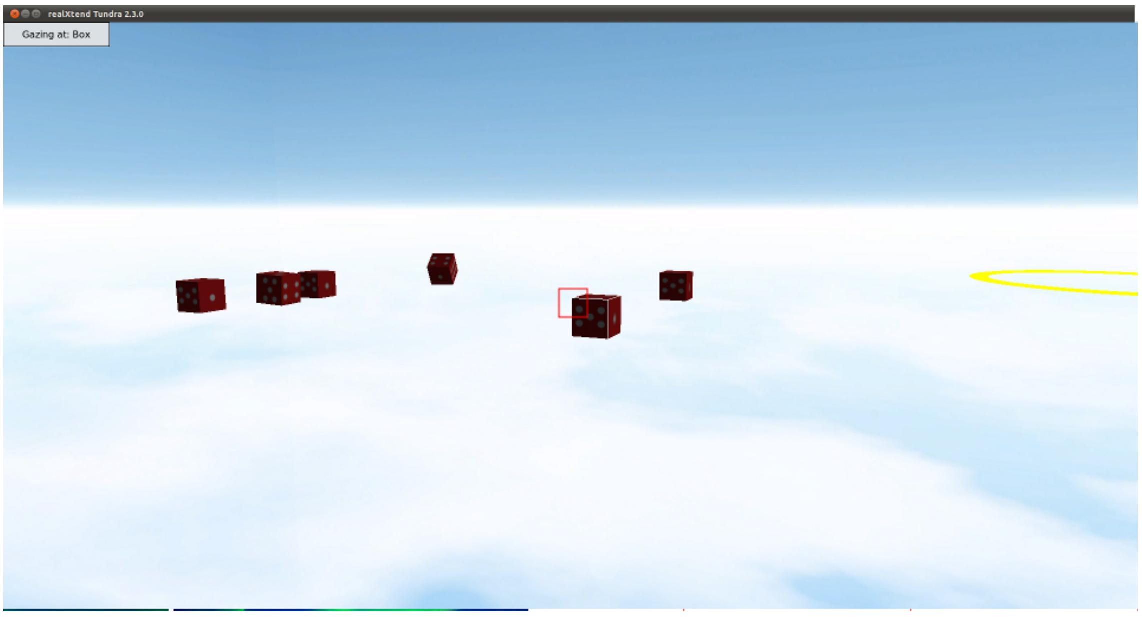


FIGURE 13. The test scene with six dice objects. The goal area can be seen on the right side of the screen

5.4.3 User test results

Nine users participated in the test. The gaze was found to be the most interesting interaction method, even though the accuracy of tracking the eye coordinates was not perfect. Overall, using the gaze and hand gestures was more interesting than using the touch screen, even if the manipulations generally proved to be more difficult with them instead of using the touch screen. None of the users had previous experience of the gaze tracking, but some had used Nintendo Wii or Microsoft Kinect. Most users had used touch screen based devices before.

Generally, even though the UI and the performance of the gaze tracking and hand gestures were suboptimal, the users thought that they would be interested in using a similar system in controlling different programs, if the user experience was improved.

6 CONCLUSION AND DISCUSSION

The main objective was to create an UI for the gaze tracking and hand gestures. This was done mainly with JavaScript in the RealXtend environment.

The available hand gestures were rotating the hand left, right, forward and backward. The unique gestures used were grasp, switch and release gesture.

The required manipulations were selecting, deselecting, rotating and moving the objects and the camera with the gaze and hand gestures. All the required manipulations were successfully implemented.

The difficulties in the implementation were related to the inaccuracy of the gaze coordinates and the limited amount of available hand gestures. Increasing the accuracy for the gaze and the amount of the hand gestures would bring more variation possibilities in the implementation.

The user tests showed that the users were interested in new ways of interaction methods, especially using the gaze, and this field is promising for future studies.

Working on this thesis taught me valuable information on different user interaction methods and their future possibilities. On the technical side, using scripts to implement new functionality to a program was a new area for me and it gave me experience on the benefits of the script support on programs. Working on a 3D environment was also a new experience for me and getting familiar with the functionality of a 3D scene was also useful, because 3D programs are becoming more popular.

The JavaScript file for the gaze and hand gesture manipulations can be found in the repository of Chiru at the following URL:

https://github.com/Chiru/naali/blob/tundra2/bin/jsmodules/startup/gazetracking.js

REFERENCES

1. Alatalo T. An Entity-Component Model for Extensible Virtual Worlds. Internet Computing, IEEE 2011 sept.-oct.;15(5):30.
2. Definition of monochrome at Merriam-Webster dictionary. Available at: http://www.merriam-webster.com/dictionary/monochrome/. Date of data acquisition 27. April 2012
3. DMK 31AU03. Available at: http://www.theimagingsource.com/en_US/products/cameras/usb-ccd-mono/dmk31au03/. Date of data acquisition 27. April 2012
4. Entity-Components. Available at: http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/. Date of data acquisition 27. April 2012
5. Git. Available at http://git-scm.com/. Date of data acquisition 27. April 2012
6. How to Make a Video Game from Scratch. Available a http://www.wikihow.com/Make-a-Video-Game-from-Scratch/. Date of data acquisition 27. April 2012
7. Junker, G. Pro OGRE 3D Programming (Expert's Voice in Open Source). the United States of America: Apress; 2006.
8. Lee, S. C. – Bohao Li – Starner, T. AirTouch: Synchronizing In-air Hand Gesture and On-body Tactile Feedback to Augment Mobile Gesture Interaction. In Anonymous Wearable Computers (ISWC), 2011 15th Annual International Symposium on. (). , 2011, 3-10.
9. Picture of a depiction of the viewport. Available at http://i.msdn.microsoft.com/dynimg/IC123215.jpg/. Date of data acquisition 27. April
10. QAction. Available at http://qt-project.org/doc/qt-4.8/qaction.html. Date of data acquisition 27. April 2012

11. Rahman, A. M. – Hossain, M. A. – Parra, J. – El Saddik, A. Motion-path based gesture interaction with smart home services. In Anonymous Proceedings of the 17th ACM international conference on Multimedia. (Beijing, China,). ACM, New York, NY, USA, 2009, 761-764.

12. Redmine. Available at http://www.redmine.org/. Date of data acquisition 27. April 2012

13. Ubuntu. Available at: http://www.ubuntu.com/. Date of data acquisition 27. April 2012

14. WAA-010. Available at: http://www.atr-p.com/sensor10.html/. Date of data acquisition 27. April 2012

APPENDICES

```
function EntitySelection()
{
    //Get the camera entity
    var cameraEnt = scene.GetEntityByName("FreeLookCamera");
    if (!cameraEnt)
        return;
    var camera_position = cameraEnt.placeable.transform.pos;
    //Do a FrustumQuery to the gaze coordinates
    var closest_entity = scene.ogre.FrustumQuery(gaze_x - rect_size, gaze_y - rect_size,
gaze_x + rect_size, gaze_y + rect_size, camera_position);

    if (closest_entity)
    {
        if (closest_entity.GetComponent("EC_Placeable"))
        {
            //Check whether a new entity is being focused on
            if (closest_entity != last_raycast_entity && last_raycast_entity)
            {
                var placeable = last_raycast_entity.placeable;
                placeable.drawDebug = false;
                last_raycast_entity.placeable = placeable;
            }
            //Display the bounding box for the entity being focused on
            last_raycast_entity = closest_entity;
            var placeable = last_raycast_entity.placeable;
            placeable.drawDebug = true;
            last_raycast_entity.placeable = placeable;
            if (use_statusbutton)
                        statusbutton.text = "Gazing at: " + last_raycast_entity.Name();
        }
    }
    else
    {
        //If no entity is detected on the gaze coordinates remove the bounding box
        if (last_raycast_entity)
        {
            var placeable = last_raycast_entity.placeable;
            placeable.drawDebug = false;
            last_raycast_entity.placeable = placeable;
        }
    }
}
```

```
function GazeCoordinates(x, y)
{
    if (!scene)
        return;
    if (gaze_counter < amount_of_points) //Wait until the table for points is populated
    {
        gaze_points_x.unshift(parseInt(x));
        gaze_points_y.unshift(parseInt(y));
        gaze_counter += 1;
    }
    else
    {
        //Start calculating the average position from the points
        //Add new coordinates to the beginning of the table
        gaze_points_x.unshift(parseInt(x));
        gaze_points_y.unshift(parseInt(y));
        //Remove the oldest coordinates from the table
        gaze_points_x.pop();
        gaze_points_y.pop();
        //Calculate the sum of the coordinates
        for (index = 0; index < amount_of_points; index++)
        {
            gaze_sum_x += gaze_points_x[index];
            gaze_sum_y += gaze_points_y[index];
        }
        //Calculate the average of the coordinates
        gaze_average_x = gaze_sum_x / amount_of_points;
        gaze_average_y = gaze_sum_y / amount_of_points;
        //Reset values of the sum
        gaze_sum_x = 0;
        gaze_sum_y = 0;
        //Set gaze coordinates to the average values
        gaze_x = gaze_average_x;
        gaze_y = gaze_average_y;
    }

    //Calculate the delta of the gaze coordinates from the center of the screen.
    //The delta is used for calculating camera rotation.
    //Notice that the first coordinates for the rotation are used and not the average.
    delta_center_x = (screen_width / 2) - gaze_points_x[0];
    delta_center_y = (screen_height / 2) - gaze_points_y[0];
    //Move the cursor to match the gaze coordinates
    if (gaze_x > screen_width)
        proxy.x = screen_width - rect_size;
    else if (gaze_x < 0)
        proxy.x = 0;
    else
        proxy.x = gaze_x - rect_size;

    if (gaze_y > screen_height)
        proxy.y = screen_height - rect_size;
    else if (gaze_y < 0)
        proxy_y = 0;
    else
        proxy.y = gaze_y - rect_size;

    proxy2.x = gaze_x - 10;
    proxy2.y = gaze_y - 10;
}
```