

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing



Kinect 3D Editor

Semester project

Krzysztof Lis

liskr@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Philipp Brandes, Laura Peer
Prof. Dr. Roger Wattenhofer

June 14, 2016

Acknowledgements

I would like to sincerely thank those whose contributions helped me complete this project:

- Epic Games and other Unreal Engine developers, for providing the Unreal Engine 4 framework [1] and associated graphical assets,
- Microsoft Zürich, for providing a Kinect v2 motion tracking sensor,
- Opaque Multimedia, for providing the *Kinect 4 Unreal* plugin [2],
- Tom Looman, for resources about creating outlines in Unreal Engine 4 [3],
- Georg Bachmeier, Pascal Bissig, Michał Borkowski, Philipp Brandes, Sebastian Brandt, Dr. Christian Decker, Manuel Eichelberger, Klaus-Tycho Förster, Pankaj Khanchandani, Michael König, Magdalena Molenda, Michalina Pacholska, Laura Peer, David Stolz for feedback and participation in the experiments.

Abstract

We have attempted to evaluate the usefulness of an application user interface based on body movement and gestures, which were detected by a Kinect v2 body tracking sensor. We have created a prototype application with this interface: a 3D editor, in which a user can move virtual objects in a 3D scene. The editor's performance was evaluated by a group of users completing a set of predefined tasks with the program. Our research revealed that while a purely movement-controlled application is possible, it lacks in precision and convenience, and an approach with additional hardware and adding alternative input methods is preferable.

Contents

Acknowledgements i

Abstract ii

1 Introduction 1

2 Kinect 2 overview 2

2.1 Kinect for Windows SDK 2.0 2

2.1.1 Skeletal tracking 3

2.1.2 Hand states 3

2.1.3 Gesture recognition 3

2.2 libfreenect2 3

3 Design 4

3.1 Overview 4

3.2 Calibration and cursors 6

3.3 Object selection 7

3.4 Object movement 7

3.5 Discrete inputs and mode switching 10

4 Implementation 13

5 Evaluation 14

5.1 Challenges 14

5.2 Testing and results 15

6 Conclusion 22

Bibliography 23

A Installation instructions A-1

Chapter 1

Introduction

The goal of this project is to evaluate the usefulness of the *Kinect v2* motion tracking sensor in user interface tasks involving spatial controls in 3D environments.

Moving virtual 3D objects is an important task in application related to graphics, games or animation. An object placed freely in 3D space has 6 degrees of freedom: 3 vector components of location, 3 angles of rotation (each around another perpendicular axis). However the computer mouse has only 2 degrees of freedom, as it can move on a flat surface. Therefore, editing 3D scenes with a traditional interface requires a significant amount of repetitive actions.

An interface based on body position and movement offers more degrees of freedom, and therefore it may be more efficient in the described situation.

Chapter 2

Kinect 2 overview

The body movement user interface described in this work is based on a motion tracking sensor *Kinect v2*. This sensor is capable of collecting the following data [4]:

- high resolution (1920×1080 pixels) video image at 30 Hz refresh rate,
- depth sensing - a 512×424 pixel video stream where pixel values correspond to the distance of the visible object from the sensor, the range of the depth sensor is approximately 5 meters,
- infra-red light emitter and camera, capable of producing a video stream despite lack of other light,
- microphone, which can be utilized for voice control.

A USB 3.0 port and specialized software, described below, is required for a computer to connect to a Kinect v2 sensor.

2.1 Kinect for Windows SDK 2.0

The recommended way to utilize the Kinect v2 sensor in a program is to use the *Kinect for Windows 2.0 SDK* which can be obtained through the manufacturer's website [5]. The SDK provides access to the data produced by the sensor: video stream, depth stream, infra-red video, however can also perform further processing on those signals to utilize them for human motion tracking [6]. The SDK is only available on the Windows operating system.

Apart from the motion tracking, Kinect's depth data can be used to scan 3D objects and environments through the *Kinect Fusion* technology.

2.1.1 Skeletal tracking

The SDK processes the sensor output with machine learning algorithms to recognize human silhouettes. The person needs to be in range of the depth sensor to be tracked, multiple users can be tracked at once. The tracked body is represented by vector positions of 25 joints located around the body. A specialized tracker is used to track human faces, however this feature was not used in this project.

2.1.2 Hand states

The sensor is also capable of recognizing human hands and classifies them into the following states [7]:

- open hand,
- closed hand,
- lasso,
- not tracked or unknown.

2.1.3 Gesture recognition

The SDK includes the *Visual Gesture Builder* tool which allows the developers to specify their own gestures which will be recognized by the sensor. The developer records footage of the desired gesture and labels it, then a machine learning classifier is trained to recognize it.

2.2 libfreenect2

Alternatively, the Kinect v2 sensor can be accessed using an open source driver *libfreenect2* [8]. It does not offer the body tracking features of the SDK, however it provides the video, depth and infra-red images and is available for Linux, Windows and OSX operating systems.

Chapter 3

Design

A prototype 3D editor application has been created to evaluate the concept of body movement controls. In the editor, the user utilizes a body tracking interface to select, move and rotate objects in a virtual 3D scene.

3.1 Overview

The editor presents a view of a 3D virtual scene, containing objects which can be moved by the user. An example view of the editor is shown in Figure 3.1. In general, interaction with the editor consists of selecting an object and then moving it. The editor can be in the following states:

1. *calibration* of the motion tracking controls - user stands still and their initial body posture is recorded, this process is further described in 3.2,
2. *base state* - in this state the user's hand movements do not select or move the objects. The user can interact with the application menu or prepare their body posture to comfortably perform the next editing action.
3. *selection mode* - the user is choosing which object will be edited, this process is further described in 3.3,
4. *movement mode* - the user is moving the previously selected object in the scene, this process is further described in 3.4.

Transitions between the states are shown in Figure 3.2. The controls allowing the user to switch between modes are described in Section 3.5.

The editor is a research prototype and supports only single object selection and movement. Advanced editing features like scaling, multiple selection, copying, creation and deletion of objects are not implemented.

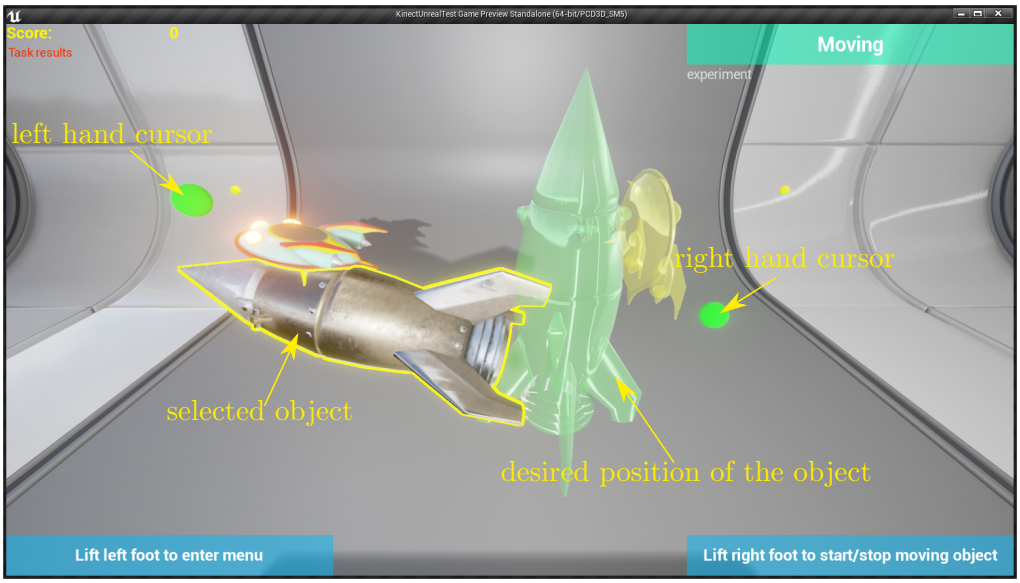


Figure 3.1: An example view of the example application. The user is currently moving the highlighted *selected object*. The cursors represent the positions of user's hands (further details in Section 3.2). The *desired position of the object* is a part of the challenge system described in Chapter 5.

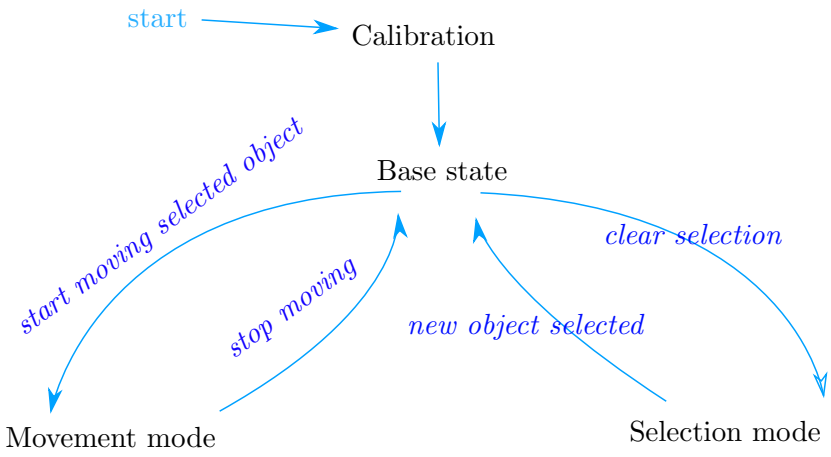


Figure 3.2: Available states of the example application and transitions between them. The details of these states are described in Section 3.1.

3.2 Calibration and cursors

In the editor, the user selects and moves virtual objects by moving their hands. A Kinect motion sensor tracks the positions of user's hands. These positions and movements are then mapped onto the space of the editor's virtual scene, the details of this process will be described further in this section. Hand movements are represented with the movements of cursors, shown in Figure 3.1, providing the user with visual feedback and allowing them to interact with the objects in the scene.

The geometric configuration of the system is shown in Figure 3.3. A Kinect motion sensor tracks the user's body posture and provides the locations of their body parts in *sensor coordinate space*, characterized by the axis unit vectors $\hat{E_x}$, $\hat{E_y}$, $\hat{E_z} \in \mathbb{R}^3$. By $\vec{H_L}(t_n)$, $\vec{H_L}(t_n) \in \mathbb{R}^3$ we denote the vectors representing temporary positions of the left and right hand respectively, in sensor's coordinate system, reported by the motion sensor at program step t_n.

The values reported by the motion sensor in its coordinate system depend on the user's physical size and the place where they are standing. The operation of the interface should not depend on those factors, therefore the hand locations are mapped to the *workspace coordinate space*, (characterized by the axis unit vectors $\hat{e_x}$, $\hat{e_y}$, $\hat{e_z} \in \mathbb{R}^3$ and center of the coordinate system $\vec{c} \in \mathbb{R}^3$), which is tied to the user's location, as seen in Figure 3.3.

The parameters of the workspace coordinate system are determined during the *calibration phase*, which takes place upon the program start. During that phase, the user is asked to stand still with their hands at stretched to the sides. The hand position vectors $\vec{H_L}$, $\vec{H_L}$ in sensor coordinates are recorded and averaged over the calibration time, yielding the averages $\vec{H_{L0}}$, $\vec{H_{R0}} \in \mathbb{R}^3$.

Once the calibration data is available, the values of workspace coordinate axis vectors in sensor coordinates are calculated. It is assumed that the sensor will be placed horizontally, so the workspace z axes are the same as the sensor space axis:

$$\hat{e_z} = \hat{E_z}.$$

The y workspace axis is the direction from the user's left hand to their right hand:

$$\hat{e_y} = \frac{\vec{H_{R0}} - \vec{H_{L0}}}{||\vec{H_{R0}} - \vec{H_{L0}}||}.$$

The x workspace axis is determined by the properties of Cartesian coordinate system, it must be normal to y and z axes and the direction is determined by a cross product of y and z axes:

$$\hat{e_x} = \hat{e_y} \times \hat{e_z}.$$

The center $\vec{c}$ of the workspace coordinate system is the center point between user's hands:

$$\vec{c} = \frac{\vec{H_{R0}} + \vec{H_{L0}}}{2}.$$

To obtain normalized coordinates, independent of user's physical dimensions, the vectors are divided by the workspace size w, which is equal to the average distance between the user's hands during calibration:

$$w = ||\vec{H_{R0}} - \vec{H_{L0}}||.$$

By $\vec{h_L}(t_n)$, $\vec{h_L}(t_n)$ we denote the vectors representing temporary positions of the left and right hand respectively, in workspace coordinate system, calculated at program step t_n. These values are obtained by transforming the hand position vectors $\vec{H_L}(t_n)$, $\vec{H_L}(t_n)$ from the sensor coordinate space with axis vectors $\hat{E_x}$, $\hat{E_y}$, $\hat{E_z}$ and center $\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$ to the workspace coordinate space with axis vectors $\hat{e_x}$, $\hat{e_y}$, $\hat{e_z}$ and center $\vec{c}$:

$$\vec{h_L}(t) = \frac{1}{w} \begin{pmatrix} \hat{e_x}^T \\ \hat{e_y}^T \\ \hat{e_z}^T \end{pmatrix} (\vec{H_L}(t) - \vec{c}),$$

$$\vec{h_R}(t) = \frac{1}{w} \begin{pmatrix} \hat{e_x}^T \\ \hat{e_y}^T \\ \hat{e_z}^T \end{pmatrix} (\vec{H_R}(t) - \vec{c}).$$

The hand position vectors in workspace coordinates are then used to display the hand cursors (as seen in Figure 3.1) and allow user's interaction with the objects in the scene.

3.3 Object selection

The object to be edited is chosen in the *selection mode*. Selection is performed using the right hand cursor, as seen in Figure 3.4. An object becomes selected when it overlaps with the cursor, the event is detected by a physics collision detection algorithm. To select an object, the user needs to move the cursor into it, as if they wanted to touch it.

3.4 Object movement

When the editor is in the *movement mode*, the selected object's location and rotation is being edited by the user's input. The control mechanism aims to be

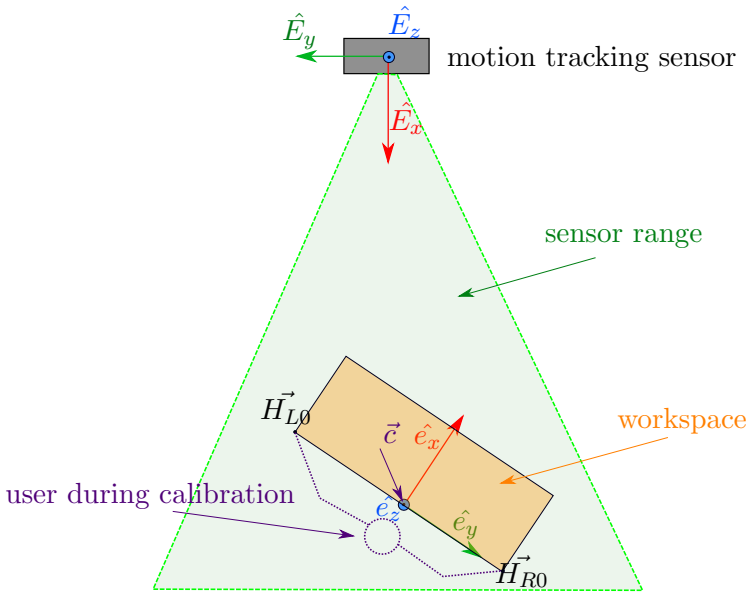


Figure 3.3: The relative locations of the sensor coordinate system (axes $\hat{E}_x$, $\hat{E}_y$, $\hat{E}_z$) and the workspace coordinate system (axes $\hat{e}_x$, $\hat{e}_y$, $\hat{e}_z$, center at $\vec{c}$) which depends on the user's position during system calibration. $\vec{H}_{L0}$, $\vec{H}_{R0}$ are average values of the vectors representing the position of user's left and right hand respectively during the calibration phase.

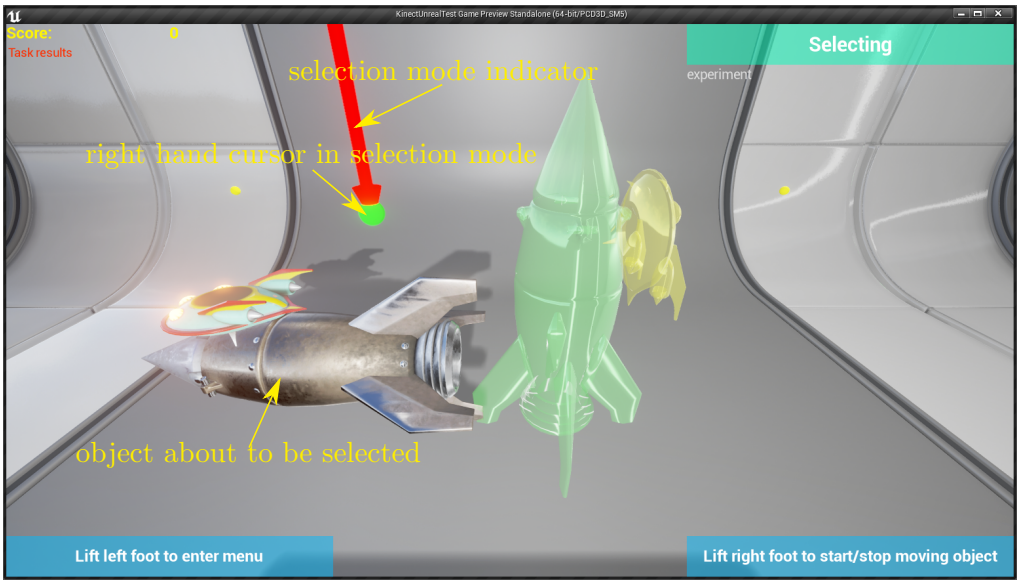


Figure 3.4: A view of the editor in selection mode. An object will be selected when the hand cursor, highlighted with the selection mode indicator, collides with the object.

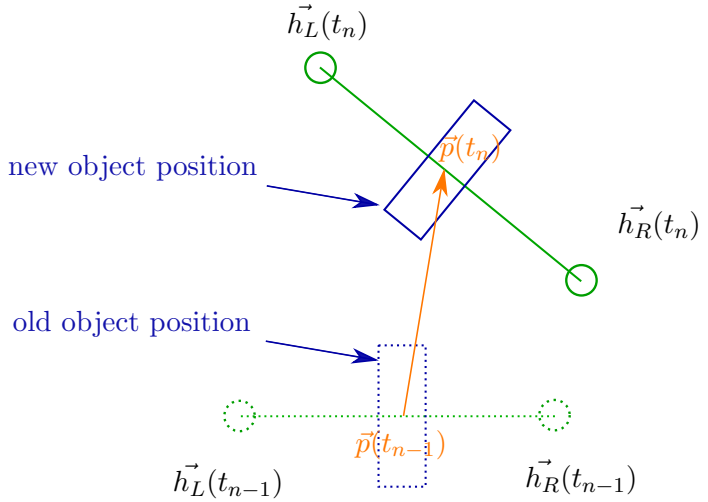


Figure 3.5: An object is being moved and rotated by the user's hand movements. The diagram shows the changes happening in a single step of the program, transition from step t_{n-1} to step t_n. $\vec{h_L}(t_{n-1})$, $\vec{h_L}(t_{n-1})$ - hand positions in the previous frame. $\vec{h_L}(t_n)$, $\vec{h_L}(t_n)$ - current hand positions. $\vec{p}(t_{n-1})$ - object's position in previous frame, $\vec{p}(t_n)$ - object's position after the current movement step.

intuitive by acting as if the object was held on a stick between the user's hands, as shown in Figure 3.5.

An object's placement in a 3D scene is described by its location $\vec{p}$ and its rotation matrix R. These values change in time as the object is moved and rotated by the user's hand movements. We denote the center point between the user's hands as $\vec{h_c}$:

$$\vec{h_c}(t) = \frac{1}{2}(\vec{h_L}(t) + \vec{h_R}(t)),$$

where $\vec{h_L}, \vec{h_R}$ are hand position vectors in workspace coordinates as defined in Section 3.2. The change in the object's position $\vec{p}$ is determined by the movement of the center point between the user's hands:

$$\vec{p}(t_n) = \vec{p}(t_{n-1}) + (\vec{h_c}(t_n) - \vec{h_c}(t_{n-1})),$$

where: $\vec{p}(t_{n-1})$ is the location of the selected object in previous program step, $\vec{p}(t_n)$ is the current location of the selected object after the change.

Likewise, the object is rotated to follow the rotation of the vector between user's hands. We denote normalized direction between the hand positions as h_d:

$$\vec{h_d}(t) = \frac{\vec{h_R}(t) - \vec{h_L}(t)}{||\vec{h_R}(t) - \vec{h_L}(t)||},$$

and the rotation matrix that transforms the previous value of $\vec{h_d}$ to the current one as Q:

$$\vec{h_d}(t_n) = Q(t_n)\vec{h_d}(t_{n-1}).$$

Then the change in selected object's rotation R is given by:

$$R(t_n) = Q(t_n)R(t_{n-1}),$$

where: $R(t_{n-1})$ is the rotation matrix of the selected object in the previous step and $R(t_n)$ is the rotation matrix of the selected object after the change.

3.5 Discrete inputs and mode switching

In addition to the spatial controls used in selection and movement of objects, the application needs discrete inputs (equivalent to button presses on traditional input devices) to preform actions like entering and exiting movement mode, choosing to select a new object, exiting the program.

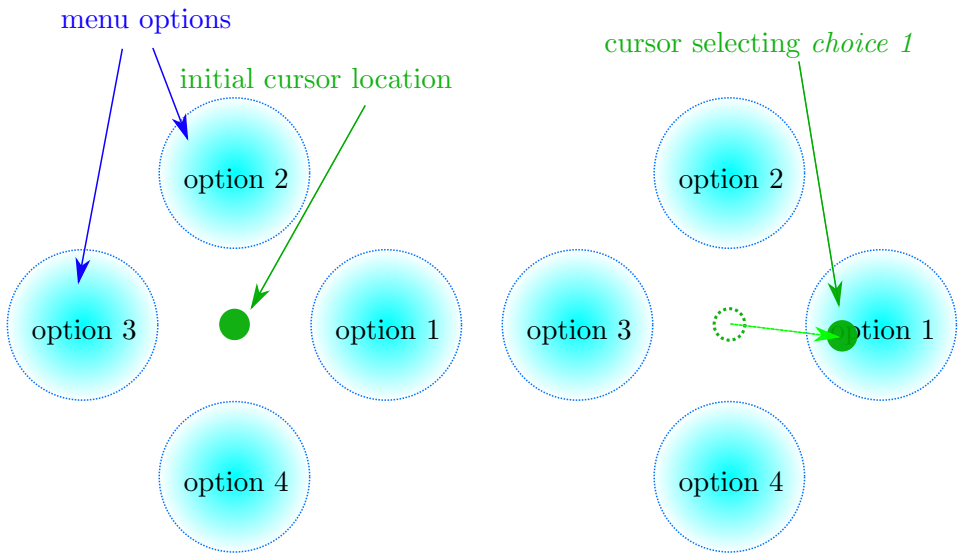
We have designed a gesture menu, which allows the user to choose one of 4 presented actions with the movement of their hand. The menu is shown in Figure 3.6. This menu is 2 dimensional and displayed directly on the screen, not as part of the virtual scene. Upon activation, the menu is displayed with the cursor in the center, as in Figure 3.6a. Then the user moves their hand to select an option. Moving the hand up, down, left or right selects a different choice. An option is selected when the cursor moves close enough to the option label, as seen in Figure 3.6b. In the example application, *entering the selection mode* is performed via the gesture menu.

However, not all inputs can be done with the hand movements. In the editor's movement mode, all hand movements influence the selected object's location and rotation. Therefore, the input exiting movement mode cannot rely on the hand positions. We have two alternative solutions to that problem:

1. *Hand states* - as discussed in Subsection 2.1.2, the Kinect sensor can distinguish different states of the hands. In this input variant we decided that the user will perform editing with open hands, and the movement mode will be toggled by closing the right hand - closing the hand once will activate the mode, closing again will exit it. Closing the left hand will open the gesture menu.

   A debouncing mechanism will prevent false-positive detections of the gesture, by requiring the hand to remain closed for a short time, around 0.15 seconds.

   However practical tests have shown that the hand state detection is highly unreliable, the closed hand state is sometimes falsely detected and the



(a) The initial state of the menu and initial location of the cursor.

(b) The user moved their hand to the right, and selected *choice 1*.

Figure 3.6: The gesture menu user interface, presenting the user with a choice of 4 options to select. The cursor is controlled by the movement of the user's left hand. One of the options will be selected when the cursor is moved close enough to the option label.

sensor will not detect the hand is closed if only the back of the hand is exposed to the camera. Also it is inconvenient for the user to forget to keep the hands open and switch the mode by unintentionally closing their hand. Due to these problems, an alternative was developed.

2. *Other body parts* - the Kinect sensor tracks positions of all body parts, so their movements can also be utilized as an input mechanism. In the example application, the movement mode is toggled upon raising the right knee, and the gesture menu is activated upon rising the left knee.

   A debouncing mechanism will prevent false-positive detections of the gesture, by requiring the knee to remain raised for a short time, around 0.15 seconds.

   This input is more reliably detected than hand states and prevents false-positive detections. However, many users have difficulty with raising their legs without moving their hands - which causes them to unintentionally move the selected object when exiting movement mode.

Voice control could be utilized in place of those discrete input mechanisms. However, we decided that this project should focus on body movement controls. Integrating voice control could be a topic for further research.

Chapter 4

Implementation

The example application was build using the Unreal Engine 4 framework, which provided the 3D graphics, user interface, vector and matrix operations functions. The application logic was developed using the C++ language and Unreal Engine's Blueprint Visual Scripting system. The graphical assets used are example assets provided by Unreal Engine and can be utilized freely in projects developed with that engine. A view of the example application is shown in Figure 3.1.

The outline used to highlight the selected object was created with the resources provided by Tom Looman [3].

The *Kinect 4 Unreal* plugin by Opaque Multimedia was used for a simple integration of the Kinect sensor with Unreal Engine-based application.

To allow later analysis of the actions performed by users within the editor, all actions were stored in a log and saved in JSON format upon closing the program.

Chapter 5

Evaluation

5.1 Challenges

In order to measure the effectiveness of the interface, a set of scored challenge levels has been created. In each level the user is presented with a different virtual scene containing a set of movable objects. Each level involves a set of tasks to perform, every task is to move and rotate an object to match the desired location and rotation of the object which is indicated by a semi-transparent ghost of the object. An example task can be seen in Figure 5.2. The task is completed when the object has been placed in the correct location and rotated accordingly, as shown in Figure 5.3. The example application contains four levels:

1. Practice level: Figure 5.1, contains no tasks and its purpose is to let the user accustom themselves to the interface,
2. Level 1: Figure 5.2, contains one task involving simple movement and rotation of a single object,
3. Level 2: Figure 5.4, contains three tasks, each involving moving a sphere, the rotations are not scored.
4. Level 3: Figure 5.5, contains two tasks involving a complex rotations of the objects.

In the example application, once the user decides the object is placed accurately enough, they can proceed to the next level by choosing an option in the gesture menu, which described in Section 3.5. Each task is scored according to the following criteria:

1. $\Delta\vec{p} = (\Delta p_x, \Delta p_y, \Delta p_z)$ - difference between the desired location vector of the object and the actual location vector representing the position where the user placed the object. From $\Delta\vec{p}$, the following quantities are calculated:



Figure 5.1: The practice level of the example application. It contains no tasks and its purpose is to let the user practice interaction with the interface.

2. $\Delta\alpha$ - smallest angle of rotation required to rotate the object to the desired rotation specified in the task. If the objects moved are spheres, this quantity is not measured.

3. Δt - time of completion, measured from the first time when an object was being selected (first start of selection mode) to the last time an object was moved (last exit from movement mode). As the user can edit objects in any order, the total time used for all tasks in a level is measured and then divided by the number of tasks in level to obtain per-task time.

These metrics, as well as a complete log of user activities in the program (selections and movements of objects), are saved by the editor in a file to enable further analysis.

5.2 Testing and results

A test was conducted in which 12 participants were asked to perform the tasks in the example editor. Each participant performed the tasks twice, in order to measure whether the results improve as the user gains more experience with the interface. The quantities described in Section 5.1 have been recorded.

The averages of values obtained in the test are shown in Table 5.1. The values of distances and vector coordinates are given in units used by the Unreal

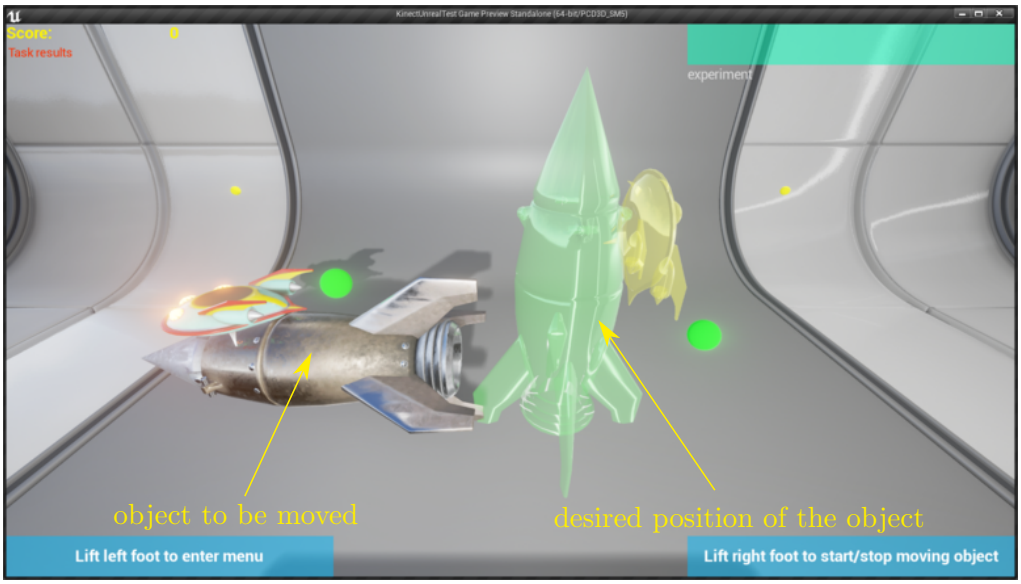


Figure 5.2: The first challenge level, contains one task involving simple movement and rotation of a single object The desired position of the object is indicated with a semi-transparent ghost of the object.

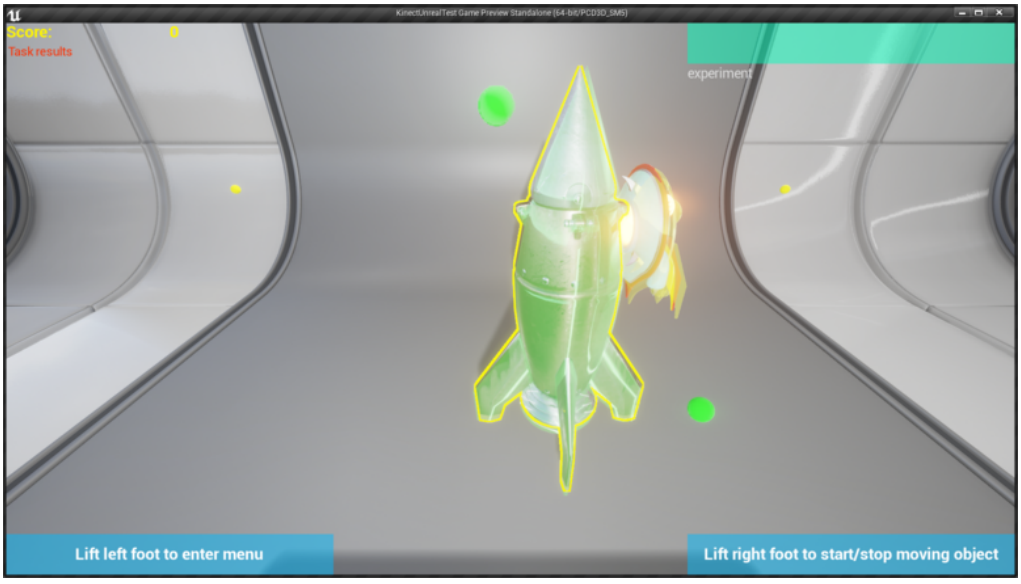


Figure 5.3: View of the completed version of the first level shown in Figure 5.2. The object has been placed inside the desired position indicator.

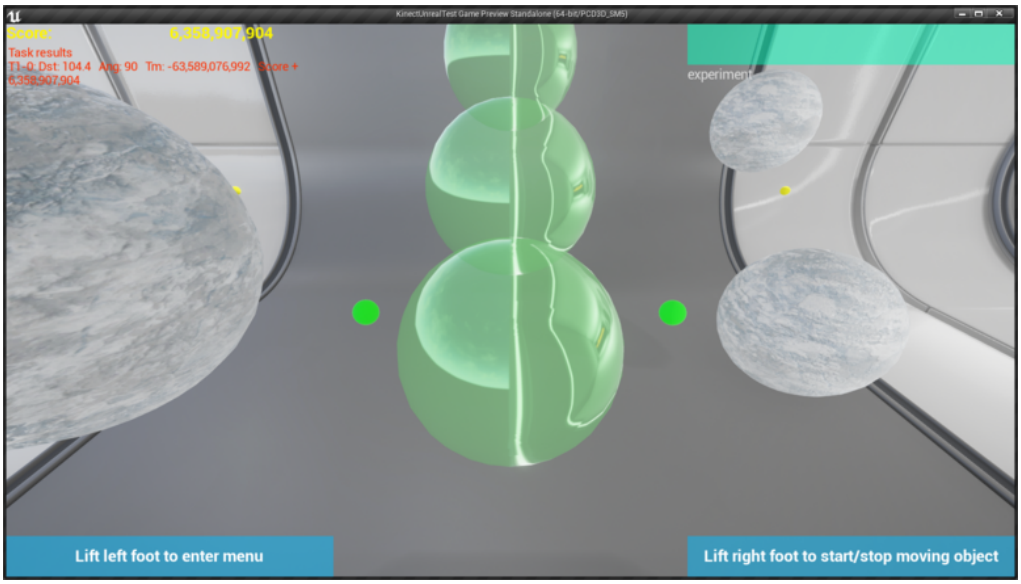


Figure 5.4: The second challenge level, contains three tasks, each involving moving a sphere in order to construct a snowman. Since the objects are spheres, rotation is not measured.



Figure 5.5: The third challenge level, contains two tasks, each involving a complex rotation of the object.

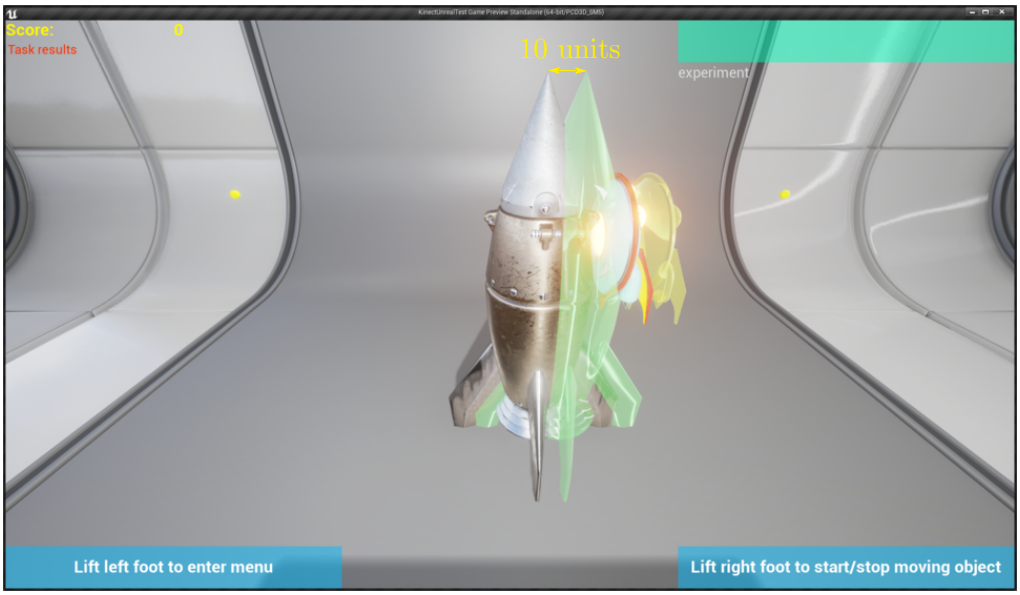


Figure 5.6: Size of Unreal Engine's distance units in comparison to the size of objects used in the example editor. The distance between the tips of the rockets is equal to 10 Unreal Engine units.

Engine 4 - the meaning of that units in terms of the virtual scene is shown in Figure 5.6.

The not insignificant values of rotation and location errors ($\Delta\alpha$, $||\Delta\vec{p}||$), indicate that the body tracking controls have a rather limited precision. During testing, noise and random hand position shifts have been observed in the data provided by the motion tracking sensor.

The times required by the users to place an object precisely in the scene are in the order of tens of seconds, quite long compared to the times expected from traditional user interface actions, like manipulating objects with a mouse. One reason for that could be the inconvenient mode-switching mechanism involving leg movement. Another contributing factor may be the limited precision of the body tracking controls, as users spent time doing small movements trying to achieve most accurate placement despite the input noise.

The location error $\Delta\vec{p}$ is significantly higher in the x axis (Δp_x value) - direction forward from the user - than in y and z axes (Δp_y, Δp_z values) - directions right, left, up and down from the user. The reason for that is most likely that the participants had difficulties with accurately judging the depth of an object in the scene, to which the x coordinate corresponds. The ability to move the camera in the scene could solve that problem, however moving the camera would require increasing the complexity of the controls, for example addition of another mode. In an advanced editor program camera controls would

Level	1, simple rotation	2, moving spheres	3, complex rotation
Δt [s]	64.9 ± 41.1	26.9 ± 12.7	43.4 ± 12.0
$\Delta\alpha$ [^{0}]	15.8 ± 8.8	not applicable	12.0 ± 7.0
$\|\|\Delta\vec{p}\|\|$	12.8 ± 10.4	6.8 ± 5.4	11.9 ± 9.4
$\|\Delta p_x\|$	11.5 ± 10.6	5.5 ± 5.3	9.4 ± 7.9
$\|\Delta p_y\|$	3.0 ± 2.7	1.5 ± 1.6	4.1 ± 4.2
$\|\Delta p_z\|$	2.5 ± 1.8	2.4 ± 2.4	4.6 ± 4.8

Table 5.1: Average values of the metrics achieved by users participating in the challenges. Δt - time of task completion. $\Delta\alpha$ - angle between the achieved object rotation and desired rotation, in degrees. $\Delta\vec{p} = (\Delta p_x, \Delta p_y, \Delta p_z)$ - difference between the desired location vector of the object and the actual object location vector achieved by the user, in Unreal Engine 4 units, as seen in Figure 5.6. The uncertainty values are equal to one standard deviation.

regardless be necessary to edit bigger scenes.

Each user performed the challenges twice. The results obtained in both attempts have been compared to determine if performance improves with training. The results are shown in Table 5.2. The ratios of values achieved in the second attempt to values achieved in the first attempt have been depicted on histograms: completion time in Figure 5.7, angle error in Figure 5.8, location error in Figure 5.9.

A slight majority of the users needed more time Δt in the second attempt, which is unexpected as then they were more accustomed to the editor interface. It is possible that in the second attempt they focused more on precision. No dominant direction of change in rotation angle error $\Delta\alpha$ has been observed. It is likely that the tested control method does not allow a higher precision in rotation editing. The location error $||\Delta\vec{p}||$ has improved for a slight majority of users. It is possible that the object location control method allows for higher precision but requires practice from the user. The observed differences were not very significant, and the number of participants (12) was low, therefore, no strong conclusions can be drawn from the analysis of the results.

Quantity	New value lower (better)	New value higher (worse)
completion time Δt	28	44
angle error $\Delta\alpha$	17	18
distance $\|\|\Delta\vec{p}\|\|$	44	28

Table 5.2: Comparison of challenge results between two consecutive attempts made by the same users. The *new value lower* column contains counts of occurrences (singular task completions) such that the value of the metric in the second attempt is higher than in the first attempt. *New value higher* is the number of occurrences such that the second value is higher. The higher the value of a metric, the worse the quality of the task completion.

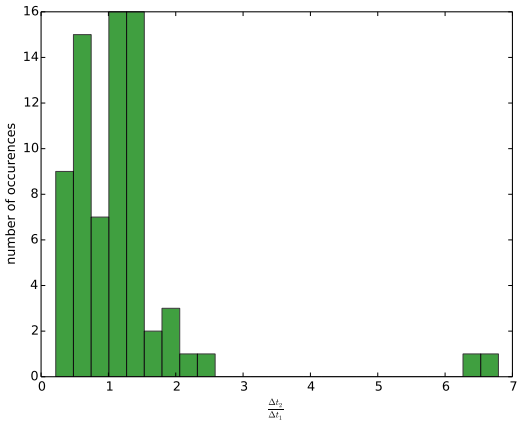


Figure 5.7: Comparison of measured challenge task completion times achieved by the test participants in two consecutive attempts. A histogram of ratios of Δt_2 - times measured at second attempt, to Δt_1 - times measured at first attempt.

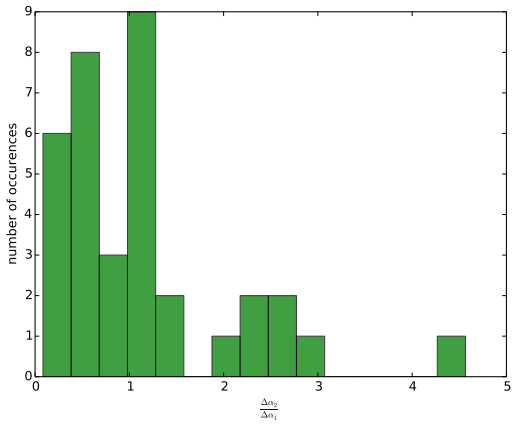


Figure 5.8: Comparison of measured angles between the desired object rotation and the object rotation achieved by the challenge test participants in two consecutive attempts. A histogram of ratios of $\Delta\alpha_2$ - angles measured at second attempt, to $\Delta\alpha_1$ - angles measured at first attempt.

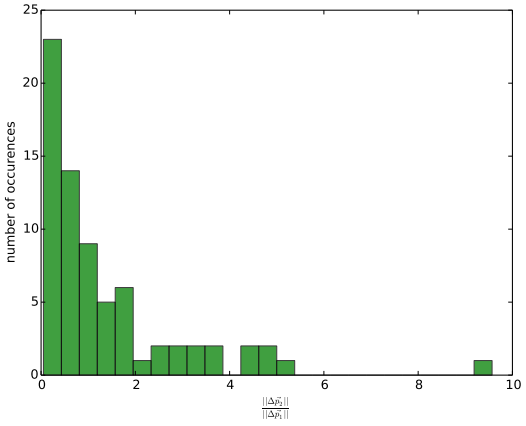


Figure 5.9: Comparison of measured distances between the desired object location and the object location achieved by the challenge test participants in two consecutive attempts. A histogram of ratios of $||\Delta\vec{p_2}||$ - distances measured at second attempt, to $||\Delta\vec{p_1}||$ - distances measured at first attempt.

Chapter 6

Conclusion

In the course of this project, we successfully developed a working prototype of a 3D scene editor, however it is not ready for non-research usage. We have reached the following conclusions:

- The Kinect v2 sensor reliably tracks the human body and location of the body parts, however the data has limited precision and suffers from noise. The hand state detection is less reliable and some hand states are only detected if the hand is correctly aligned towards the sensor.
- Body movement and gesture controls are not enough to efficiently control a complex application. Methods of providing discrete inputs, like button presses, are required. Gestures can fill that role, but not when body movement controls are needed for a different part of the interface (in out case, the gesture menu cannot be used during object movement). Applications based on body movement controls would benefit from using a hand-held device with buttons, like the *Wii Remote* device.
- Controlling the application simultaneously with several body parts, like both hands and one leg in the example application, is inconvenient for the user and may lead to loss of precision in control.
- The existing software tools allow for easy and efficient development of programs making use of the Kinect v2 motion tracking sensor.
- Body movement controls prevent the user from sitting close to the computer screen, therefore a big screen, a projector or head mounted display is preferred for comfortable operation.
- The body movement interface based on a Kinect v2 sensor can be used to effectively move objects in a virtual scene, however with a limited precision. While that may not be suitable for professional editor software, it could be used in games or for fast prototyping, where precision is not of utmost importance.

Bibliography

[1] Unreal Engine 4 main site. https://www.unrealengine.com

[2] Opaque Multimedia: Kinect 4 Unreal plugin. http://www.opaque.media/kinect-4-unreal

[3] Tom Looman: Multi-color outline post process in unreal engine 4. http://www.tomlooman.com/multi-color-outline-post-process-in-unreal-engine-4/ (October 2015)

[4] Microsoft: Kinect hardware. https://dev.windows.com/en-us/kinect/hardware

[5] Microsoft: Kinect for Windows SDK 2.0 - Download. https://www.microsoft.com/en-us/download/details.aspx?id=44561

[6] Microsoft: Kinect for Windows SDK 2.0 - Features. https://msdn.microsoft.com/en-us/library/dn782025.aspx

[7] Microsoft: Kinect for Windows SDK 2.0 - Body tracking. https://msdn.microsoft.com/en-us/library/dn799273.aspx

[8] libfreenect2 project site. https://github.com/OpenKinect/libfreenect2

Appendix A

Installation instructions

The following instructions describe how to install and execute the example editor application created in the course of this project.

- Install *Visual Studio 2015*, which will serve as a C++ compiler for the project.
- Install *Unreal Engine 4* [1]. Open the launcher and download version 4.10 of the engine.
- Download the project files. They should be stored in the directory `KinectUnrealTest`.
- Rename the subdirectory `Plugins` to a different name, for example `Plugins2`. This step is a necessary part of building the C++ sources of the application, because it uses the Kinect 4 Unreal, for which source code is not provided. Unreal Engine will try to build every module from source, and will abort the build process upon failing to build the plugin. The plugin is only accessed through the visual scripting interface, therefore it is not needed during C++ compilation and can be safely hidden at that step. The problem may be saved in later versions of the engine.
- Open the `KinectUnrealTest.sln` in Visual Studio and run the *Build Solution* action.
- Rename the subdirectory back to `Plugins`. It is important to do this before opening the project in Unreal Engine, as without this directory, the engine will fail to load the plugin. If the project is accidentally opened without the plugin, restart the engine. Do not save any files in that state as it may corrupt the visual scripting files.
- Open the project in Unreal Engine by launching the engine and choosing the `KinectUnrealTest.uproject` file in the dialog window.
- Connect the Kinect v2 sensor.

- In Unreal Editor, click the *Play* button to launch the program. Switch to *New Editor Window* option to display it full screen.