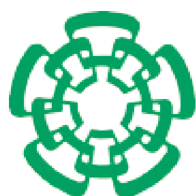

Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional

Unidad Zacatenco

**Departamento de Computación**

# Toolkit para el desarrollo de soportes de interacción multi-usuario basados en Leap Motion

Tesis que presenta

**César Adrián Ordaz Santiago**

para obtener el Grado de

**Maestro en Ciencias en Computación**

Directora de la Tesis

**Dr. Sonia Guadalupe Mendoza Chapa**

**México, D.F.** **Mayo 2015**

# Resumen

Actualmente, el desarrollo de aplicaciones de visualización digital, e.g., realidad virtual y aumentada, se encuentra en una etapa de crecimiento, gracias al advenimiento de nuevas y mejores tecnologías, las cuales son capaces de soportar los exigentes requerimientos gráficos de la visualización en 3D. Por otra parte, las personas tienen al alcance de su mano nuevos dispositivos que les permiten controlar sus aplicaciones, mediante la entrada de gestos o señas que les resultan familiares. Uno de los dispositivos de entrada gestual más actuales es Leap Motion, el cual capaz de detectar el movimiento natural de las manos del usuario.

Las investigaciones sobre Leap Motion se han limitado principalmente al soporte de interacciones entre un usuario y un dispositivo. Asimismo, dichas investigaciones se han centrado en la utilización de gestos como modalidad de entrada en aplicaciones basadas en Leap Motion. Por lo tanto, resulta interesante explorar el uso de Leap Motion en aplicaciones multi-usuario, en las cuales varios usuarios distribuidos concurren simultáneamente para realizar una tarea común o para interactuar entre ellos a través de la aplicación.

La principal contribución de este trabajo es un *toolkit* que proporciona, a los desarrolladores de aplicaciones, diversas herramientas para la identificación de los usuarios y la interpretación de las señas utilizadas para controlar el espacio compartido de la aplicación multi-usuario. Además, el *toolkit* propuesto proporciona el código de las funciones correspondientes a dichas señas, las cuales representan los datos de entrada capturados por Leap Motion. Nuestra propuesta utiliza señas en vez de gestos, los cuales están compuestos de una dirección, una duración, un sentido y, en algunos casos, otros gestos y señas, puesto que los requerimientos de procesamiento de una seña son menores que los de un gesto. De hecho, una seña solo necesita una dirección y un sentido.

Para validar el carácter genérico de nuestro *toolkit*, se han implementado diversas aplicaciones mono-usuario y multi-usuario que explotan, de manera parcial o total, las APIs ofrecidas por dicho *toolkit*. La usabilidad de estas aplicaciones ha sido evaluada con grupos de usuarios finales.

Palabras clave: entrada gestual y con señas, interacción distribuida, Leap Motion, toolkits, aplicaciones multi-usuario

# Abstract

Nowadays, the development of digital visualization applications, e.g., virtual and augmented reality, is experiencing an important growing stage, owing to the coming of new and better technologies, which are able to support the exigent graphical requirements of a 3D visualization. On the other hand, people have new devices at hand that allow them to control their applications by means of gestures or signs common to users. One of these new gestural input devices is Leap Motion, which is able to detect the natural movement of the users hands.

Research works on Leap Motion has been mainly limited to support interactions between one user and one device. In the same way, such research works have been centered on the use of gestures as an input modality in Leap Motion-based applications. Therefore, it is interesting to study the usage of Leap Motion in multi-user applications, in which several distributed users simultaneously get together, in order to carry out a common task or to interact among them through the application.

The main contribution of this work is a *toolkit* that provides application developers with several tools for the identification of users and the interpretation of the signs used to control the shared space of the multi-user application. In addition, the proposed *toolkit* provides the code of functions corresponding to such signs, which represents the input data captured by Leap Motion. Our proposal relies on signs instead of gestures, which are composed of a direction, a length, a sense and, in some cases, other gestures and signs, since the processing requirements of a sign are fewer than the ones of a gesture. In fact, a sign only needs a direction and a sense.

In order to validate the generic character of our *toolkit*, we have implemented several single-user and multi-user applications (mainly videogames) that exploit, in a partial or total way, the APIs offered by such a *toolkit*. The usability of these applications has been evaluated with groups of end-users.

Keywords: gestural and sing input, distributed interactions, Leap Motion, toolkits, multi-user applications.

# Índice general


**Resumen** I

**Abstract** III

**Índice de figuras** VII

**Índice de tablas** XI

**Índice de Algoritmos** XII

**1. Introducción** **1**
- 1.1. Contexto de la investigación . . . . . 1
  - 1.1.1. Trabajo Colaborativo Asistido por Computadora . . . . . 1
  - 1.1.2. Interacción Humano Máquina . . . . . 2
- 1.2. Justificación . . . . . 3
- 1.3. Antecedentes y motivación . . . . . 3
- 1.4. Planteamiento del problema . . . . . 4
  - 1.4.1. Propuesta de solución . . . . . 4
- 1.5. Hipótesis . . . . . 4
- 1.6. Objetivos del proyecto . . . . . 5
- 1.7. Organización de la tesis . . . . . 5

**2. Marco teórico y trabajo relacionado** **7**
- 2.1. Leap Motion . . . . . 7
  - 2.1.1. Características . . . . . 7
  - 2.1.2. API y funciones requeridas . . . . . 9
- 2.2. Trabajo relacionado . . . . . 13
  - 2.2.1. Uso de Leap Motion . . . . . 13
  - 2.2.2. Soporte de trabajo colaborativo . . . . . 15
- 2.3. Análisis comparativo . . . . . 19

**3. Análisis y diseño del *toolkit*** **21**
- 3.1. Caso de uso del *toolkit* . . . . . 23
  - 3.1.1. Casos de uso para el control de usuarios . . . . . 23
  - 3.1.2. Casos de uso para el procesamiento de dedos y vectores . . . . . 26

3.1.3. Casos de uso para la interpretación de señas . . . . . . . . . . . 28
3.2. APIs del *toolkit* . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
3.3. API de control de usuarios . . . . . . . . . . . . . . . . . . . . . . . 32
3.3.1. Función *asignarId*() . . . . . . . . . . . . . . . . . . . . . . . 33
3.3.2. Función *obtenerId*() . . . . . . . . . . . . . . . . . . . . . . . 35
3.3.3. Función *borrarId*() . . . . . . . . . . . . . . . . . . . . . . . 37
3.3.4. Función *obtenerLista*() . . . . . . . . . . . . . . . . . . . . . . 40
3.3.5. Función *contarId*() . . . . . . . . . . . . . . . . . . . . . . . 41
3.3.6. Función *verificarLista*() . . . . . . . . . . . . . . . . . . . . . 41
3.4. API de procesamiento de dedos y vectores . . . . . . . . . . . . . . . . 41
3.4.1. Función *contarDedosExtendidos*() . . . . . . . . . . . . . . . . . 42
3.4.2. Función *calcularColision*() . . . . . . . . . . . . . . . . . . . . 43
3.4.3. Función *obtenerCuadranteDedo*() . . . . . . . . . . . . . . . . . 45
3.5. API de interpretación de señas . . . . . . . . . . . . . . . . . . . . . 46
3.5.1. Función *generarSeniaSinDir*() . . . . . . . . . . . . . . . . . . . 47
3.5.2. Función *generarInfoSenia*() . . . . . . . . . . . . . . . . . . . . 50
3.5.3. Función *generarSeniaConDir*() . . . . . . . . . . . . . . . . . . . 51

**4. Implementación del *toolkit* 55**
4.1. Lenguaje de programación . . . . . . . . . . . . . . . . . . . . . . . 55
4.2. Implementación de la API de control de usuarios . . . . . . . . . . . 56
4.2.1. Función *asignarId*() . . . . . . . . . . . . . . . . . . . . . . . 57
4.2.2. Función *obtenerId*() . . . . . . . . . . . . . . . . . . . . . . . 58
4.2.3. Función *borrarId*() . . . . . . . . . . . . . . . . . . . . . . . 59
4.2.4. Función *obtenerLista*() . . . . . . . . . . . . . . . . . . . . . . 61
4.2.5. Función *contadorId*() . . . . . . . . . . . . . . . . . . . . . . . 61
4.2.6. Función *verificarLista*() . . . . . . . . . . . . . . . . . . . . . 62
4.3. Implementación de la API de procesamiento de dedos y vectores . . . 62
4.3.1. Función *contarDedosExtendidos*() . . . . . . . . . . . . . . . . . 62
4.3.2. Función *calcularColision*() . . . . . . . . . . . . . . . . . . . . 63
4.3.3. Función *obtenerCuadranteDedo*() . . . . . . . . . . . . . . . . . 64
4.4. Implementación de la API de interpretación de señas . . . . . . . . . 65
4.4.1. Función *generarSeniaSinDir*() . . . . . . . . . . . . . . . . . . . 66
4.4.2. Función generarInfoSenia() . . . . . . . . . . . . . . . . . . . . . 70
4.4.3. Función generarSeniaConDir() . . . . . . . . . . . . . . . . . . . . 71

**5. Implementación de las aplicaciones de prueba 79**
5.1. Aplicaciones mono-usuario . . . . . . . . . . . . . . . . . . . . . . . 79
5.1.1. Crear señas para Leap Motion . . . . . . . . . . . . . . . . . . . 80
5.1.2. Ratón . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
5.1.3. Rompecabezas . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83
5.2. Aplicaciones multi-usuario . . . . . . . . . . . . . . . . . . . . . . . 85
5.2.1. PingPong . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 87
5.2.2. Zombis invasores . . . . . . . . . . . . . . . . . . . . . . . . . . . 88

5.3. Características generales de las aplicaciones . . . . . 92
5.4. Pruebas con usuarios finales . . . . . 94

**6. Conclusiones y trabajo a futuro 101**
6.1. Recapitulación del problema . . . . . 101
6.2. Conclusiones y contribuciones . . . . . 101
6.3. Limitaciones . . . . . 102
6.4. Trabajo a futuro . . . . . 103

**Bibliografía 104**

# Índice de figuras

1.1. Contexto específico de investigación . . . 2
1.2. Organización de la tesis . . . 6

2.1. Mapa cartesiano de Leap Motion . . . 8
2.2. Información de un marco obtenido de la interacción con Leap Motion 8
2.3. Visualizador de Leap Motion . . . 9
2.4. Jerarquía de las clases de la API . . . 11
2.5. Huesos que componen los dedos de una mano . . . 12
2.6. Conjunto de gestos reconocidos por la API de Leap Motion . . . 13
2.7. Visualización del dispositivo Leap Motion (a) imagen infrarroja y (b) vista esquemática . . . 14
2.8. Interfaz de usuario de Leap Trainer . . . 14
2.9. Ejemplo de control de la aplicación SprBlender . . . 15
2.10. Funcionamiento de 3D Helping Hands en un ambiente colaborativo con usuarios físicamente distantes . . . 16
2.11. (a) Visualización del usuario interactuando con el sistema y (b) vista de la transformación digital del espacio de trabajo mediante el uso de información proxémica del usuario . . . 17
2.12. Proyección de los teclados virtuales sobre una superficie relativamente plana . . . 18
2.13. Vista de la mesa de entrada virtual con Kinect Arms . . . 18

3.1. Casos de uso para el control de usuarios . . . 23
3.2. Casos de uso para el procesamiento de dedos y vectores . . . 26
3.3. Casos de uso para la interpretación de señas . . . 28
3.4. APIs del *toolkit* desarrollado . . . 32
3.5. Lista doblemente enlazada con datos de usuarios conectados . . . 33
3.6. Primer nodo en la lista de usuarios conectados . . . 34
3.7. Agregación de un nodo a la lista de usuarios conectados . . . 34
3.8. Asignación de nuevo identificador . . . 35
3.9. Búsqueda exitosa de una dirección IP dentro de la lista de usuarios conectados . . . 36
3.10. Búsqueda fallida de una dirección IP dentro de la lista de usuarios conectados . . . 36

3.11. Búsqueda de un usuario en la lista de usuarios conectados . . . . . . 37
3.12. Eliminación del primer usuario de la lista . . . . . . . . . . . . . . . . 38
3.13. Eliminación del último usuario de la lista . . . . . . . . . . . . . . . . 38
3.14. Eliminación del registro de un usuario que no está al inicio o al final de la lista . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
3.15. Eliminación del registro de un usuario de la lista de usuarios conectados 39
3.16. Contenido de la lista de usuarios conectados . . . . . . . . . . . . . 40
3.17. Obtención de datos en la lista de usuarios conectados . . . . . . . . . 40
3.18. Determinación del número de usuarios conectados en la lista . . . . . 41
3.19. Verificación de la lista de usuarios conectados . . . . . . . . . . . . . 41
3.20. Obtención del número de dedos extendidos de una o ambas manos del usuario . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
3.21. Visualización de las dos manos del usuario con diez dedos extendidos 43
3.22. Colision de un punto $P3$ en un rectángulo formado por $P1$ y $P2$ . . . 44
3.23. Verifica si hubo o no una colisión . . . . . . . . . . . . . . . . . . . . 44
3.24. Dos dedos extendidos del usuario hacia el cuadrante uno . . . . . . . 45
3.25. Obtención del cuadrante hacia donde apunta cada dedo del usuario . 46
3.26. Seña realizada con dos dedos extendidos del usuario . . . . . . . . . . 47
3.27. Almacenamiento en un archivo del código fuente en lenguaje C de una seña realizada por el usuario . . . . . . . . . . . . . . . . . . . . . . . 48
3.28. Validación de una seña realizada por el usuario . . . . . . . . . . . . 49
3.29. Almacenamiento en un archivo de la información de una seña realizada por el usuario . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
3.30. Seña realizada con una mano y los cinco dedos extendidos . . . . . . 51
3.31. Almacenamiento en un archivo del código fuente en lenguaje C de una seña, empleando el cuadrante de cada dedo extendido del usuario . . 52
3.32. Validación de una seña con dirección realizada por el usuario . . . . . 53

5.1. Menú para crear señas para el dispositivo Leap Motion . . . . . . . . 81
5.2. Diagrama de casos de uso de la aplicación Ratón . . . . . . . . . . . . 81
5.3. Seña para emular el clic del ratón . . . . . . . . . . . . . . . . . . . . 82
5.4. Diagrama de casos de uso de la aplicación *Rompecabezas* . . . . . . . 83
5.5. Seña para finalizar la ejecución de la aplicación Rompecabezas . . . . 83
5.6. El clásico juego *PingPong* . . . . . . . . . . . . . . . . . . . . . . . . 88
5.7. Caso de uso de la aplicación *PingPong* . . . . . . . . . . . . . . . . . 88
5.8. Seña para realizar un disparo . . . . . . . . . . . . . . . . . . . . . . 90
5.9. Pantalla de juego . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 90
5.10. Diagrama de casos de uso de la aplicación *Zombis invasores* . . . . . 91
5.11. Puntaje ajustado de cada subescala para la realización de una seña con una mano del usuario . . . . . . . . . . . . . . . . . . . . . . . . . . 96
5.12. Puntaje ajustado de cada subescala para la realización de una seña con ambas manos del usuario . . . . . . . . . . . . . . . . . . . . . . . . . 98
5.13. Puntaje ajustado de cada subescala para la interacción con las aplicaciones y el dispositivo Leap Motion . . . . . . . . . . . . . . . . . . . 100

# Índice de cuadros


2.1. Clases disponibles en la API de Leap Motion . . . . . . . . . . . . . . . 10
2.2. Tabla comparativa de los trabajos relacionados . . . . . . . . . . . . . 20

3.1. Caso de uso para agregar un usuario a la lista de usuarios conectados 24
3.2. Caso de uso para consultar un identificador asociado a un usuario conectado . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
3.3. Caso de uso para eliminar un usuario de la lista de usuarios conectados 25
3.4. Caso de uso para obtener los datos de cada usuario registrado en la lista de usuarios conectados . . . . . . . . . . . . . . . . . . . . . . . 25
3.5. Caso de uso para contar los usuarios conectados . . . . . . . . . . . 25
3.6. Caso de uso para verificar si la lista de usuarios conectados está vacía 26
3.7. Caso de uso para calcular una colisión entre un punto y un rectángulo 27
3.8. Caso de uso para contar los dedos de una o ambas manos de un usuario 27
3.9. Caso de uso para calcular hacia qué cuadrante apunta cada dedo del usuario . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
3.10. Caso de uso para crear código en lenguaje C de una seña sin dirección 29
3.11. Caso de uso para guardar en un documento de texto la información de una seña realizada por el usuario . . . . . . . . . . . . . . . . . . . . 30
3.12. Caso de uso para crear código en lenguaje C de una seña con dirección 30

5.1. Caso de uso para mover el ratón utilizando una mano cualquiera del usuario . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82
5.2. Caso de uso para enviar el evento de clic al sistema operativo utilizando una seña en particular . . . . . . . . . . . . . . . . . . . . . . . . 82
5.3. Caso de uso para comenzar el juego de *Rompecabezas* . . . . . . . . . 84
5.5. Caso de uso para mover una imagen del *Rompecabezas* . . . . . . . . 84
5.4. Caso de uso para salir del juego utilizando una seña . . . . . . . . . . 84
5.6. Caso de uso para mover las raquetas del juego utilizando Leap Motion 89
5.7. Caso de uso para comenzar el juego de *PingPong* . . . . . . . . . . . 89
5.8. Caso de uso para salir del juego utilizando una seña . . . . . . . . . . 89
5.9. Caso de uso para mover las naves del juego utilizando una mano especifica de cada usuario . . . . . . . . . . . . . . . . . . . . . . . . . 91
5.10. Caso de uso para disparar láser utilizando una seña . . . . . . . . . . 91
5.11. Caso de uso para comenzar el juego de *Zombis invasores* . . . . . . . 92

5.12. Caso de uso para salir del juego utilizando una seña . . . . . . . . . . . 92
5.13. Características de aplicaciones creadas . . . . . . . . . . . . . . . . . . . 93
5.14. Resultados del cuestionario de carga de trabajo para la realización de una seña con una mano del usuario . . . . . . . . . . . . . . . . . . . . . 95
5.15. Resultados del cuestionario de carga de trabajo para la realización de una seña con ambas manos del usuario . . . . . . . . . . . . . . . . . . . 97
5.16. Resultados del cuestionario de carga de trabajo experimentada al interactuar con las aplicaciones . . . . . . . . . . . . . . . . . . . . . . . . 99

# List of Algorithms

1. Asignar identificador a un usuario nuevo en la lista de usuarios conectados 58
2. Buscar el identificador asociado a una dirección IP . . . . . . . . . . . 59
3. Borrar a un usuario de la lista de usuarios conectados) . . . . . . . . 60
4. Desplegar la información de la lista de usuarios conectados. . . . . . . 61
5. Conocer el número de usuarios conectados. . . . . . . . . . . . . . . . 61
6. Verificar si lista de usuarios conectados se encuentra vacía. . . . . . . 62
7. Suma de dedos extendidos de una o ambas manos del usuario. . . . . 63
8. Detectar la colisión de un punto dentro de un rectángulo . . . . . . . . 63
9. Calcular el cuadrante al que apunta cada uno de los dedos del usuario 65
10. Almacenar el código en C de una seña realizada por el usuario . . . . 67
11. Validación de una seña realizada con una mano específica del usuario 68
12. Validación de una seña realizada con ambas manos del usuario . . . . 70
13. Almacenar la información de una seña realizada por el usuario . . . . 71
14. Almacenar el código en C de una seña con dirección realizada por el usuario . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
15. Validación de una seña con dirección realizada con una mano específica del usuario . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
16. Validación de la una seña con dirección realizada con ambas manos del usuario . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
17. Bucle de ejecución del cliente . . . . . . . . . . . . . . . . . . . . . . 86
18. Bucle de ejecución del servidor . . . . . . . . . . . . . . . . . . . . . 87

# Capítulo 1

# Introducción

## 1.1. Contexto de la investigación

Actualmente no existen trabajos que permitan a los dispositivos Leap Motion una interacción adecuada en un espacio de trabajo multi-usuario, aun así se han realizado proyectos que están dirigidos al manejo de objetos 3D, en los cuales se presentan espacios multi-usuario donde el usuario puede interaccionar con el sistema y los objetos a través de diferentes Leap Motion conectados cada uno a una máquina distinta.

Ellis et al. sugieren que el incremento de las nuevas tecnologías desarrolladas refleja un cambio en el énfasis de emplear sistemas computacionales para facilitar la interacción humana y la resolución de problemas [1]. Al mismo tiempo, la necesidad de comunicarnos en el mundo digital crece pues estas tecnologías permiten gráficos más reales, necesarios para el desenvolvimiento de aplicaciones demandantes, e.g., videojuegos. Es por ello que el desarrollo de dispositivos de entrada gestual, e.g., Leap Motion, está en incremento, tratando de satisfacer las necesidades del consumidor, como facilidad y comodidad en el control de dichas aplicaciones.

### 1.1.1. Trabajo Colaborativo Asistido por Computadora

El TCAC [1] es una de las áreas más sobresalientes de este trabajo, cuya función consiste en permitir que un grupo de personas trabaje y produzca una tarea en común de manera coordinada, teniendo como resultado un incremento en las oportunidades de trabajo en grupo, e.g., datos compartidos [2].

Es debido al incremento del éxito que brinda el trabajo en grupo que en la actualidad, muchas organizaciones optan por trabajar conjuntamente, ya sea entre la misma organización o en conjunto con otras en diferentes partes del mundo, sin importar la distancia. Debido a esa necesidad de trabajar en grupo, los desarrolladores de sistemas

[1] Computer Supported Cooperative Work (CSCW) por sus siglas en inglés

colaborativos han puesto un énfasis en la construcción de aplicaciones que soporten colaboradores físicamente distantes.

### 1.1.2. Interacción Humano Máquina

La disciplina IHM [2] se puede abordar como un tema compuesto de varias disciplinas científicas y culturales tales como Psicología, Sociología, Filosofía, Lingüística y Ciencias de la computación, pues reúne conocimientos de estas ciencias para complementarse. Está área de investigación surgió a principios de los años 80, inicialmente como una especialidad de las Ciencias de la Computación [3].

La IHM, según Truong et al., ha jugado un papel muy importante en el desarrollo computacional de la sociedad, pues las investigaciones, en esta disciplina, tienen como objetivo mejorar el confort, la ergonomía y la portabilidad, así como ahorrar tiempo a los usuarios al momento de interaccionar con los dispositivos y/o aplicaciones que permiten realizar las actividades de los usuarios [4].

Según D. Norman y S. Draper, la capacidad de los usuarios para controlar y evaluar el estado de un sistema mejora la calidad de sus actividades [5].

Ambas áreas pueden coexistir y expandir los resultados que se presenten a lo largo de esta investigación, pues TCAC necesita de IHM para permitir que múltiples usuarios desarrollen las utilidades necesarias para interaccionar con las computadoras y/o dispositivos al tiempo que trabajan en un ambiente colaborativo.

Por lo tanto, el contexto de investigación de este trabajo está basado en la clasificación de la ACM de 1998 y se encuentra representado en la Figura 1.1.

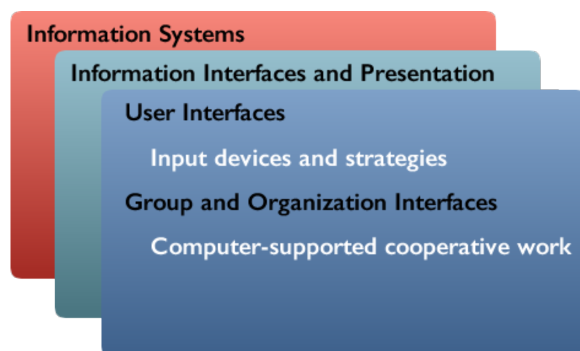


Figura 1.1: Contexto específico de investigación

[2]Human Computer Interaction (HCI) por sus siglas en inglés

## 1.2. Justificación

Como se ha mencionado anteriormente, existe un incremento de nuevas tecnologías en el campo de la visualización digital y ésto ha originado la creación de sistemas que soporten la comunicación y el control de un espacio de trabajo idóneo para explotar al máximo las capacidades y características de dichas tecnologías. Un espacio de trabajo multi-usuario ofrece la oportunidad de incrementar la experiencia de los usuarios, ya sea al realizar tareas en común o diferentes, incluso estando físicamente distantes.

Actualmente, existen trabajos de aplicaciones colaborativas que permiten a los usuarios interaccionar con los objetos de la aplicación y diversos Leap Motion conectados cada uno a una máquina diferente. Es por ello que resulta interesante el hecho de explotar las capacidades de este Leap Motion en entornos multi-usuario, pues los desarrolladores no han logrado tener éxito al realizar una interacción adecuada entre estas aplicaciones y los dispositivos, debido a que no cuentan con un soporte de interacción.

Aunque se han realizado aplicaciones que funcionan en entornos multi-usuario y realizan interacciones entre los dispositivos Leap Motion y los objetos creados por la misma aplicación, no han tenido como propósito principal crear un soporte de interacción basado en Leap Motion de entrada gestual. Es debido a los problemas de interacción en entornos multi-usuario que se necesita un soporte de interacción para Leap Motion con la finalidad de que las aplicaciones, que trabajan de manera colaborativa, puedan interaccionar de manera correcta.

Un ejemplo de un soporte de interacción sería un *toolkit* que permita a los programadores interaccionar sus aplicaciones con los dispositivos Leap Motion, mediante mecánicas previamente estipuladas, e.g., métodos o funciones, que proporcionen control sobre la comunicación entre la aplicación y los dispositivos conectados.

## 1.3. Antecedentes y motivación

El estudio de la precisión del dispositivo Leap Motion, realizada por Weichert et al., trajo como resultado la especificación de pruebas objetivas para sensores 3D y la definición de criterios de calidad para este tipo de sensores [6]. Este estudio permite utilizar Leap Motion de manera más precisa y comprender porqué ha sido utilizado en lugar de otros dispositivos en investigaciones que emplean dispositivos de entrada gestual. Sin embargo desde el lanzamiento de Leap Motion, las investigaciones relacionadas con Leap Motion se han limitado a la experimentación y desarrollo de aplicaciones que permiten a un solo usuario interaccionar con un solo Leap Motion, dejando a un lado el estudio de la interacción de varios dispositivos en un entorno multi-usuario. Es por ello que la principal motivación de este trabajo de tesis es facilitar dicha interacción entre los usuarios y la aplicación colaborativa mediante un conjunto de funciones que encapsulen las instrucciones necesarias para dicho

propósito. Otra de las motivaciones es que Leap Motion es compacto y de bajo costo. Aunque existen dispositivos como las cámaras de profundidad, e.g. Kinect, éstas no hacen seguimiento de partes específicas del cuerpo, pues calculan las posiciones de los elementos rastreados mediante la distancia que existe entre éstos y la cámara. Esta limitación hace que Leap Motion sea uno de los principales dispositivos a utilizar cuando se requiere el seguimiento de nuestras manos.

## 1.4. Planteamiento del problema

Como se ha mencionado, Leap Motion ha sido utilizado, en la mayoría de las investigaciones, por un solo usuario, el cual puede interaccionar con un solo Leap Motion, lo cual se debe a que su diseño está pensado para trabajar con una sola persona y en una computadora. Sin embargo, los esfuerzos de investigación no se han limitado a esto, pues existen trabajos que han empleado varios dispositivos Leap Motion con el propósito de permitir la interacción de diferentes usuarios con una aplicación. Aunque no se ha documentado la manera en que se realizó la interacción, se sabe que el problema que se presentó fue la colisión de datos de entrada de Leap Motion, ocasionando que el medio digital se congelara.

Este trabajo de maestría abordará la investigación de cómo permitir la interacción entre diferentes dispositivos Leap Motion para crear funciones que faciliten al programador la creación de aplicaciones que necesiten dos o más dispositivos Leap Motion.

### 1.4.1. Propuesta de solución

Una solución propuesta para este documento de tesis es el desarrollo de un *toolkit*, el cual debe ofrecer un conjunto de funciones que permitan ahorrar tiempo y optimizar el trabajo de construir un soporte multi-usuario que requiera el uso de diversos dispositivos Leap Motion. Además de brindarle ayuda al programador con mecánicas de control que permitan orquestar una comunicación correcta entre los usuarios de la aplicación.

## 1.5. Hipótesis

Es posible obtener métodos de solución a problemas recurrentes al momento de compartir el espacio de trabajo entre una aplicación colaborativa y los dispositivos Leap Motion. Y construir una solución general, e.g., *toolkit*, que permita a los desarrolladores realizar una buena interacción entre sus aplicaciones, los dispositivos y los

usuarios.

## 1.6. Objetivos del proyecto

### General

Realizar un conjunto de herramientas para desarrolladores de aplicaciones multiusuario que facilitará la implementación de soportes de interacción humano-computadora mediante el uso de varios dispositivos Leap Motion.

### Particulares

- Comprender el funcionamiento de Leap Motion a profundidad para determinar las funciones que se deben ofrecer en el *toolkit*.
- Poner a prueba la interacción de los dispositivos Leap Motion mediante una aplicación multi-usuario con la finalidad de comprobar la funcionalidad de el *toolkit*.

## 1.7. Organización de la tesis

La presente tesis está estructurada en seis capítulos, como se puede apreciar en la Figura 1.2.

El Capítulo 2 (marco teórico), comprende la explicación detallada de Leap Motion, muchas de sus características estructurales como de programación, así como el estado del arte de Leap Motion. Los Capítulos 3 y 4 son las contribuciones de la tesis y explican el desarrollo del trabajo práctico de la tesis, así como la implementación del *toolkit* en las diferentes aplicaciones programadas. El Capítulo 5 contiene las aplicaciones realizadas para comprobar la efectividad de dicho *toolkit*. Finalmente, el Capítulo 6 está compuesto de un análisis del trabajo práctico, en el cual se exponen algunas características del *toolkit* y de los trabajos a futuro.

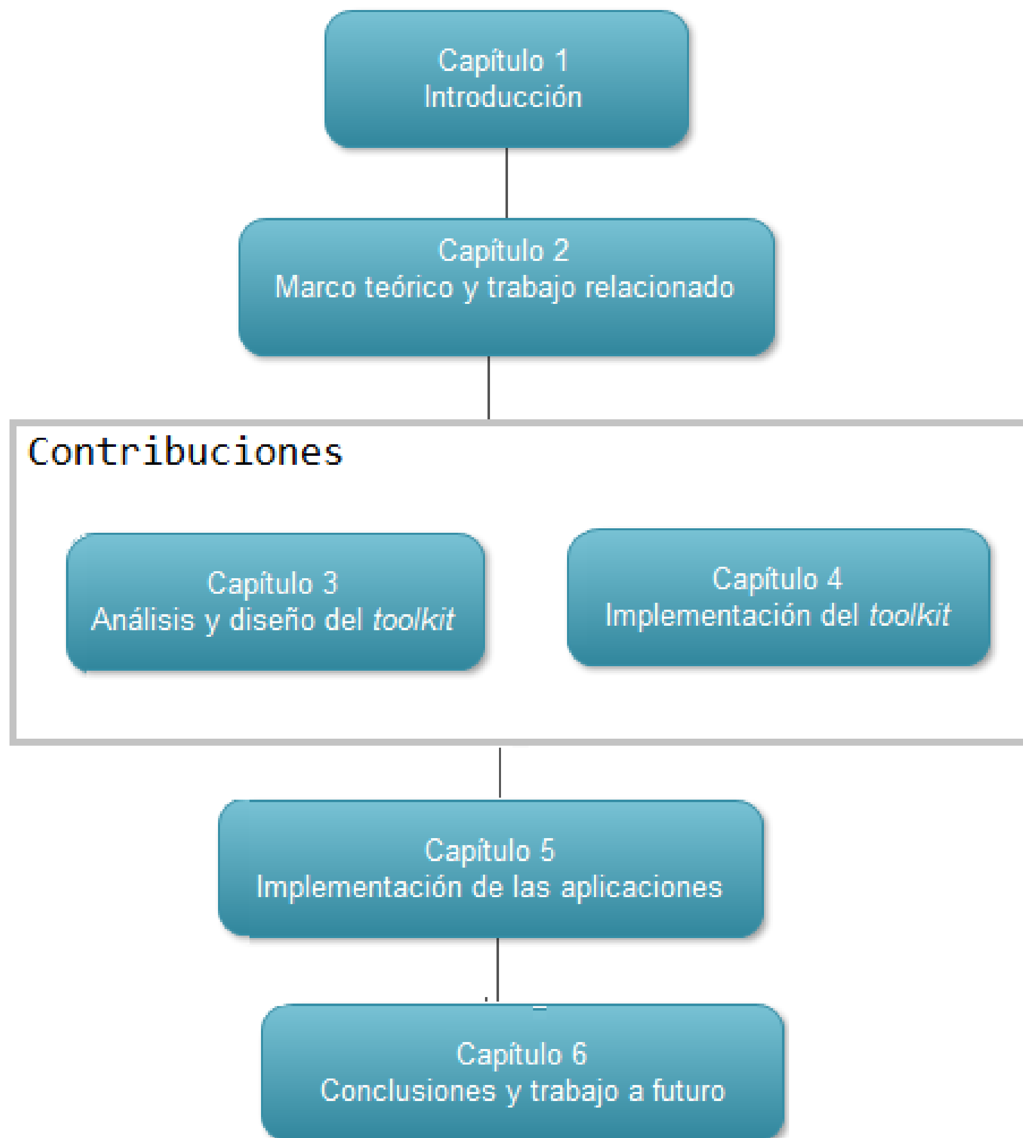


Figura 1.2: Organización de la tesis

# Capítulo 2

# Marco teórico y trabajo relacionado

En la Sección 2.1 se explican algunas característica del dispositivo Leap Motion así como las especificaciones de su API y algunas funciones representativas. En la Sección 2.2 se realiza un análisis de los diferentes trabajos relacionados con el tema de tesis y, finalmente, en la Sección 2.3 se hace un análisis comparativo de dichos trabajos.

## 2.1. Leap Motion

Leap Motion es un dispositivo de entrada, el cual se conecta a una computadora mediante un puerto USB. Fue elegido como el dispositivo base de esta tesis porque: 1) es un dispositivo nuevo capaz de detectar el movimiento de las manos del usuario, 2) es económico y muy potente pese a su precio, 3) la API de Leap Motion permite su utilización en diferentes lenguajes de programación y 4) a diferencia de las cámaras de profundidad, Leap Motion permite rastrear y capturar información de las manos del usuario y de algunas herramientas, e.g., lápices y bolígrafos, de manera más precisa.

En la Sección 2.1.1 se presentan las características esenciales de Leap Motion. Posteriormente, en la Sección 2.1.2, se exponen las funciones requeridas para la creación de nuestro *toolkit*, así como las clases más representativas de la API de Leap Motion.

### 2.1.1. Características

Leap Motion está compuesto por dos cámaras infrarrojas y tres *leds* [6] encargados de reconocer y rastrear las manos y los dedos del usuario, así como algunos herramientas, en un espacio cartesiano. Los sensores de Leap Motion están acomodados sobre el eje X y están dirigidos hacia el eje Y para realizar mediciones, como se observa en la Figura 2.1.

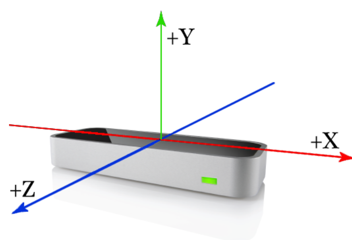


Figura 2.1: Mapa cartesiano de Leap Motion

El rango de posicionamiento de las manos y/o herramientas para la interacción con Leap Motion es de 7 a 25 centímetros y puede ser modificado mediante software. La información capturada dentro de dicho rango se guarda en marcos ($frames$), a una tasa promedio de 200 marcos por segundo.

Cada marco contiene información de la escena capturada, como la distancia entre el objeto identificado y el dispositivo, así como las coordenadas del objeto en el mapa cartesiano. Los marcos están identificados con un número ($ID$), el cual en ocasiones puede no ser consecutivo debido a la pérdida de algunas capturas de marcos. Esta información permite a los desarrolladores de aplicaciones conocer los datos de los objetos rastreados.

En la Figura 2.2 se muestra un ejemplo de la información capturada. Se puede observar que en la primera línea hay un identificador de marco, posteriormente se encuentra la distancia en milímetros entre las manos y el dispositivo, así como la dirección a la que apunta la mano del usuario ($Vector < x, y, z >$).

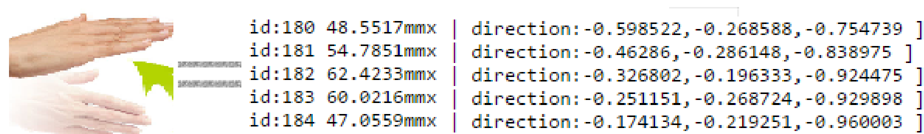


Figura 2.2: Información de un marco obtenido de la interacción con Leap Motion

Leap Motion también cuenta con un software que sirve de controlador, el cual permite modificar el rango de captura de las herramientas o mejorar el rastreo en condiciones de iluminación intensa, entre otras muchas opciones.

También cuenta con un visualizador del espacio de trabajo, el cual se aprecia en la Figura 2.3. Su función es desplegar lo que Leap Motion está capturando, pues dibuja las manos del usuario y/o herramientas de manera dinámica. Tanto el visualizador como muchas otras aplicaciones para mejorar la captura pueden ser obtenidas en la tienda de Leap Motion y tienen la funcionalidad de enriquecer la experiencia de interacción del usuario con el dispositivo.

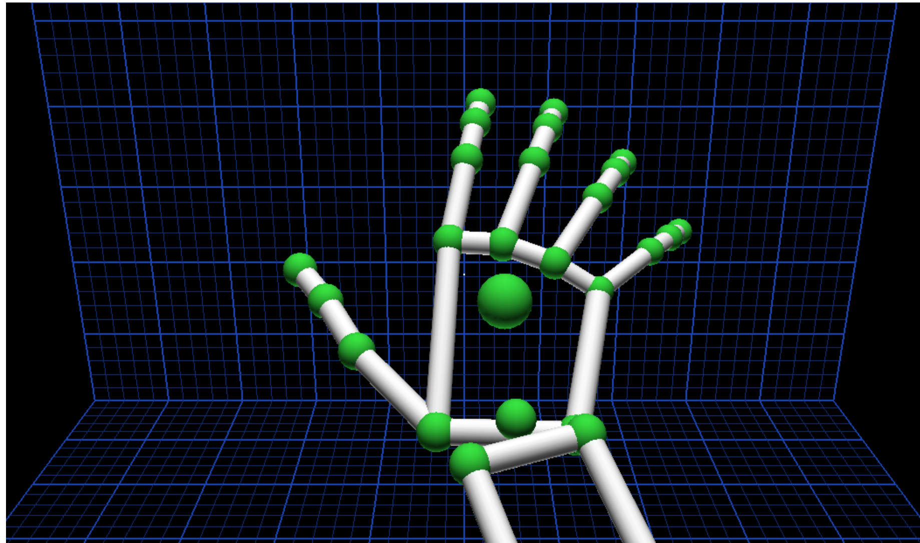

Figura 2.3: Visualizador de Leap Motion

### 2.1.2. API y funciones requeridas

La API de Leap Motion es multiplataforma, pues el *kit* de desarrollador se puede instalar en Linux, Windows y Mac y está desarrollada en varios lenguajes de programación como son Javascript, C#, C++, Java, Python y Objetive-C. Muchos de los módulos de Leap Motion han sido integrados en motores de videojuegos, siendo los más usados Unity[1] y Unreal Engine[2].

La API de Leap Motion contiene 30 clases, las cuales se listan en el Cuadro 2.1 y están diseñadas para trabajar con los datos recopilados por Leap Motion. Las clases más representativas siguen un orden jerárquico, el cual puede observarse en la Figura 2.4.

La clase principal, llamada `Frame`, está encargada de crear marcos (explicados en la Sección 2.1.1) y define un conjunto de métodos que proveen acceso a la información de cada marco[3]. Un objeto *frame* contiene información de otros objetos, como características y atributos, que pueden ser leídos y modificados por la clase `Controller` y `Listener`.

Cada objeto *frame* está compuesto por un conjunto de objetos que se describen a continuación:

[1]https://unity3d.com/es
[2]https://www.unrealengine.com/what-is-unreal-engine-4
[3]https://developer.leapmotion.com/documentation/cpp/devguide/Leap_Frames.html

| | | |
|---|---|---|
| `Arm` | `Gesture` | `Mask` |
| `Bone` | `GestureList` | `MaskList` |
| `CircleGesture` | `Hand` | `Pointable` |
| `Config` | `HandList` | `PointableList` |
| `Controller` | `Image` | `ScreenTapGesture` |
| `Device` | `ImageList` | `SwipeGesture` |
| `DeviceList` | `InteractionBox` | `Tool` |
| `Finger` | `KeyTapGesture` | `ToolList` |
| `FingerList` | `Listener` | `TrackedQuad` |
| `Frame` | `Matrix` | `Vector` |

Cuadro 2.1: Clases disponibles en la API de Leap Motion

- *hand* (`Hand`): contiene la posición y orientación de una mano del usuario, así como el movimiento, la velocidad y una lista de los dedos correspondientes a dicha mano.
- *arm* (`Arm`): incluye la posición, orientación y dirección del brazo del usuario.
- *pointable* (`Pointable`): comprende las características básicas de dedos y herramientas del usuario. Esta información sirve para que una clase identifique si un objeto es un dedo o una herramienta, dependiendo de sus características.
- *finger* (`Finger`): es una extensión del objeto *pointable* y engloba la información necesaria para determinar si el objeto capturado es un dedo.
- *tool* (`Tool`): es una extensión del objeto *pointable* y comprende la información requerida para determinar si el objeto capturado es una herramienta.
- *bone* (`Bone`): describe la posición y orientación de cada uno de los huesos asociados a los dedos y manos del usuario (ver Figura 2.5).
- *gesture* (`Gesture`): brinda la información necesaria para determinar si un conjunto de marcos contienen un gesto; la información puede ser la duración del gesto, así como el conjunto de manos, dedos y/o herramientas del usuario.
- *image* (`Image`): provee los datos recién capturados por los sensores de Leap Motion, además de una tabla para calibrar sus cámaras.

Algunos objetos como *hand*, *pointable*, *gesture*, *tool* y *finger* se pueden agrupar en listas, las cuales pueden ser modificadas mediante métodos definidos en diferentes clases como `HandList, PointableList y GestureList`.

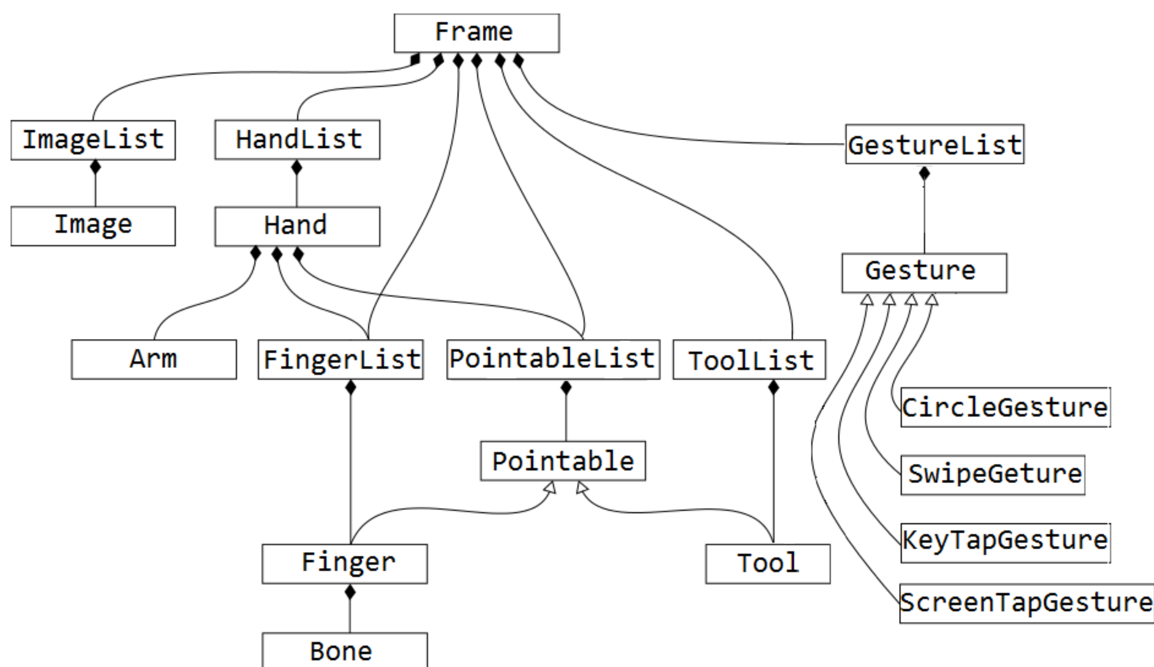


Figura 2.4: Jerarquía de las clases de la API

Los objetos anteriormente explicados son creados y modificados por sus respectivas clases. A continuación se explica el uso de cada clase:

- `Arm`: contiene métodos que permiten trabajar con los datos relacionados con el antebrazo.
- `Bone`: brinda métodos que modifican la información sobre los huesos de los dedos del usuario.
- `Controller`: crea una instancia para obtener la información de cada marco.
- `Device`: es la representación de Leap Motion conectado físicamente a una computadora y contiene información específica, como la posición y orientación del dispositivo relativas al usuario.
- `DeviceList`: proporciona métodos que permiten listar objetos de la clase `Device`.
- `Finger`: incluye métodos que ayudan a obtener la información de cada uno de los dedos de las manos del usuario.
- `FingerList`: brinda métodos que permiten listar objetos de la clase `Finger`.
- `Frame`: contiene métodos que representan un conjunto de manos, dedos y/o herramientas del usuario que Leap Motion detecta (ver Sección 2.1.2).

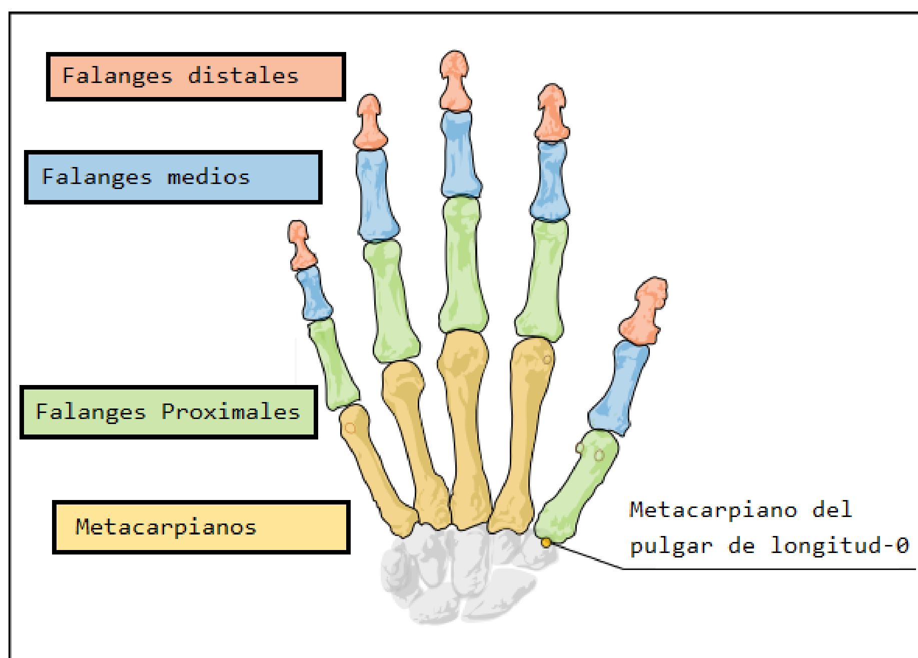


Figura 2.5: Huesos que componen los dedos de una mano

- `Gesture`: representa un conjunto de movimientos de brazos, manos y/o dedos del usuario y contiene métodos que permiten establecer el tiempo que dura un gesto, cuántas manos se usan, entre otras características.

- `GestureList`: comprende un conjunto de métodos que permiten listar los cuatro objetos de la clase `Gesture`, los cuales pueden verse en la Figura 2.6.

- `Hand`: los métodos de esta clase permiten obtener información física de las manos del usuario, como la posición de la palma, la velocidad de movimiento de la mano, la dirección de los dedos y la lista de los dedos de la mano seleccionada.

- `HandList`: brinda métodos para listar objetos de la clase `Hand`.

- `Image` : permite modificar cada una de las imágenes capturadas por las cámaras de Leap Motion.

- `ImageList`: permite listar objetos de la clase `Image`.

- `Listener`: contiene métodos que pueden ser usados para responder a eventos realizados por la clase `Controller`.

- `Pointable`: brinda métodos que permiten obtener y modificar la información física de los dedos y herramientas del usuario. Los objetos de esta clase contienen las características básicas de dedos y herramientas del usuario.

- `PointableList`: posibilita la creación de listas de objetos de la clase `Pointable`.

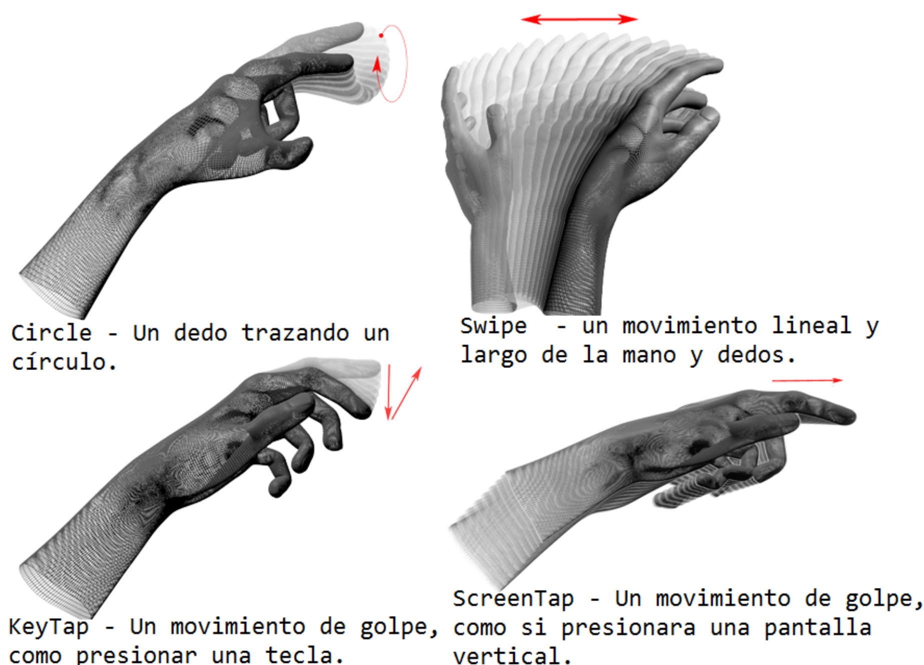


Figura 2.6: Conjunto de gestos reconocidos por la API de Leap Motion

- `Vector`: comprende un conjunto de métodos que permiten conocer y modificar la información sobre cada uno de los ejes del mapa cartesiano, como ángulos, coordenadas y puntos.

## 2.2. Trabajo relacionado

A continuación, se describen algunas de las implementaciones relacionadas con el cometido de esta tesis. En la Sección 2.2.1 se describen trabajos relacionados con el uso del dispositivo Leap Motion. En la Sección 2.2.2 se explican algunos estudios relevantes que utilizan dispositivos para rastrear partes del cuerpo. Estos trabajos fueron realizados en el área de Trabajo Colaborativo Asistido por Computadora, la cual constituye una de las áreas principales del contexto de investigación de esta tesis.

### 2.2.1. Uso de Leap Motion

El trabajo de Weichert et al. [6] realiza una investigación extensa y profunda de las funciones y alcances de los componentes de Leap Motion, como la especificación y precisión de sus sensores, además de diversos criterios de calidad para sensores 3D parecidos a los de Leap Motion. Este trabajo es indispensable para conocer a fondo su funcionamiento. En la Figura 2.7, se presentan dos vistas del dispositivo Leap Motion, donde se aprecian las dimensiones físicas y la ubicación de los sensores de dicho dispositivo.

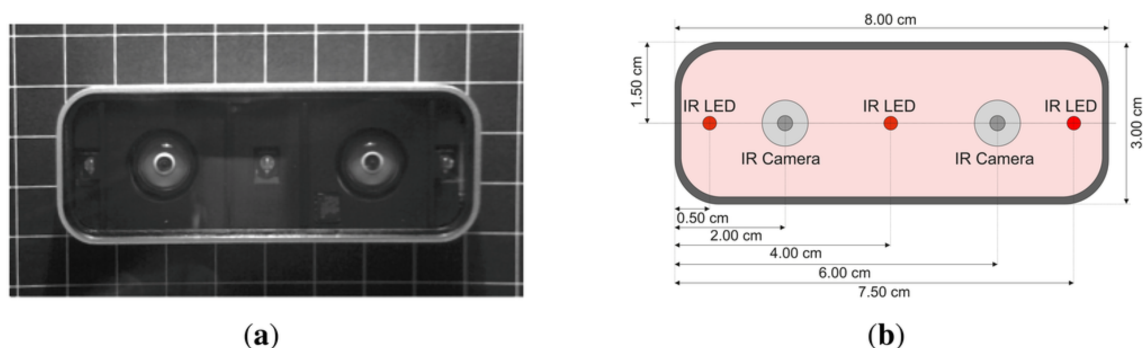


Figura 2.7: Visualización del dispositivo Leap Motion (a) imagen infrarroja y (b) vista esquemática

## Leap Trainer

O'Leary diseñó e implementó un sistema de gestos llamado Leap Trainer [4], el cual fue codificado en lenguaje JavaScript. Este sistema es capaz de interpretar y comparar gestos predefinidos con gestos realizados por el usuario, así como generar código en el mismo lenguaje a partir de la interpretación de éstos.

Leap Trainer ofrece al usuario una interfaz amigable, pues es simple en cuanto a funciones y aspecto, como se muestra en la Figura 2.8. Este sistema funciona con Leap Motion, el cual captura la información de los movimientos de las manos. Leap Trainer devuelve la información recopilada en código JavaScript, el cual puede ser añadido a cualquier documento HTML. Además, Leap Tainer es capaz de reconocer gestos de múltiples trazos (como realizar una Z), aunque no ha tenido un uso concreto hasta ahora.

Figura 2.8: Interfaz de usuario de Leap Trainer

[4] https://github.com/roboleary/LeapTrainer.js?utm_source=javascriptweekly&utm_medium=email#importing-and-exporting-from-leaptrainer

### 2.2.2. Soporte de trabajo colaborativo

A continuación, se analizan trabajos relacionados en los cuales se utilizan dispositivos que permiten rastrear ciertas partes del cuerpo, e.g., brazos, manos y dedos de los usuarios. En dichos trabajos, se pusieron en evidencia algunos problemas importantes en la adopción de aplicaciones colaborativas.

#### SprBlender

El soporte de interacción de múltiples usuarios, usando cada uno un Leap Motion, que se pretende realizar en este trabajo de investigación, es similar al que se realizó en el trabajo presentado por Mitake et al. [12] donde se muestra un nuevo entorno, llamado SprBlender, para crear personajes en 3D, como se aprecia en la Figura 2.9.

SprBlender permite, a los usuarios de esta aplicación, crear y manipular personajes en 3D mediante motores de física. Además, este entorno permite la interacción de múltiples usuarios con los personajes, mediante la comunicación de múltiples dispositivos Leap Motion conectados en diferentes computadoras.

Para la demostración del funcionamiento de la aplicación, se conectaron diversos dispositivos Leap Motion a una computadora cada uno. Sin embargo, cuando los usuarios interaccionaron con la aplicación mediante los dispositivos, ocurrieron colisiones en el espacio de trabajo virtual y físico (i.e. más de un usuario intentaba tomar el mismo objeto y en algunas ocasiones, un solo dispositivo captaba las manos de dos o más usuarios, debido al poco espacio existente entre un dispositivo y otro).

Figura 2.9: Ejemplo de control de la aplicación SprBlender

#### 3D Helping Hands

Entrando más a fondo en la investigación de trabajos realizados con dispositivos de entrada de gestos, tenemos las cámaras de profundidad, donde encontramos el

trabajo desarrollado por Tecchia et al. [13]. Este trabajo se centra en la categoría específica de los sistemas de colaboración a distancia (ver Figura 2.10), donde gestos con las manos son realizados por una persona, la cual tiene como objetivo ayudar a un trabajador físicamente distante a realizar tareas manuales, en las que se necesita manipular máquinas o herramientas.

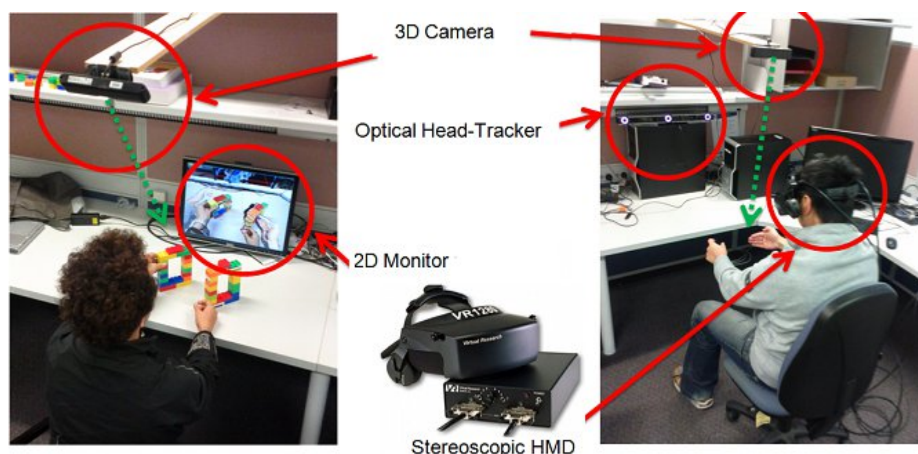


Figura 2.10: Funcionamiento de 3D Helping Hands en un ambiente colaborativo con usuarios físicamente distantes

La mayoría de los sistemas que permiten la colaboración y manipulación de maquinaria a distancia, utilizan tecnologías que permiten manipular maquinaria con movimientos en dos dimensiones (i.e. permiten movimientos horizontales y verticales) lo cual es un problema si los usuarios de dichos sistemas necesitan manipular objetos que se encuentren detrás de otro (i.e. movimientos con profundidad). 3D Helping Hands utiliza cámaras de profundidad para capturar la escena de trabajo, solventando el problema de la profundidad en el espacio de trabajo.

## Proximity Toolkit

El problema de distribuir la información entre los diferentes miembros de un grupo de trabajo puede ser complicado, pues no solo es el medio dónde se decide compartir la información, sino también los dispositivos y las aplicaciones que se utilicen para coordinar la manera en que se compartirá.

Proximity Toolkit es sin duda el trabajo con mayor relación al objetivo general de esta investigación, pero utiliza cámaras de profundidad, en este caso Kinect. Este *toolkit* creado por Marquardt et al. [14] simplifica la exploración de técnicas de interacción mediante el uso de información proxémica entre personas, dispositivos portátiles, grandes superficies interactivas y objetos no digitales.

Aunque las relaciones proxémicas son comprendidas de manera innata entre los seres

humanos, muchos de los sistemas de cómputo ubicuo no cuentan con esta característica.

Proximity Toolkit es la solución que permite a los desarrolladores concentrarse en la construcción de sus sistemas, sin tener que lidiar con la configuración y programación de bajo nivel que involucra el uso de sensores que permiten extraer la información del entorno. Además de brindar un *toolkit* para extraer la información del entorno, Marquardt et al. ofrecen un sistema cliente/servidor que permite a los desarrolladores accesar a información proxémica capturada mediante herramientas visuales que ellos mismos proporcionan. En la Figura 2.11 de muestra un ejemplo del despliegue de información proxémica.

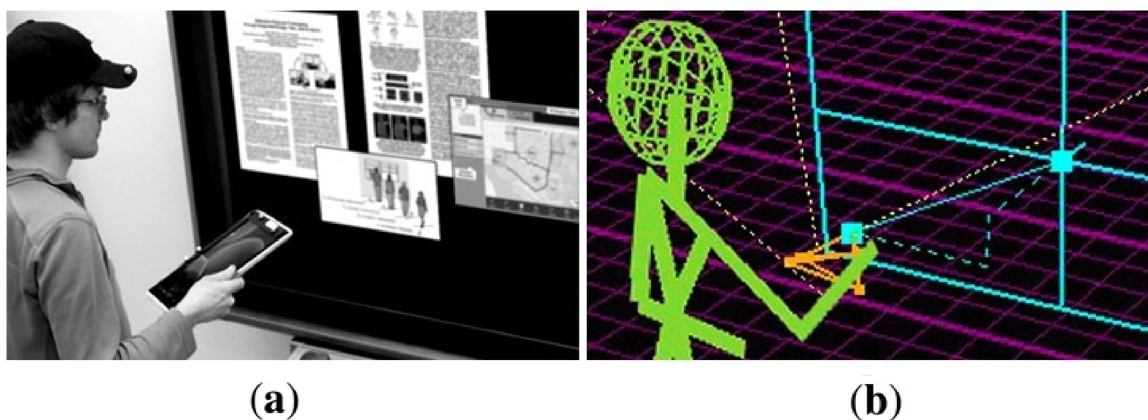


Figura 2.11: (a) Visualización del usuario interactuando con el sistema y (b) vista de la transformación digital del espacio de trabajo mediante el uso de información proxémica del usuario

## Virtual KeyBoard

El sistema Virtual KeyBoard, realizado por Truong et al. [4], muestra teclados inteligentes con la función de predicción de palabras, al momento de que el usuario se encuentra escribiendo. Además, Virtual Keyboard es capaz de almacenar las palabras más usadas por el usuario para posteriormente utilizarlas en la corrección de errores.

Virtual Keyboard puede proyectar varios teclados simultáneamente sobre cualquier superficie relativamente plana. Este sistema utiliza un proyector para desplegar la imagen de los teclados y Kinect para capturar las palabras introducidas en cada uno de ellos. En la Figura 2.12, se muestra la proyección de dos teclados virtuales.

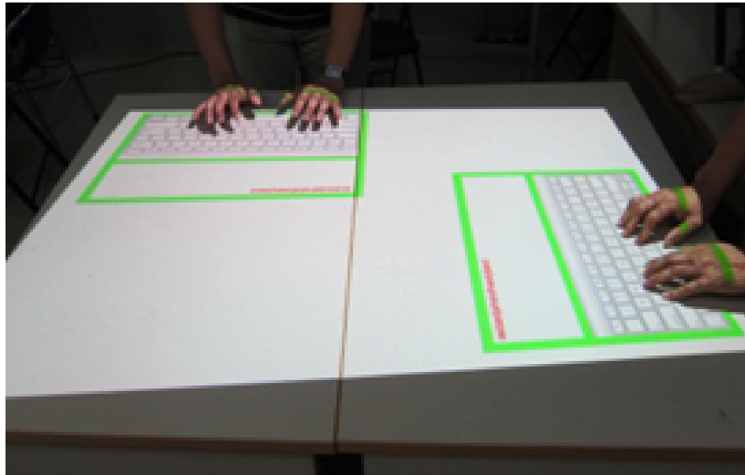

Figura 2.12: Proyección de los teclados virtuales sobre una superficie relativamente plana

## Kinect Arms

Genest et al. [16] realizaron un *toolkit*, llamado Kinect Arms, que permite la captura y visualización de los brazos de los usuarios sobre una superficie. Este sistema captura los brazos mediante Kinect, además de que ofrece una diversidad de funciones para controlar el despliegue en una mesa de entrada virtual colaborativa.

Kinect Arms resuelve el problema del uso de gestos para controlar el espacio de trabajo, utilizando técnicas como la visualización de los brazos sobre el fondo de la aplicación, la diferenciación de usuarios en un entorno multi-usuario y la selección de objetos dentro de la aplicación. En la Figura 2.13 se muestra la vista de la mesa de entrada virtual utilizando Kinect Arms

Figura 2.13: Vista de la mesa de entrada virtual con Kinect Arms

## 2.3. Análisis comparativo

Como se ha observado en los diferentes trabajos antes mencionados, el uso del dispositivo Leap Motion se ha limitado a la interacción de un solo dispositivo con un usuario. Aunque se ha logrado crear una aplicación que permite la manipulación del espacio virtual mediante la interacción de múltiples dispositivos conectados a una computadora cada uno, no existe la documentación donde se explique la manera en que se logró dicha interacción.

Las demás investigaciones de relevancia para este trabajo se han enfocado en cámaras de profundidad. Dichas investigaciones han logrado implementaciones capaces de hacer funcionar diversos dispositivos para dar soporte al trabajo colaborativo, sea a distancia o presencial.

Las características a analizar de los trabajos relacionados son el contexto de investigación, la capacidad del sistema para facilitar código al programador para que este pueda extender las funcionalidades de sus aplicaciones (e.g. señas, gestos y módulos creados por el sistema), el lenguaje de programación utilizado para crear el sistema y el dispositivo que se empleó para facilitar la interacción de los usuarios. En el Cuadro 2.2 se exponen estas características.

El contexto de investigación (ver segunda columna del Cuadro 2.2) de los sistemas descritos en la Sección 2.2, es una característica importante debido a que la mayoría de los trabajos relacionados se inscriben en el mismo contexto de investigación de este trabajo de tesis. Así, SprBlender, 3D Helping Hands, Proximity Toolkit, Virtual Keyboard y Kinect Arms son investigaciones de HCI[5] y CSCW[6], mientras que Leap Trainer incide en las áreas de HCI y *Machine Learning*.

El soporte de programación (ver tercera columna del Cuadro 2.2) que ofrecen los sistemas analizados, sirve para que el programador pueda extender la funcionalidad de sus aplicaciones. En el análisis de los trabajos relacionados se encuentran tres tipos de sistemas: 1) los sistemas que proporcionan un *toolkit*, 2) los sistemas que permiten exportar o importar módulos y 3) los sistemas que no ofrecen ningún tipo de soporte al programador.

Proximity Toolkit y Kinect Arms, son sistemas que facilitan un *toolkit*. En particular, Proximity Toolkit proporciona un sistema cliente/servidor y algunas herramientas que permiten visualizar la información analizada por el sistema, mientras que Kinect Arms otorga una variedad de funciones para el control del espacio de trabajo de la aplicación.

[5]Human Computer Interaction (HCI) por sus siglas en inglés
[6]Computer Supported Cooperative Work (CSCW) por sus siglas en inglés

Por otra parte, Leap Trainer y SprBlender son sistemas que no proporcionan un *toolkit*, pero permiten importar y exportar módulos del sistema. Leap Trainer permite a los programadores realizar un gesto y exportar el código del mismo y SprBlender posibilita la opción de importar módulos que permitan al espacio de trabajo virtual realizar cambios que el sistema, por omisión, no puede.

Finalmente, 3D Helping Hands y Virtual Keyboard son trabajos que no dan soporte a los programadores.

El lenguaje de programación (ver cuarta columna del Cuadro 2.2) y el dispositivo de rastreo que se utilizó (ver quinta columna del Cuadro 2.2) son relevantes, debido a que la mayoría de los sistemas que utilizan Kinect solo pueden estar escritos en .NET, mientras que los sistemas que utilizan Leap Motion tienen una amplia selección de lenguajes de programación a elegir. Entonces, los sistemas que utilizan Kinect como cámara de profundidad y están programados en C# son 3D Helping Hands, Proximity Toolkit, Virtual Keyboard y Kinect Arms. Por su parte, Proximity Toolkit utiliza otro lenguaje .NET para el sistema cliente/servidor y las herramientas que proporciona. Finalmente, los sistemas Leap Trainer y SprBlender, que utilizan Leap Motion para controlar la aplicación, están programados en JavaScript y Python respectivamente.

| Sistema | Contexto de investigación | Soporte a los programadores | Lenguaje de programación | Dispositivo utilizado |
|---|---|---|---|---|
| Leap Trainer | HCI y *Machine Learning* | Entrega código en JavaScript del gesto creado | JavaScript | Leap Motion |
| SprBlender | HCI y CSCW | Facilita módulos en Python pero la aplicación no permite exportar código | Python | Leap Motion |
| 3D Helping Hands | HCI y CSCW | No da soporte | C# | Kinect |
| Proximity Toolkit | HCI y CSCW | Facilita un *toolkit* | C# / .NET | Kinect |
| Virtual Keyboard | HCI y CSCW | No da soporte | C# | Kinect |
| Kinect Arms | HCI y CSCW | Facilita un *toolkit* | C# | Kinect |

Cuadro 2.2: Tabla comparativa de los trabajos relacionados

# Capítulo 3

# Análisis y diseño del *toolkit*

Según Gamma et al., un *toolkit* es un conjunto de clases reusables y relacionadas, diseñadas para proveer funcionalidades de propósito general que resuelven problemas específicos de una aplicación. También, mencionan que un *toolkit* es independiente de la arquitectura de las aplicaciones, pues debe estar desarrollado para ser usado en muchas aplicaciones. Por tanto, un *toolkit* es flexible y consecuentemente, aplicable y efectivo [8].

Dumas y Redish advierten que el producto, en nuestro caso un *toolkit*, debe permitir a las personas la posibilidad de hacer sus tareas de forma rápida y fácil [9]. Para el caso del *toolkit* propuesto en este trabajo de tesis, se pretende que los programadores puedan crear aplicaciones de manera más rápida y fácil, pues se les proporcionarán las herramientas necesarias para permitir la interacción entre aplicaciones y usuarios mediante dispositivos Leap Motion.

Para determinar el conjunto de funciones de nuestro *toolkit*, es necesario pensar en la funcionalidad que ofrecerá, la cual debe resolver los principales problemas que se presentan al momento de compartir el espacio de trabajo de una aplicación entre múltiples usuarios. Tales problemas son:

- identificación de los usuarios en el espacio de trabajo,
- manipulación de la información de los usuarios conectados,
- control del espacio de trabajo de la aplicación.

Estos problemas pueden ser resueltos de la siguiente manera:

- otorgar un identificador a cada usuario cuando se conecte para mantener un control de su dispositivo.
- permitir la agregación, modificación, eliminación y búsqueda de usuarios.

- establecer mecanismos, e.g., señas, que permitan a los usuarios interactuar con el espacio de trabajo de la aplicación.

Los puntos anteriores influyen en el diseño de nuestro *toolkit*, el cual tiene el propósito de servir en la creación de soportes de interacción, mediante el uso de dispositivos Leap Motion.

Es importante el hecho de que los sistemas colaborativos actuales se enfrentan a problemas de control y manejo más intuitivo del espacio colaborativo, pues la interacción mediante gestos, es una parte importante de la configuración y manejo de un sistema colaborativo, debido a que facilita el trabajo a los usuarios del mismo sistema [17, 19]. Aunque el uso de los gestos debe darse de manera sencilla, no es así con su interpretación, ya que depende del hardware que se tenga para capturarlos y sobre todo el software que los interpreta. Muchas veces los gestos son muy difíciles de capturar, interpretar y enviar, lo que los vuelve costosos, no solo en tiempo sino en la tecnología para darles seguimiento [17, 18].

Por lado, las señas son más sencillas de realizar, ya que no se necesitan movimientos largos o entrelazados ni grandes cantidades de tiempo y esfuerzo de procesamiento para capturarlas e interpretarlas. En esencia, son menos costosas. Aunque se cree que señas y gestos significan lo mismo, en el idioma de señas para personas con pérdida de audición, una seña es una sola palabra, una unidad lingüística, mientras que un gesto es la composición de varias expresiones que pueden ser realizadas con diversas partes del cuerpo[1], con base en la investigación y experimentación realizada en este trabajo de tesis, los gestos están compuestas por duración, sentido, dirección y, en algunos casos, otros gestos y/o señas; mientras que una seña carece de duración y composición de otras señas (en la Sección 3.5.1 se explican los componentes de una seña). Un ejemplo de gesto sería mover el brazo derecho y tocarse la cabeza, mientras que una seña podría ser simplemente mostrar la palma. Es debido al poco procesamiento requerido y a las cualidades antes mencionadas que se optó por el manejo y detección de señas en vez de gestos.

En la Sección 3.1 se presentan los diagramas de casos de uso del *toolkit* desarrollado. Más adelante, en la Sección 3.2, se describe de manera general las APIs que ofrece dicho *toolkit*. Posteriormente, en la Sección 3.3, se explican cada una de las funciones que forman parte del control de usuarios. Las funciones de procesamiento de dedos y vectores son presentadas en la Sección 3.4. Finalmente, en la sección 3.5, se exponen las funciones utilizadas para generar información o código en lenguaje C de señas interpretadas.

[1] http://cenarec-lesco.org/index.php/gramar/personal-68/algunos-conceptos-necesarios/senas-y-gestos

# 3.1. Caso de uso del *toolkit*

Los casos de uso del *toolkit* desarrollado están divididos en tres grupos:

1. casos de uso para el control de usuarios (ver Sección 3.1.1)
2. casos de uso para el procesamiento de dedos y vectores (ver Sección 3.1.2)
3. casos de uso para la interpretación de señas (ver Sección 3.1.3)

## 3.1.1. Casos de uso para el control de usuarios

En la Figura 3.1 se aprecia el diagrama que contiene seis casos de uso para el control de usuarios, los cuales son: 1) agregar un usuario a la lista de usuarios conectados y posteriormente asignarle un identificador, 2) consultar un identificador de usuario, 3) eliminar un usuario de la lista de usuarios conectados, 4) obtener la lista de usuarios conectados para visualizar sus datos, 5) contar el número de usuarios conectados y 6) verificar si la lista de usuarios conectados se encuentra vacía.

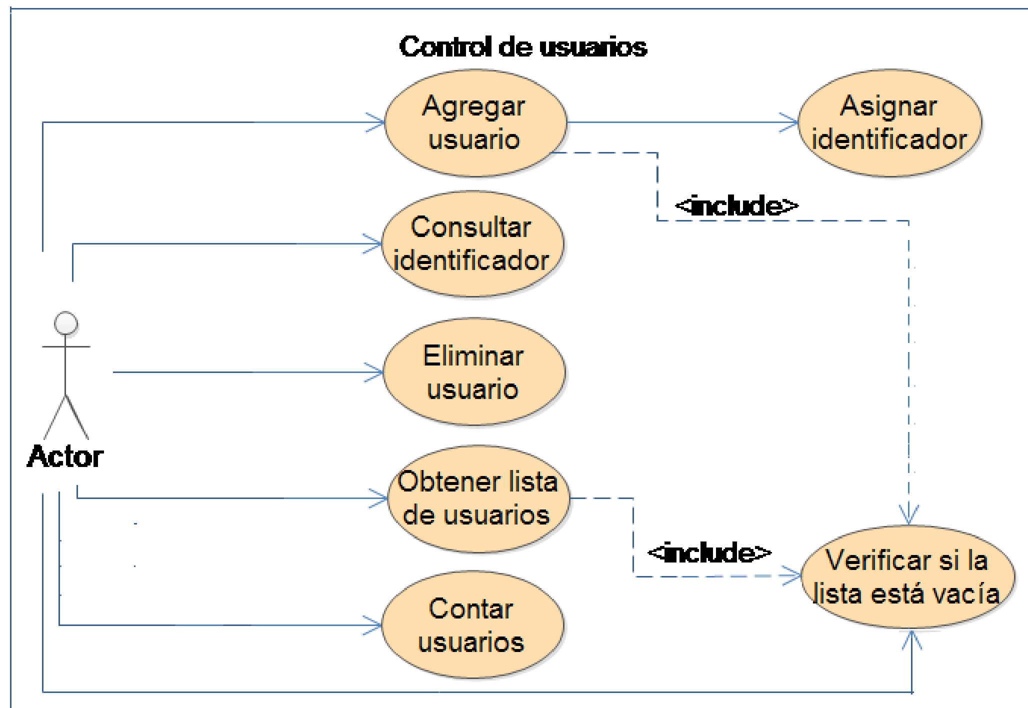


Figura 3.1: Casos de uso para el control de usuarios

El Cuadro 3.1 describe agregación de un usuario a la lista de usuarios conectados y, en el Cuadro 3.2, se explica la manera en la que el programador puede obtener el identificador de un usuario específico. La eliminación de un usuario de dicha lista se presenta en el Cuadro 3.3. Por otra parte, el Cuadro 3.4 se refiere a la visualización de toda la información correspondiente a cada usuario. En el Cuadro 3.5 se muestra el caso de uso para conocer el número de usuarios conectados. Finalmente, el Cuadro 3.6 explica los pasos para averiguar si la lista de usuarios conectados está vacía.

| CASO DE USO: | Agregar un usuario |
|---|---|
| ACTOR: | Programador |
| DESCRIPCIÓN: | Agrega un usuario a la lista y asignarle un identificador |
| PRECONDICIONES: | Ser un nuevo usuario. |
| CURSO NORMAL: | ALTERNATIVAS: |
| 1) Recibir los datos del usuario | |
| 2) Agregar un usuario a la lista | |
| 3) Devolver el identificador asignado | |

Cuadro 3.1: Caso de uso para agregar un usuario a la lista de usuarios conectados

| CASO DE USO: | Consultar un identificador |
|---|---|
| ACTOR: | Programador |
| DESCRIPCIÓN: | Devuelve el identificador asignado a un usuario |
| PRECONDICIONES: | El usuario debe estar registrado en la lista de usuarios conectados |
| CURSO NORMAL: | ALTERNATIVAS: |
| 1) Recibir los datos del usuario | |
| 2) Buscar al usuario en la lista | |
| 3) Devolver el identificador del usuario | |

Cuadro 3.2: Caso de uso para consultar un identificador asociado a un usuario conectado

| Caso de uso: | Eliminar un usuario |
|---|---|
| Actor: | Programador |
| Descripción: | Elimina un usuario de la lista de usuarios conectados |
| Precondiciones: | El usuario a eliminar debe estar registrado en la lista de usuarios conectados |
| Curso normal: | Alternativas: |
| 1) Recibir los datos del usuario | |
| 2) Eliminar al usuario de la lista | |

Cuadro 3.3: Caso de uso para eliminar un usuario de la lista de usuarios conectados

| Caso de uso: | Obtener la lista de usuarios |
|---|---|
| Actor: | Programador |
| Descripción: | Obtiene la lista de usuarios conectados |
| Precondiciones: | La lista debe contener al menos un usuario conectado |
| Curso normal: | Alternativas: |
| 1) Obtiene los datos de cada usuario | |

Cuadro 3.4: Caso de uso para obtener los datos de cada usuario registrado en la lista de usuarios conectados

| Caso de uso: | Contar usuarios |
|---|---|
| Actor: | Programador |
| Descripción: | Devuelve el total de usuarios conectados |
| Precondiciones: | |
| Curso normal: | Alternativas: |
| 1) Devolver el número de usuarios conectados | |

Cuadro 3.5: Caso de uso para contar los usuarios conectados

| Caso de uso: | Verificar si la lista está vacía |
|---|---|
| Actor: | Programador |
| Descripción: | Verifica si la lista de usuarios conectados está vacía |
| Precondiciones: | |
| Curso normal: | Alternativas: |
| 1) Devolver verdadero si la lista de usuarios conectados está vacía | Devolver falso si la lista de usuarios conectados no está vacía |

Cuadro 3.6: Caso de uso para verificar si la lista de usuarios conectados está vacía

### 3.1.2. Casos de uso para el procesamiento de dedos y vectores

En la Figura 3.2 se muestran los casos de uso para el procesamiento de dedos y vectores. Los tres casos de uso que contiene explican el comportamiento del *toolkit* cuando se requiere: 1) para detectar una colisión entre un punto y un rectángulo , 2) conocer el total de dedos extendidos de una o ambas manos del usuario y 3) obtener el cuadrante de cada dedo del usuario.

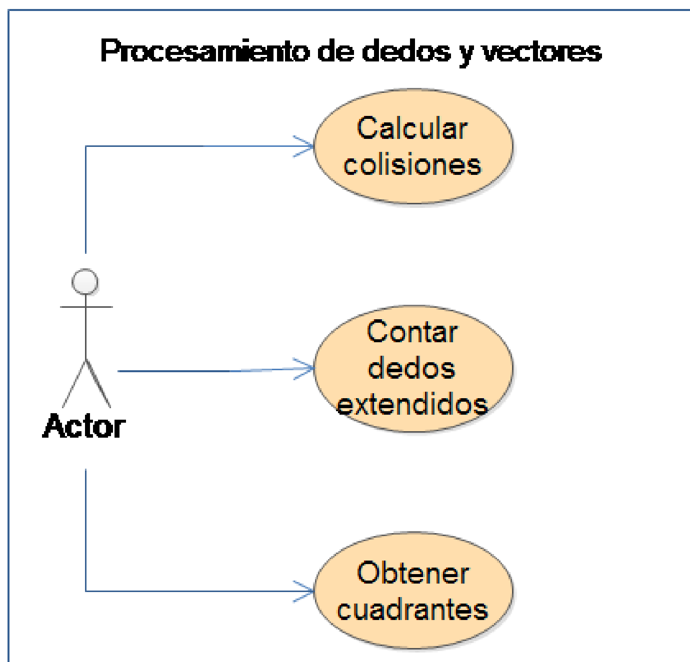


Figura 3.2: Casos de uso para el procesamiento de dedos y vectores

El Cuadro 3.7 explica el caso de uso para calcular la colisión entre un punto, obtenido de un vector, y un rectángulo formado por cuatro puntos. En el Cuadro 3.8 se exponen los pasos para que el *toolkit* entregue el total de dedos extendidos de una o ambas manos del usuario. Finalmente, en el Cuadro 3.9, se presenta el comportamiento del *toolkit* al momento de obtener el cuadrante hacia donde apunta cada dedo del usuario.

| Caso de uso: | Calcular colisiones |
|---|---|
| Actor: | Programador |
| Descripción: | Calcula si ha ocurrido una colisión entre un punto y un rectángulo. |
| Precondiciones: | Tener un vector (punto) y cuatro enteros para construir un rectángulo |
| Curso normal: | Alternativas: |
| 1) Recibir un vector (punto) y cuatro enteros (rectángulo) | |
| 2) Calcular si existe una colisión entre el vector (punto) y los cuatro enteros (rectángulo) | |
| 3) Devolver '1' si existe colisión | Devolver '0' si no existe colisión |

Cuadro 3.7: Caso de uso para calcular una colisión entre un punto y un rectángulo

| Caso de uso: | Contar dedos extendidos |
|---|---|
| Actor: | Programador |
| Descripción: | Devuelve el total de dedos extendidos de una o ambas manos del usuario |
| Precondiciones: | Tener la lista de una o ambas manos del usuario |
| Curso normal: | Alternativas: |
| 1) Recibir la lista de una o ambas manos del usuario | |
| 2) Devolver el número total de dedos extendidos de ambas manos | Devolver el número total de dedos extendidos de una mano |

Cuadro 3.8: Caso de uso para contar los dedos de una o ambas manos de un usuario

| Caso de uso: | Obtener cuadrantes |
|---|---|
| Actor: | Programador |
| Descripción: | Calcula hacia qué cuadrante apunta cada dedo del usuario |
| Precondiciones: | Tener la lista de los dedos de una o ambas manos del usuario y tener un apuntador hacia un arreglo para almacenar los datos |
| Curso normal: | Alternativas: |
| 1) Recibir la lista de dedos del usuario y un apuntador del arreglo donde se guardarán los datos | |
| 2) Calcular el cuadrante de cada dedo del usuario | |
| 3) Almacenar los datos en el arreglo | |

Cuadro 3.9: Caso de uso para calcular hacia qué cuadrante apunta cada dedo del usuario

### 3.1.3. Casos de uso para la interpretación de señas

En la Figura 3.3 se presenta el diagrama de casos de uso para la interpretación de señas, el cual contiene tres casos de uso pertenecientes a este grupo y dos casos de uso referentes al procesamiento de dedos y vectores. Los casos de uso para la interpretación de señas detallan el comportamiento del *toolkit* al momento de que el programador solicita el código en lenguaje C o la información de las señas realizadas por el usuario.

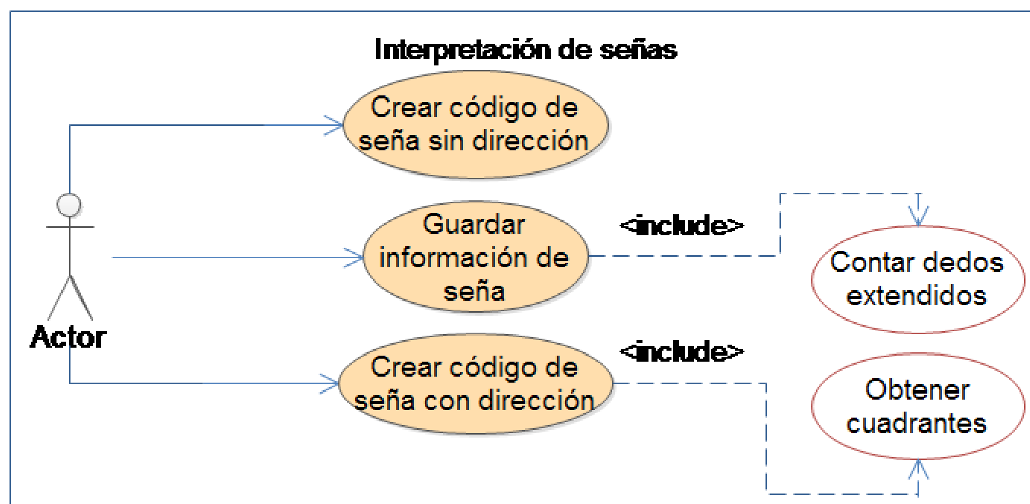


Figura 3.3: Casos de uso para la interpretación de señas

En el Cuadro 3.10 se muestran los pasos que se siguen para entregar una función, en código de lenguaje C, de una seña sin dirección realizada por un usuario (i.e. sin información de los cuadrantes hacia donde apuntan los dedos extendidos de una o ambas manos del usuario). El código de la seña sin dirección interpretada se guarda en un documento de texto nombrado de la misma forma que la función creada.

Posteriormente, en el Cuadro 3.11 se presenta el caso de uso para detallar cómo el programador puede almacenar, en un documento de texto, la información de una seña realizada por un usuario.

Finalmente, en el Cuadro 3.12 se exponen los pasos que debe seguir el programador para crear y nombrar una seña y su respectiva función en código de lenguaje C, en la cual cada uno de los dedos extendidos del usuario apunta a un cuadrante (i.e. un cuadrante para cada dedo extendido de una o ambas manos del usuario). El código de la seña con dirección es almacenada en un documento de texto nombrado de la misma forma que la función creada.

| | |
|---|---|
| CASO DE USO: | Crear código de seña sin dirección |
| ACTOR: | Programador |
| DESCRIPCIÓN: | Guarda en un documento de texto el código en lenguaje C de una seña sin dirección realizada por el usuario |
| PRECONDICIONES: | Tener la lista de una o ambas manos del usuario. El actor debe elegir un nombre para la seña y para el documento de texto |
| CURSO NORMAL: | ALTERNATIVAS: |
| 1) Recibir la lista de una o ambas manos de un usuario | |
| 2) Crear el código de la seña | |
| 3) Nombrar la seña | |
| 4) Guardar el código C de una seña sin dirección realizada con una mano. | Guardar el código C de una seña sin dirección realizada con ambas manos. |
| 4.1) Mano específica del usuario. | Mano cualquiera del usuario. |

Cuadro 3.10: Caso de uso para crear código en lenguaje C de una seña sin dirección

| Caso de uso: | Guardar información de seña |
|---|---|
| Actor: | Programador |
| Descripción: | Guarda en un documento de texto la información de una seña realizada por el usuario |
| Precondiciones: | Tener la lista de una o ambas manos del usuario. El actor debe elegir un nombre para el documento de texto |
| Curso normal: | Alternativas: |
| 1) Recibir la lista de una o ambas manos del usuario | |
| 2) Guardar en el documento de texto la información de la seña | |

Cuadro 3.11: Caso de uso para guardar en un documento de texto la información de una seña realizada por el usuario

| Caso de uso: | Crear código de seña con dirección |
|---|---|
| Actor: | Programador |
| Descripción: | Guarda el código C de una seña con dirección realizada por el usuario |
| Precondiciones: | Tener la lista de una o ambas manos del usuario. El actor debe elegir un nombre para la seña creada y para el documento de texto |
| Curso normal: | Alternativas: |
| 1) Recibir la lista de una o ambas manos de un usuario | |
| 2) Crear el código de la seña | |
| 3) Nombrar la seña | |
| 4) Guardar el código C de la seña con dirección realizada con una mano. | Guardar el código C de la seña con dirección realizada con ambas manos. |
| 4.1) Mano específica del usuario. | Mano cualquiera del usuario. |

Cuadro 3.12: Caso de uso para crear código en lenguaje C de una seña con dirección

## 3.2. APIs del *toolkit*

El *toolkit* desarrollado ofrece funciones que aseguran el control de cada dispositivo Leap Motion utilizado por cada uno de los usuarios. Además, provee herramientas para la interpretación de señas y el procesamiento de los datos de dichas señas que permiten a los usuarios controlar el espacio de trabajo.

Las funciones del *toolkit* desarrollado están clasificadas en tres APIs, las cuales corresponden a cada uno de los grupos de casos de uso descritos en la Sección 3.1 (ver Figura 3.4) y son las siguientes:

1. API de control de usuarios

2. API de procesamiento de dedos y vectores

3. API de interpretación de señas

La primera API contiene funciones que permiten añadir usuarios en la lista de usuarios conectados y asígnarles un identificador ($asignarId()$), buscar usuarios ($obtenerId()$), eliminarlos ($borrarId()$), obtener la información de la lista de usuarios conectados ($obtenerLista()$), conocer el número de usuarios conectados ($contarId()$ y verificar si la lista está vacía ($verificarLista()$).

La segunda API está compuesta por funciones que permiten al programador procesar datos como son: 1) una lista de una o ambas manos del usuario para averiguar el total de dedos extendidos ($contarDedosExtendidos()$), 2) un vector y un rectángulo para saber si existe una colisión entre un punto y dicho rectángulo ($calcularColision()$) y 3) cuadrantes en el plano cartesiano hacia los cuales podría apuntar cada uno de los dedos del usuario ($obtenerCuadranteDedo()$).

Por último, la tercera API está compuesta por tres funciones. La primera permite al programador obtener código en lenguaje C, como resultado de la abstracción de una seña sin dirección ($generarSeniaSinDir()$), i.e. sin tomar en cuenta la información del cuadrante de cada dedo extendido del usuario. La segunda función sirve para que el programador pueda almacenar, en un documento de texto, la información de una seña realizada por el usuario ($generarInfoSenia()$). La tercera función permite generar el código en lenguaje C de la interpretación de una seña con dirección realizada por el usuario ($generarSeniaConDir()$), i.e, que considera el cuadrante al que apunta cada dedo extendido del usuario.

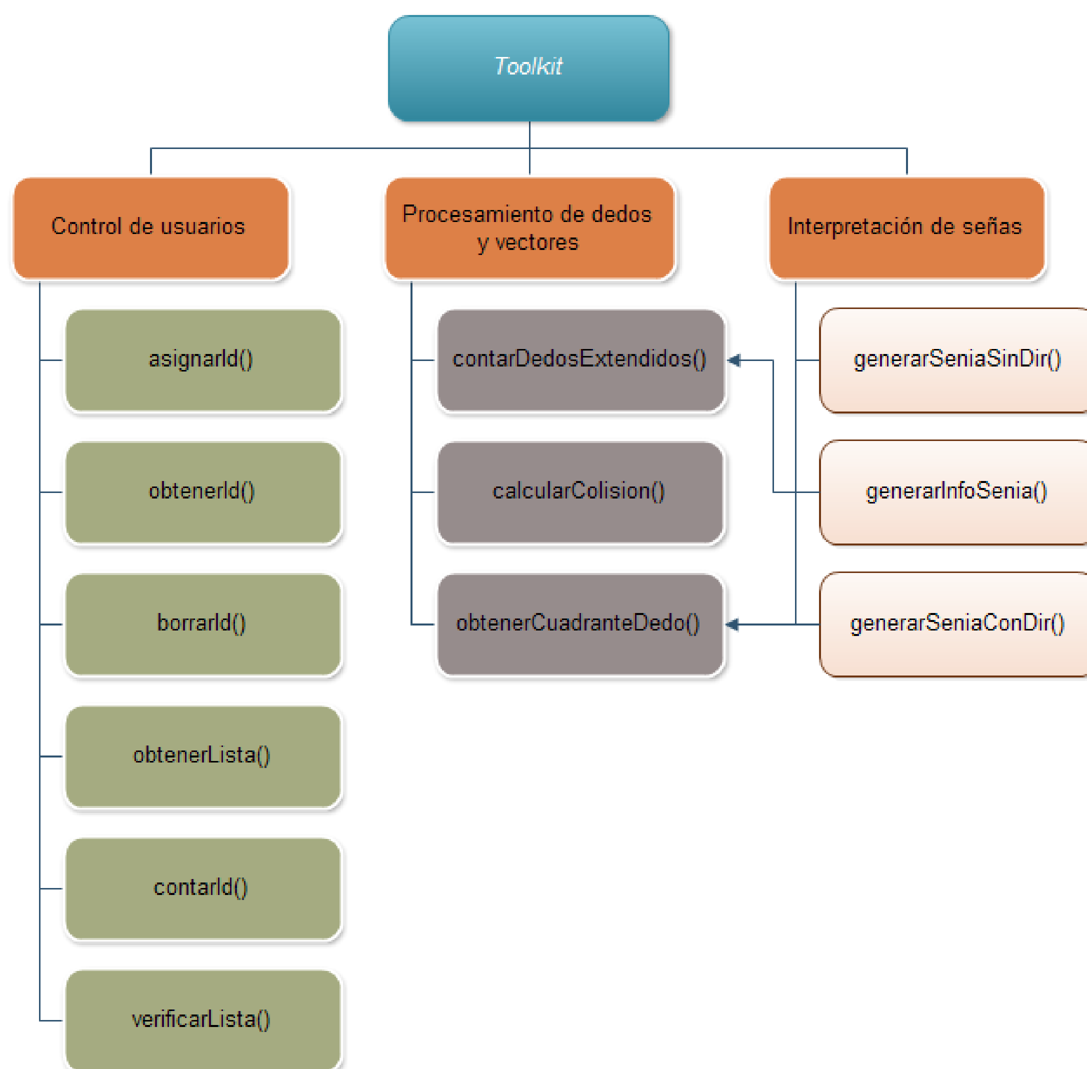


Figura 3.4: APIs del *toolkit* desarrollado

## 3.3. API de control de usuarios

En esta sección se presentan las funciones que manejan identificadores de usuario como: *asignarId*(), *obtenerId*(), *borrarId*(), *obtenerLista*(), *contarId*() y por último, *verificarLista*(), las cuales están basadas en el concepto de manejo de listas, pues ofrecen un mejor control sobre los datos del usuario. Dichas funciones permiten la agregación de usuarios a la lista, la obtención de un identificador asociado a un usuario conectado, la eliminación de usuarios, la adquisición de la lista de usuarios conectados y la verificación de la lista de usuarios. En este caso se utiliza una lista doblemente enlazada, la cual es lineal y posee nodos. Cada nodo tiene un enlace que apunta al nodo siguiente y uno que apunta al nodo anterior.

En la Figura 3.5 se muestra un ejemplo de cómo trabaja una lista doblemente ligada, donde la lista de usuarios conectados cuenta con cuatro nodos y cada nodo contiene la información necesaria (e.g., dirección IP e identificador) de cada usuario conectado.

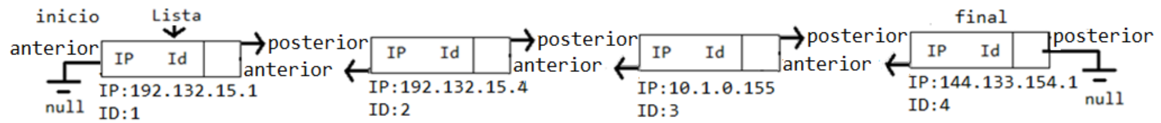


Figura 3.5: Lista doblemente enlazada con datos de usuarios conectados

La lista es iniciada con dos apuntadores de la estructura "Lista" llamados *inicio* y $final$, que sirven de ayuda para marcar el inicio y el final de la lista, respectivamente. Cada nodo tiene dos apuntadores llamados *anterior* y *posterior*, para apuntar al nodo anterior y al nodo siguiente, respectivamente. La lista es almacenada en memoria secundaria y los datos son destruidos cuando la sesión del sistema termina. Los identificadores no son reasignables mientras la sesión del sistema esté activa pero, cuando la sesión termina, los identificadores pueden ser reasignados. El número de identificador se guarda en una variable global llamada $_ID$, la cual se incrementa cada vez que se agrega un nuevo usuario a la lista. También se almacena el número de usuarios conectados en una variable global denominada $_cont$, la cual aumenta al momento de agregar un nuevo usuario y disminuye cuando se borra un usuario.

Enseguida se presentan las características de las funciones que componen la API de control de usuarios del *toolkit* desarrollado.

## 3.3.1. Función *asignarId*()

Esta función recibe como parámetro una dirección IP (ver Interfaz 3.1). Se encarga de otorgar un nuevo identificador (int) y almacenar en un nuevo nodo los datos de la dirección IP que recibe y del identificador asignado. La función devuelve '-1' si no existe memoria secundaria para crear el nodo nuevo; en caso de que todo resulte exitoso, se incrementa el número de usuarios conectados en la lista y se devuelve el identificador asignado al usuario.

$$int \quad asignarId(char * _dir) \tag{3.1}$$

Cuando un nuevo usuario intenta conectarse, el programador deberá asegurarse de obtener la dirección IP del dispositivo del nuevo usuario y enviarla como parámetro de entrada a la función $asignarId()$ la cual devolverá un identificador para el nuevo usuario.

Al momento de agregar registros de usuarios a la lista de usuarios conectados existen dos casos a considerar: 1) que el nuevo nodo sea el primero en la lista de usuarios conectados (ver Figura 3.6) y 2) que existan otros usuarios conectados (ver Figura 3.7).

Si la lista de usuarios conectados se encuentra vacía, se crea el primer nodo (ver Figura 3.6). La información almacenada en este nodo es $_dir$ = “192.132.15.1” e $_ID = 1$ para el primer usuario conectado y los apuntadores *anterior* y *posterior* apuntan a *null*. De igual forma los apuntadores *inicio* y $final$ de la lista de usuarios conectados apuntan hacia el primer nodo de la lista.

Figura 3.6: Primer nodo en la lista de usuarios conectados

En la Figura 3.7 se agrega el registro de un nuevo usuario a la lista de usuarios conectados, la cual que actualmente contiene tres usuarios conectados. La información que se almacenará en el nuevo nodo es $_dir$ = “144.133.154.1” e $_ID = 4$. Como el nuevo nodo no es el primero en la lista de usuarios conectados, se posiciona al final de la lista, su apuntador *anterior* se asocia al nodo precedente (i.e. usuario con $_ID = 3$) y *posterior* apunta a *null*. Por otro lado, el apuntador $final$ se posiciona al final de la lista (i.e. usuario con $_ID = 4$).

Figura 3.7: Agregación de un nodo a la lista de usuarios conectados

En la Figura 3.8 se aprecia el diagrama de flujo para guardar el registro de un nuevo usuario en la lista de usuarios conectados.

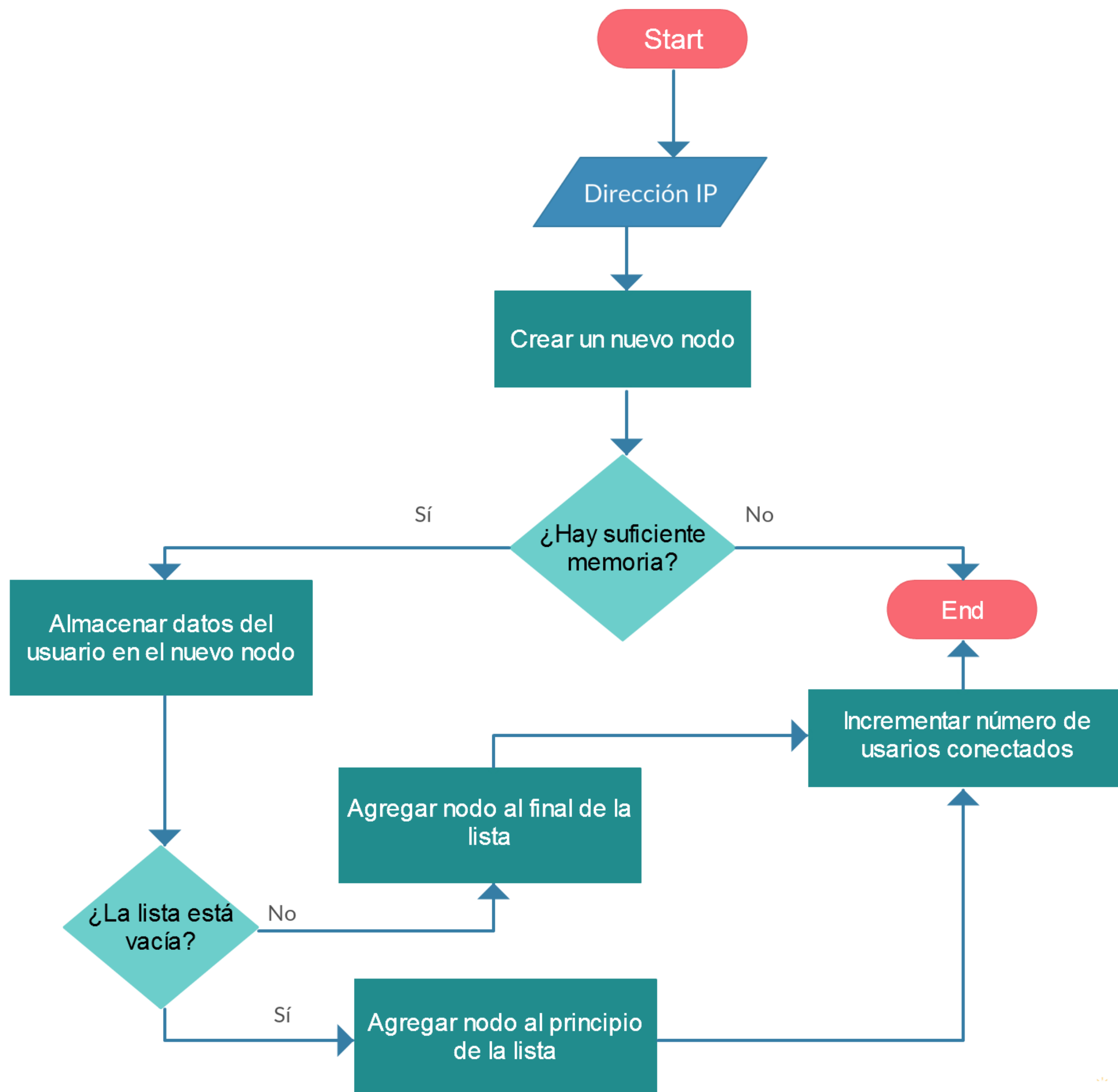


Figura 3.8: Asignación de nuevo identificador

### 3.3.2. Función *obtenerId*()

Esta función recibe como parámetro una dirección IP (ver Interfaz 3.2). Tiene como propósito buscar un usuario dentro de la lista de usuarios conectados. En caso de encontrar algún dato que coincida con la dirección IP, retorna el identificador (int) asignado al usuario de dicha dirección que coincidió durante la búsqueda; en caso de no encontrar coincidencias, retorna '0'.

$$int \quad obtenerId(char * _dir) \tag{3.2}$$

En la Figura 3.9, se muestra la lista de usuarios conectados con cuatro nodos en los cuales se buscará, uno a uno, que exista coincidencia con *_dir*= "10.1.0.155". Para este caso de ejemplo, la coincidencia se encuentra en el tercer nodo, entonces solo se debe regresar el $_ID = 3$ asociado a dicho nodo.

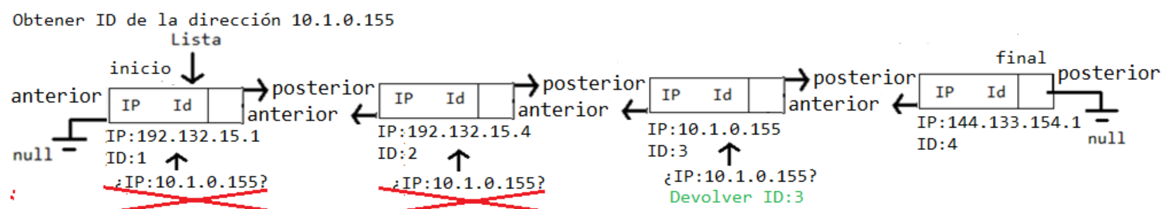


Figura 3.9: Búsqueda exitosa de una dirección IP dentro de la lista de usuarios conectados

En la Figura 3.10 se observa la lista de usuarios conectados con cuatro nodos en los cuales se buscará que exista coincidencia entre *_dir* = "10.1.250.3" y la dirección IP asociada a un nodo. Para este ejemplo, no se encontraron coincidencias, por lo cual la función *obtenerId*() devuelve '0'.

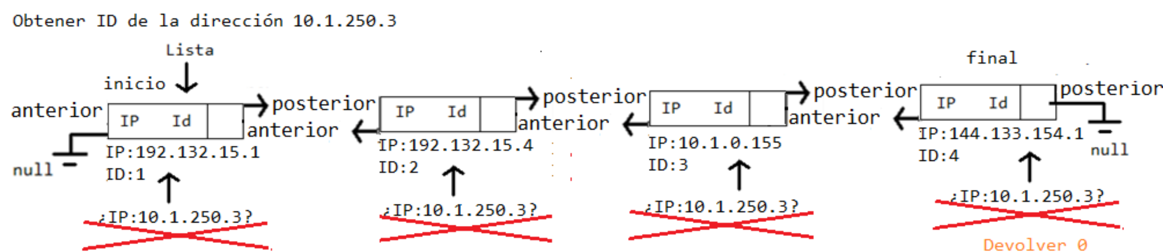


Figura 3.10: Búsqueda fallida de una dirección IP dentro de la lista de usuarios conectados

En la Figura 3.11 se presenta el diagrama de flujo de cómo se realiza una búsqueda de una dirección IP dentro de la lista de usuarios conectados.

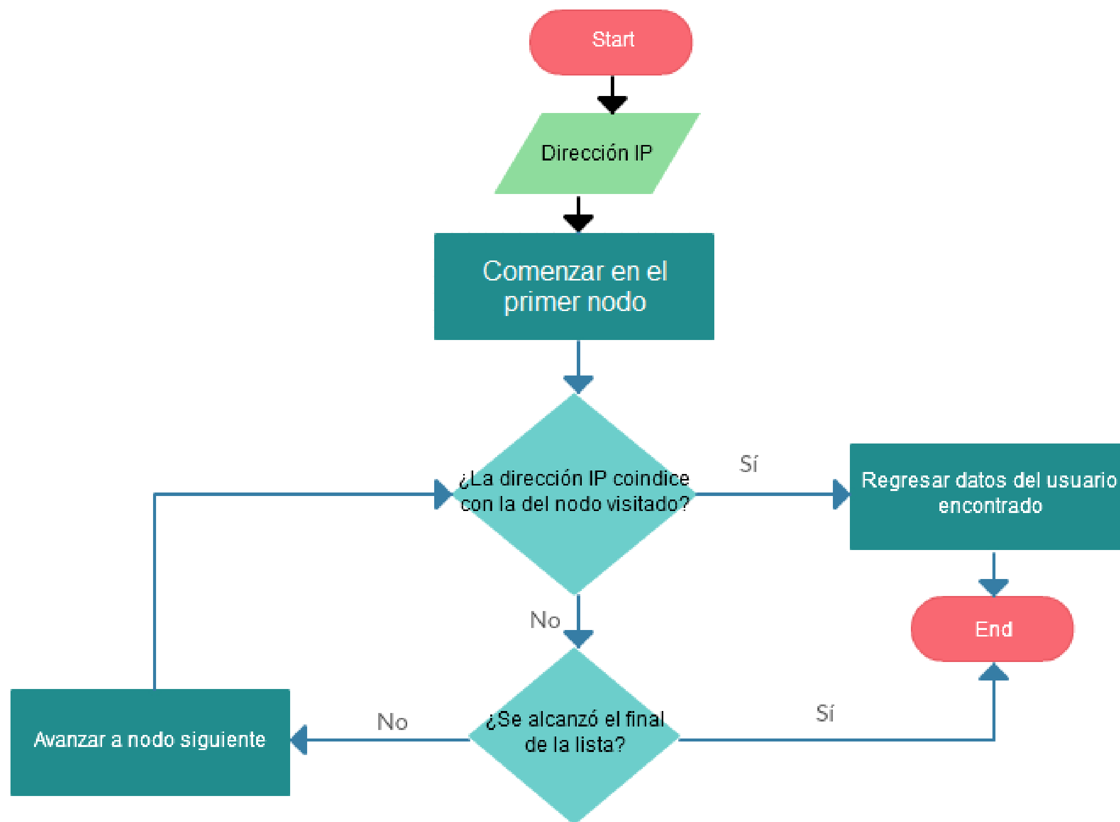


Figura 3.11: Búsqueda de un usuario en la lista de usuarios conectados

### 3.3.3. Función *borrarId*()

Esta función recibe un identificador (ver Interfaz 3.3) como parámetro de entrada. La función tiene como propósito eliminar el registro de un usuario de la lista de usuarios conectados. En caso de tener éxito devuelve '1' y '0' en caso de no encontrar registro del usuario en la lista.

$$int \quad borrarId(int \quad _id) \tag{3.3}$$

El procedimiento para eliminar el registro de un usuario dentro de la lista de usuarios conectados consiste en posicionarse al inicio de la lista y buscar en cada nodo una coincidencia entre el identificador (int) del usuario que se quiere eliminar y el identificador (int) asociado al nodo.

En la Figura 3.12 se muestra una lista de cuatro registros de usuarios, de los cuales se eliminará uno. Para este ejemplo se propone eliminar el primer nodo. Una vez eli-

minado este nodo, se actualiza el apuntador *inicio* de la lista, así como el apuntador *anterior* del nodo que ahora es el primero en la lista de usuarios conectados.

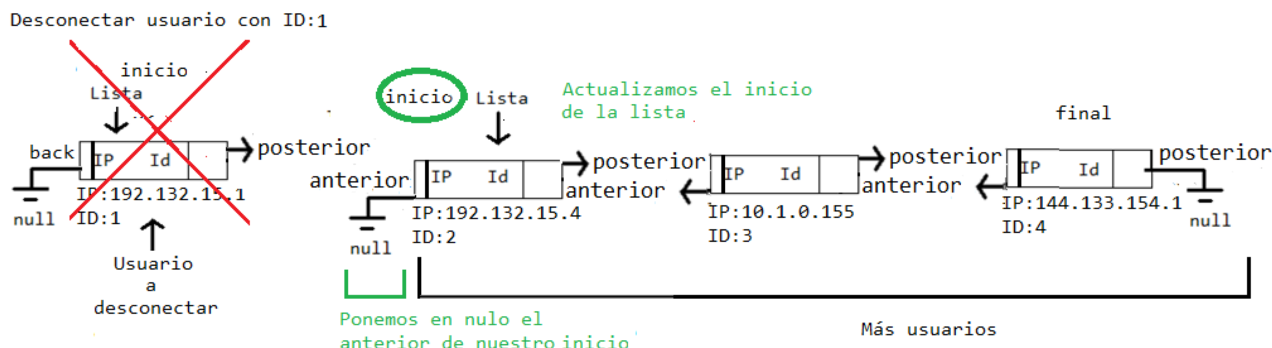


Figura 3.12: Eliminación del primer usuario de la lista

En la Figura 3.13 se aprecia la lista de usuarios conectados con cuatro registros de usuarios, de los cuales se debe eliminar uno de ellos, en este caso el último. Una vez eliminado este nodo, se actualiza el apuntador *final* de la lista y el apuntador *posterior* del nodo que ahora es el último en la lista de usuarios conectados.

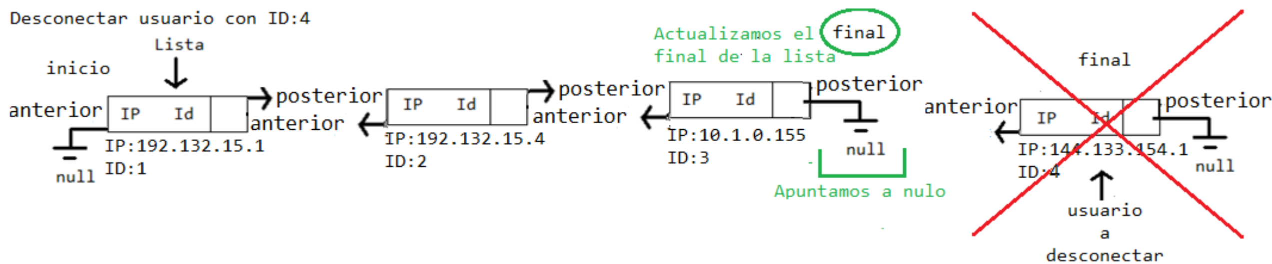


Figura 3.13: Eliminación del último usuario de la lista

En la Figura 3.14 se presenta la lista de usuarios conectados con cuatro registros de usuarios, de los cuales se eliminará uno, en este caso el tercer nodo. Una vez eliminado dicho nodo, se actualiza el apuntador *posterior* del nodo anterior al nodo eliminado (segundo nodo), así como el apuntador *anterior* del nodo que era siguiente al nodo eliminado (nodo final).

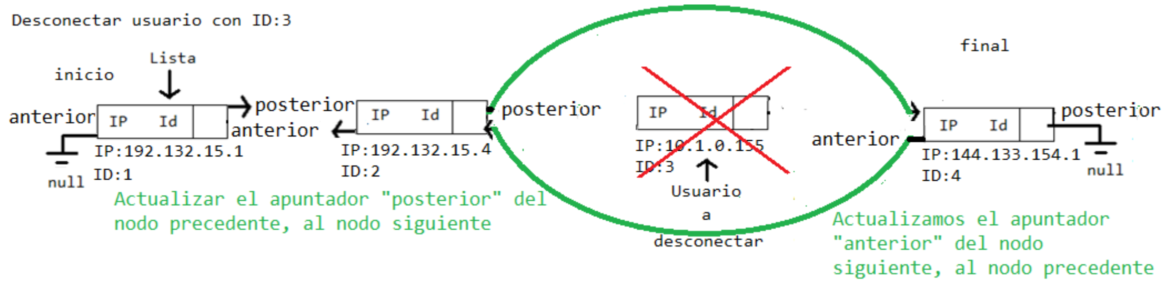


Figura 3.14: Eliminación del registro de un usuario que no está al inicio o al final de la lista

En la Figura 3.15 se presenta un diagrama de flujo que muestra la forma en la que el registro de un usuario es eliminado de la lista de usuarios conectados.

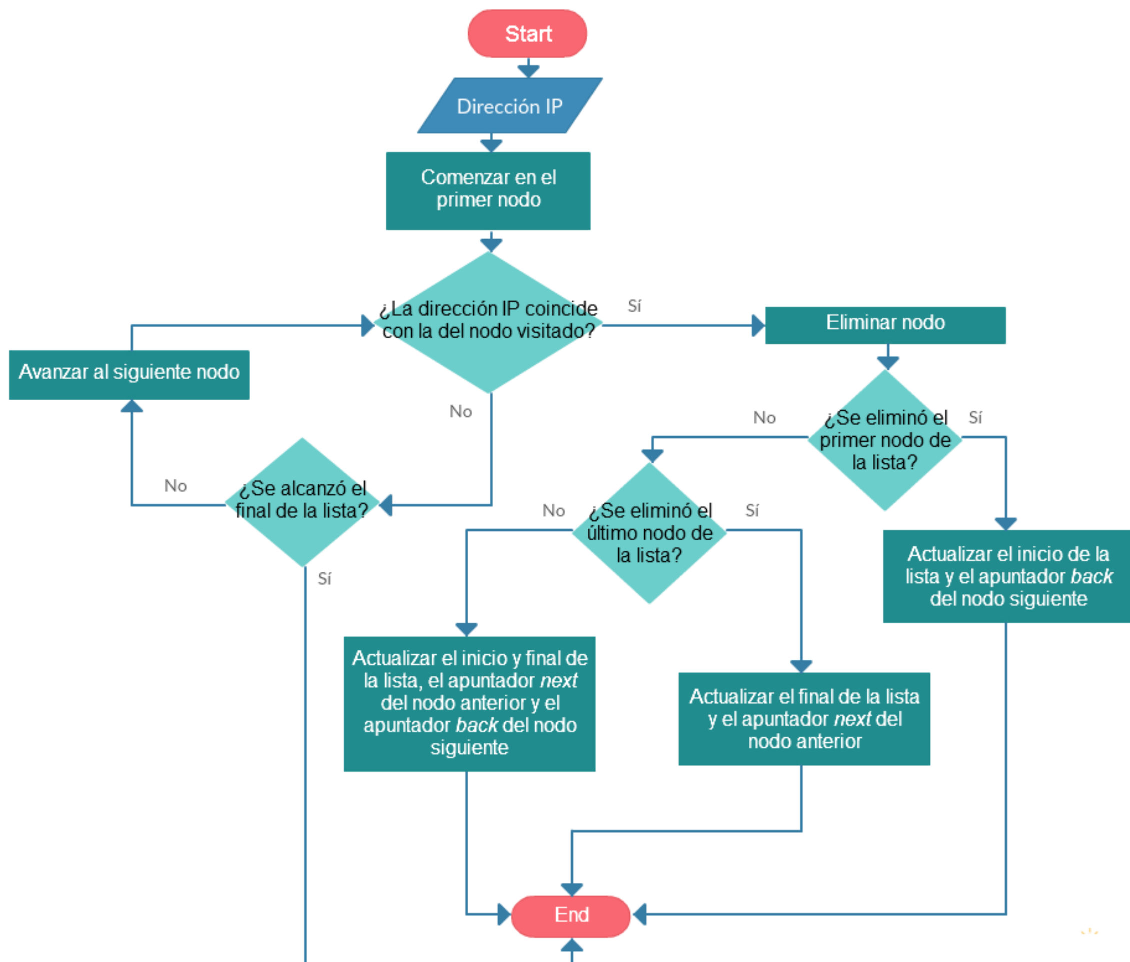


Figura 3.15: Eliminación del registro de un usuario de la lista de usuarios conectados

### 3.3.4. Función *obtenerLista*()

Esta función no recibe parámetros (ver Interfaz 3.4). Se encarga de obtener la lista de usuarios conectados. Devuelve '0' en caso de que no se encuentren usuarios y '1' en caso contrario.

$$int \quad obtenerLista() \tag{3.4}$$

En la Figura 3.16 se observa una lista con cuatro registros de usuarios, los cuales serán recorridos para visualizar los datos que contienen (e.g., dirección IP e identificador del usuario) como se muestra en dicha figura.

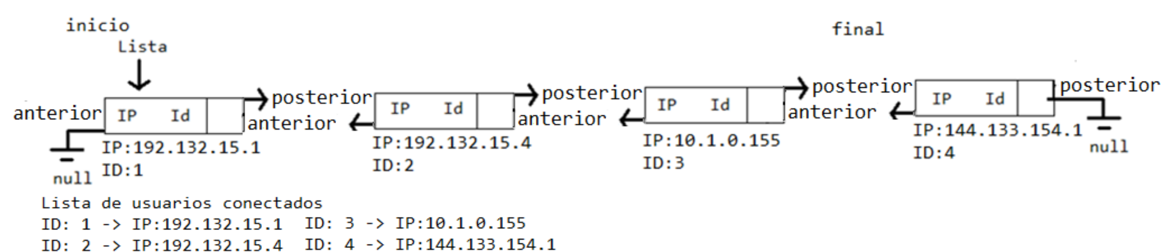


Figura 3.16: Contenido de la lista de usuarios conectados

El funcionamiento para obtener los datos contenidos en la lista de usuarios conectados se muestra en el diagrama de la Figura 3.17.

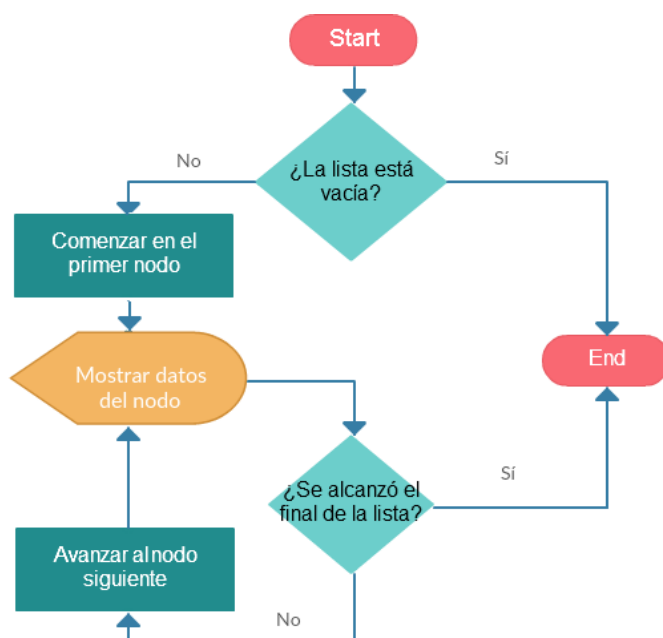


Figura 3.17: Obtención de datos en la lista de usuarios conectados

### 3.3.5. Función *contarId*()

Esta función no recibe parámetros (ver Interfaz 3.5). Devuelve el valor de la variable global llamada *_cont*, la cual contiene el número de usuarios que actualmente estén registrados en la lista de usuarios conectados.

$$int \quad contarId() \tag{3.5}$$

En la Figura 3.18 se representa el diagrama de flujo para saber cuántos usuarios conectados hay en la lista.

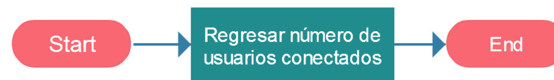


Figura 3.18: Determinación del número de usuarios conectados en la lista

### 3.3.6. Función *verificarLista*()

Esta función no recibe parámetros (ver Interfaz 3.6). Devuelve $false$ si la lista de usuarios conectados contiene al menos un usuario y *true* si la lista no contiene usuarios.

$$bool \quad verificarLista() \tag{3.6}$$

En 3.19 se observa el diagrama de flujo para averiguar si la lista de usuarios conectados está vacía.

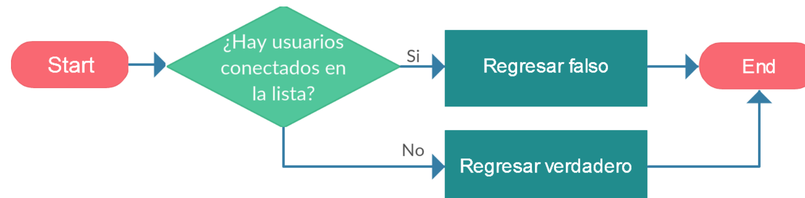


Figura 3.19: Verificación de la lista de usuarios conectados

## 3.4. API de procesamiento de dedos y vectores

En esta API del *toolkit* se presentan las funciones que tienen como objetivo procesar datos (e.g. listas de dedos del usuario y vectores). En la Sección 3.4.1 se presenta la función que permite conocer el número de dedos extendidos de una o ambas manos del usuario. En la Sección 3.4.2 se detalla la función que detecta colisiones entre un punto y un rectángulo (una de las figuras más utilizadas en videojuegos 2D). Finalmente, en la Sección 3.4.3, se explica la función que obtiene los cuadrantes, en un plano cartesiano 2D, hacía los que apunta cada uno de los dedos de una o ambas manos del usuario.

### 3.4.1. Función *contarDedosExtendidos*()

Esta función recibe una lista de las manos del usuario como parámetro (ver Interfaz 3.7). Se encarga de recorrer cada uno de los dedos de una o ambas manos del usuario e incrementar un contador cada que se encuentre un dedo extendido. La función devuelve la suma total de dedos extendidos de ambas manos.

$$int \quad contarDedosExtendidos(HandList \quad _hands) \tag{3.7}$$

El funcionamiento de *contarDedosExtendidos*() se explica en la Figura 3.20.

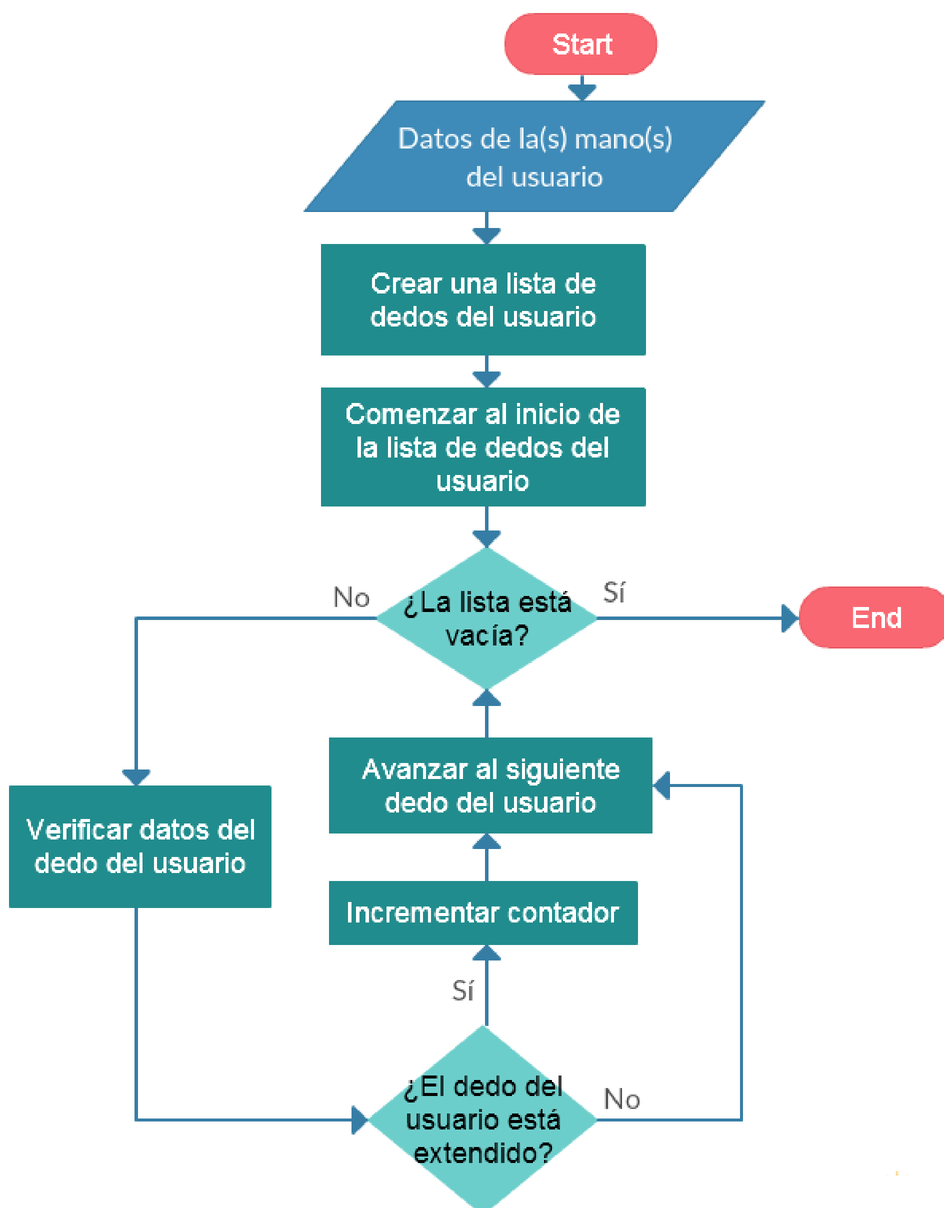


Figura 3.20: Obtención del número de dedos extendidos de una o ambas manos del usuario

Como ejemplo, si la función *contarDedosExtendidos*() recibe una lista de las dos manos del usuario que aparecen en la Figura 3.21, la función devolverá '10', que es el número total de dedos extendidos en ambas manos.

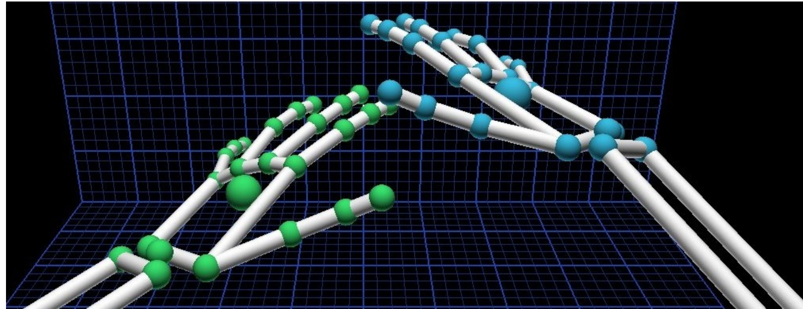

Figura 3.21: Visualización de las dos manos del usuario con diez dedos extendidos

### 3.4.2. Función *calcularColision*()

Esta función recibe como parámetro un objeto *vector* y cuatro números que son las coordenadas de las esquinas superior izquierda e inferior derecha de un rectángulo (ver Interfaz 3.8). Se encarga de calcular, en un mapa cartesiano 2D, si un punto se encuentra en el área delimitada por un rectángulo.

$$bool\ calcularColision(Vector\ _vector, float\ _a, float\ _b, float\ _c, float\ _d) \tag{3.8}$$

El rectángulo es creado a partir de dos puntos, $P1(a, b)$ y $P2(c, d)$, los cuales se obtienen de los cuatro enteros recibidos en la función. El punto $P3(x, y)$, que se utiliza para calcular la colisión, es obtenido a partir de los dos componentes del *vector*, como se muestra en Figura 3.22.

El punto $P3$ entra en contacto con el área del rectángulo si cumple las siguientes condiciones [2]:

1. la coordenada $x$ del *vector* debe ser mayor o igual que la coordenada $a$ del punto $P1$, pero menor o igual que la coordenada $c$ del punto $P2$

2. la coordenada $y$ del *vector* debe ser mayor o igual que la coordenada $b$ del punto $P1$, pero menor o igual que la coordenada $d$ del punto $P2$.

Si ambas condiciones son verdaderas, la función *calcularColision*() regresa *true* y en caso de que alguna no sea verdadera, regresa *false*.

[2] http://www.yaldex.com/game-programming/0131020099_ch22lev1sec1.html

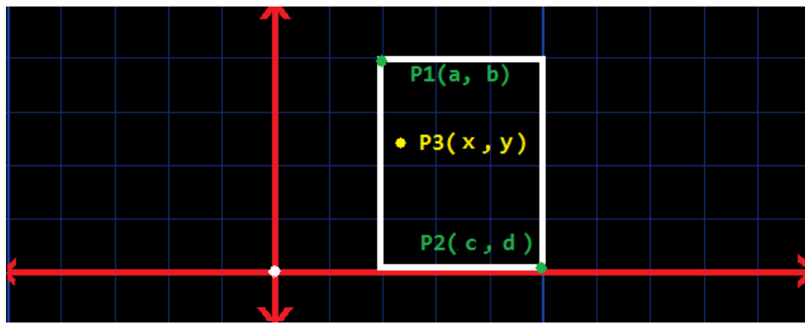


Figura 3.22: Colision de un punto $P3$ en un rectángulo formado por $P1$ y $P2$

En la Figura 3.23 se observa el diagrama de flujo para determinar si se produjo o no una colisión.

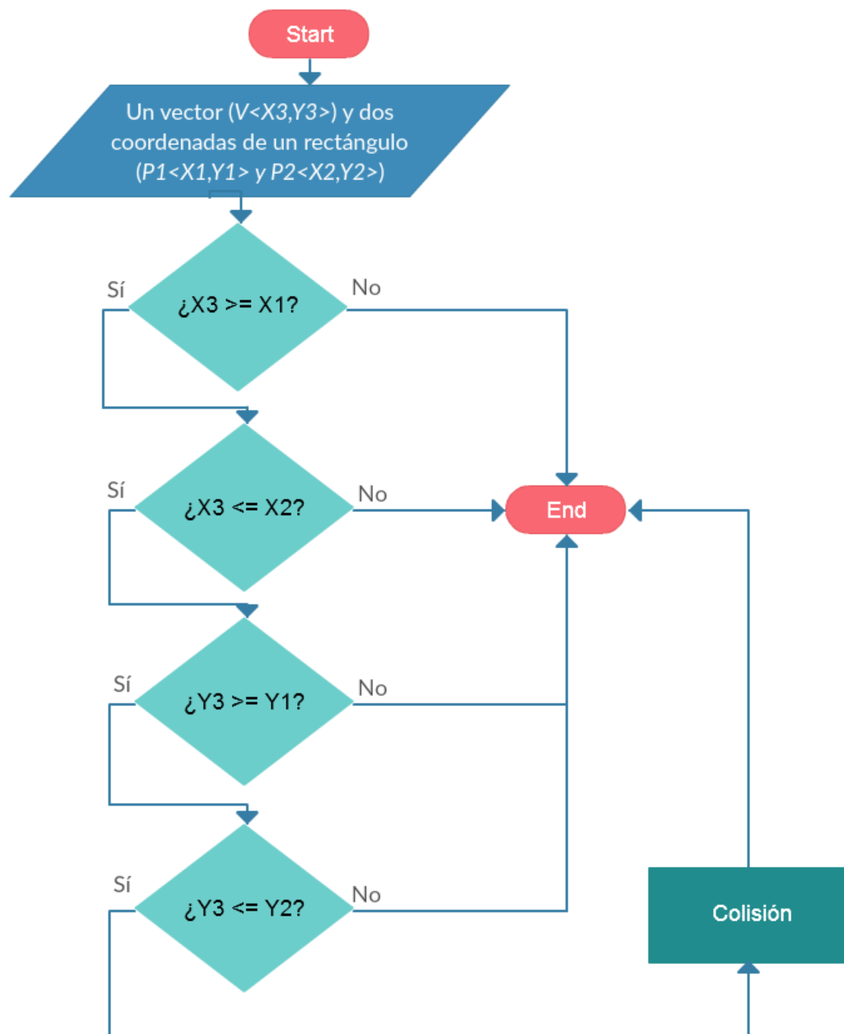


Figura 3.23: Verifica si hubo o no una colisión

### 3.4.3. Función *obtenerCuadranteDedo*()

Esta función recibe como parámetro una lista de dedos del usuario y un apuntador hacia un arreglo (ver Interfaz 3.9). Se encarga de calcular el cuadrante de un mapa cartesiano de 2D hacia el que apunta cada uno de los dedos del usuario contenido en la lista (no importa si los dedos están extendidos o no) y almacena en el arreglo la información obtenida. El formato utilizado para almacenar la información de los dedos es *número asociado al dedo y cuadrante al cual apunta*. La función devuelve '0' si no hay dedos en la lista de dedos del usuario y '1' si el conteo fue un éxito.

$$int \quad quadrantFingers(FingerList \quad _Fingers, int \quad _array[]) \tag{3.9}$$

En la Figura 3.24 se observa un ejemplo de una seña para la cual solo nos interesa conocer el cuadrante de cada dedo extendido (se deberá enviar a la función *obtenerCuadranteDedo*() una lista de dos dedos extendidos y un apuntador a un arreglo). En este ejemplo, los dedos del usuario están apuntando hacia el cuadrante 1. Como resultado, la función *obtenerCuadranteDedo*() almacena el número de cada dedo del usuario (0-pulgar, 1-indice, 2-medio, 3-anular y 4-meñique) y el cuadrante al que apuntada cada uno de ellos, quedando de la siguiente manera $_array = [1, 1, 2, 1]$, pues los dedos levantados son el índice (1) y el medio (2).

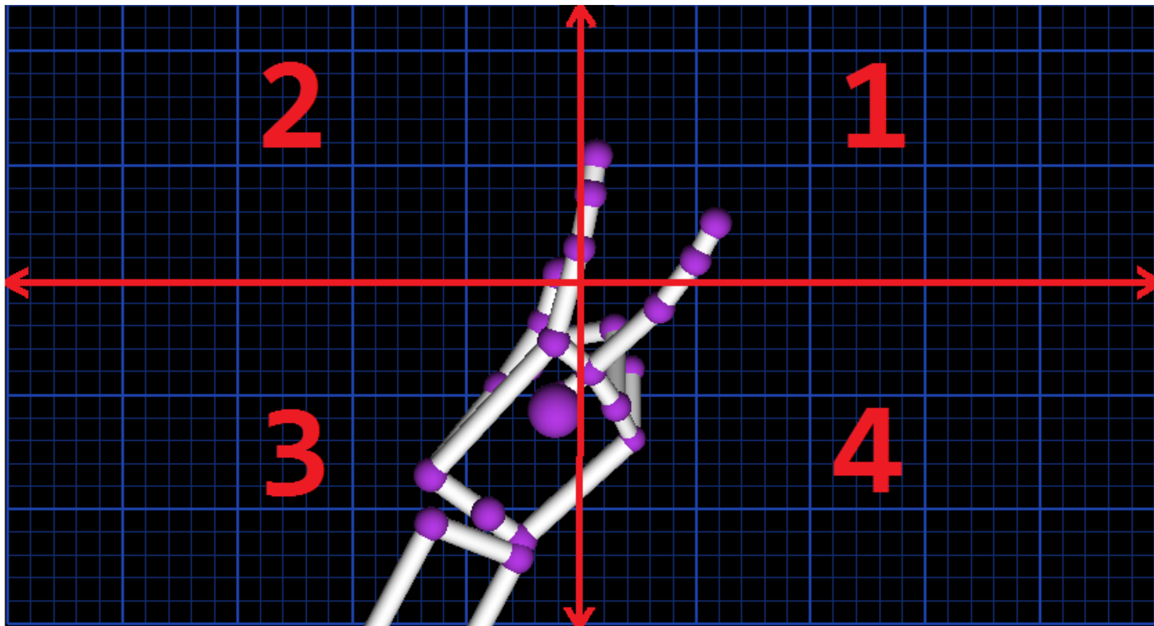


Figura 3.24: Dos dedos extendidos del usuario hacia el cuadrante uno

En la Figura 3.25 se muestra el diagrama de flujo que explica los pasos para determinar el cuadrante al que apunta cada dedo del usuario.

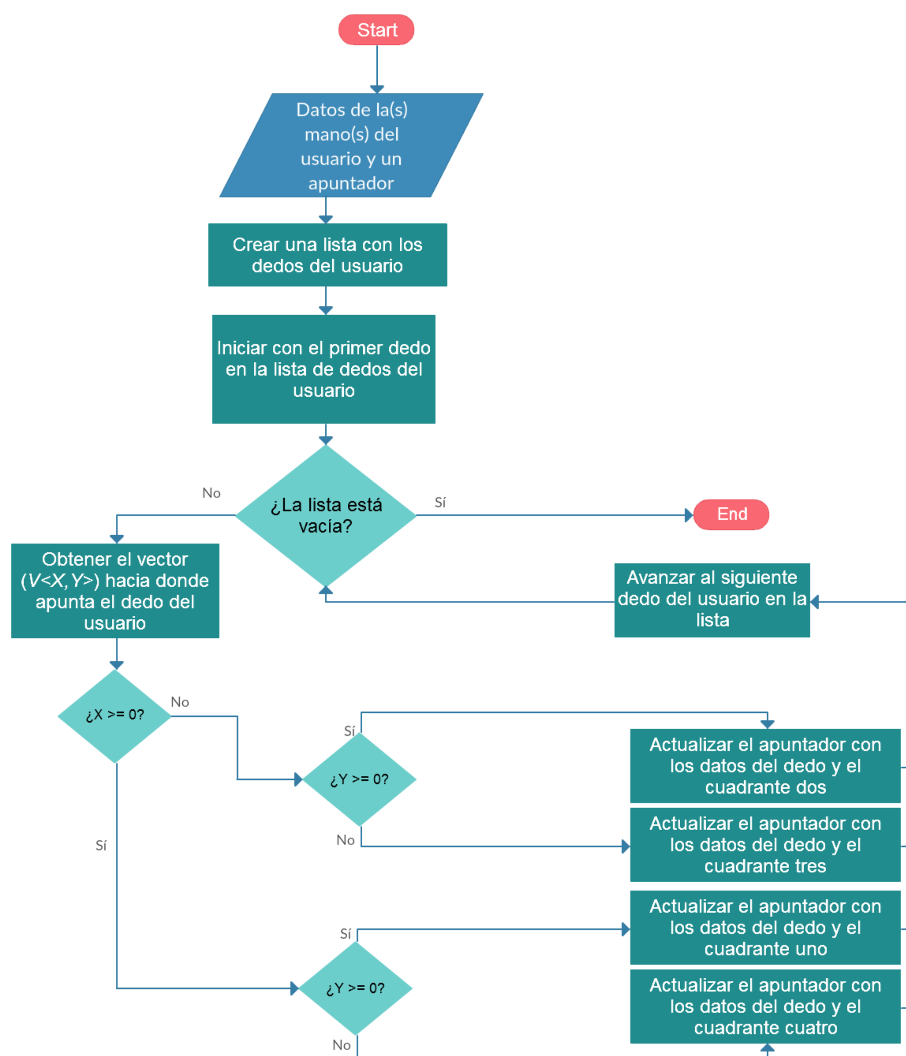


Figura 3.25: Obtención del cuadrante hacia donde apunta cada dedo del usuario

## 3.5. API de interpretación de señas

La tercera API del *toolkit* está compuesta por funciones que permiten interpretar señas realizadas por el usuario. Además, estas funciones permiten almacenar el código fuente para integrar dichas señas a cualquier aplicación que utilice el API de Leap Motion o guardar su información. En la Sección 3.5.1 se presenta la función que permite interpretar una seña realizada por el usuario y guardar la información de ésta en un documento de texto. En la Sección 3.5.2 se expone la función que permite crear código fuente en lenguaje C para validar una seña realizada por el usuario. Finalmente, en la Sección 3.5.3, se detalla la función que permite interpretar una seña realizada por el usuario, contemplando la dirección de cada dedo extendido y, posteriormente, crear el código fuente en lenguaje C de la seña interpretada.

### 3.5.1. Función *generarSeniaSinDir*()

Esta función recibe una lista de una o ambas manos del usuario, un entero (int) y un nombre (ver Interfaz 3.10). Se encarga de guardar en un documento de texto una función construida con código en C y nombrada de la siguiente forma $< senia_x() >$, donde $x$ es el nombre que se utilizó como parámetro para la función $generarSeniaSinDir()$. El entero que esta función recibe como parámetro puede ser: 1) cero, para crear la función de una seña incluyendo la verificación del tipo de mano (izquierda o derecha) del usuario que realiza la seña o 2) uno, para crear la función de una seña que no incluya la verificación del tipo de mano con la que el usuario realiza la seña. Si se decide utilizar ambas manos para realizar una seña, entonces el número que recibe $generarSeniaSinDir()$ no es utilizado y esta función guardará el código de dicha seña.

$$bool \quad generarSeniaSinDir(HandList \quad _hands, int \quad _aux, char \quad *_name) \tag{3.10}$$

La información que se necesita para construir una seña es la siguiente: número total de manos del usuario en la lista, identificación de la mano izquierda o derecha del usuario (solo si el número que recibe $generarSeniaSinDir()$ es 0 o si la seña es creada con dos manos) y la lista de los dedos extendidos del usuario con los números que Leap Motion asigna a cada dedo (0-pulgar, 1-indice, 2-medio, 3-anular y 4-meñique).

La función construida es la abstracción de una seña y recibe como parámetro una lista de una o ambas manos del usuario, la cual contiene los elementos necesarios, anteriormente explicados, de la seña realizada por el usuario (ver Interfaz 3.11). El código creado en lenguaje C puede ser usado inmediatamente en cualquier aplicación que trabaje con Leap Motion. La función creada tiene la única tarea de validar si la lista que recibe contiene la seña deseada. La función $generarSeniaSinDir()$ devuelve $false$ si hubo algún error al momento de crear o guardar el código en lenguaje C en el documento de texto y regresa $true$ si no hubo errores.

$$bool \quad sign_x(HandList \quad _hands) \tag{3.11}$$

En la Figura 3.26 se muestra una seña efectuada por el usuario, la cual se realizó con dos dedos extendidos. La función $generarSeniaSinDir()$ genera el código en lenguaje C correspondiente a la seña realizada.

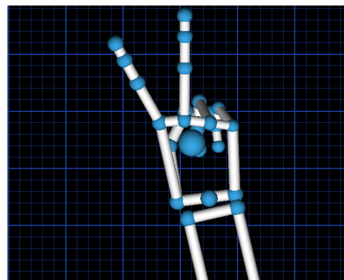

Figura 3.26: Seña realizada con dos dedos extendidos del usuario

En la Figura 3.27 se aprecia el flujo del funcionamiento de *generarSeniaSinDir*() para interpretar una seña realizada por el usuario y posteriormente crear el código en lenguaje C de dicha seña.

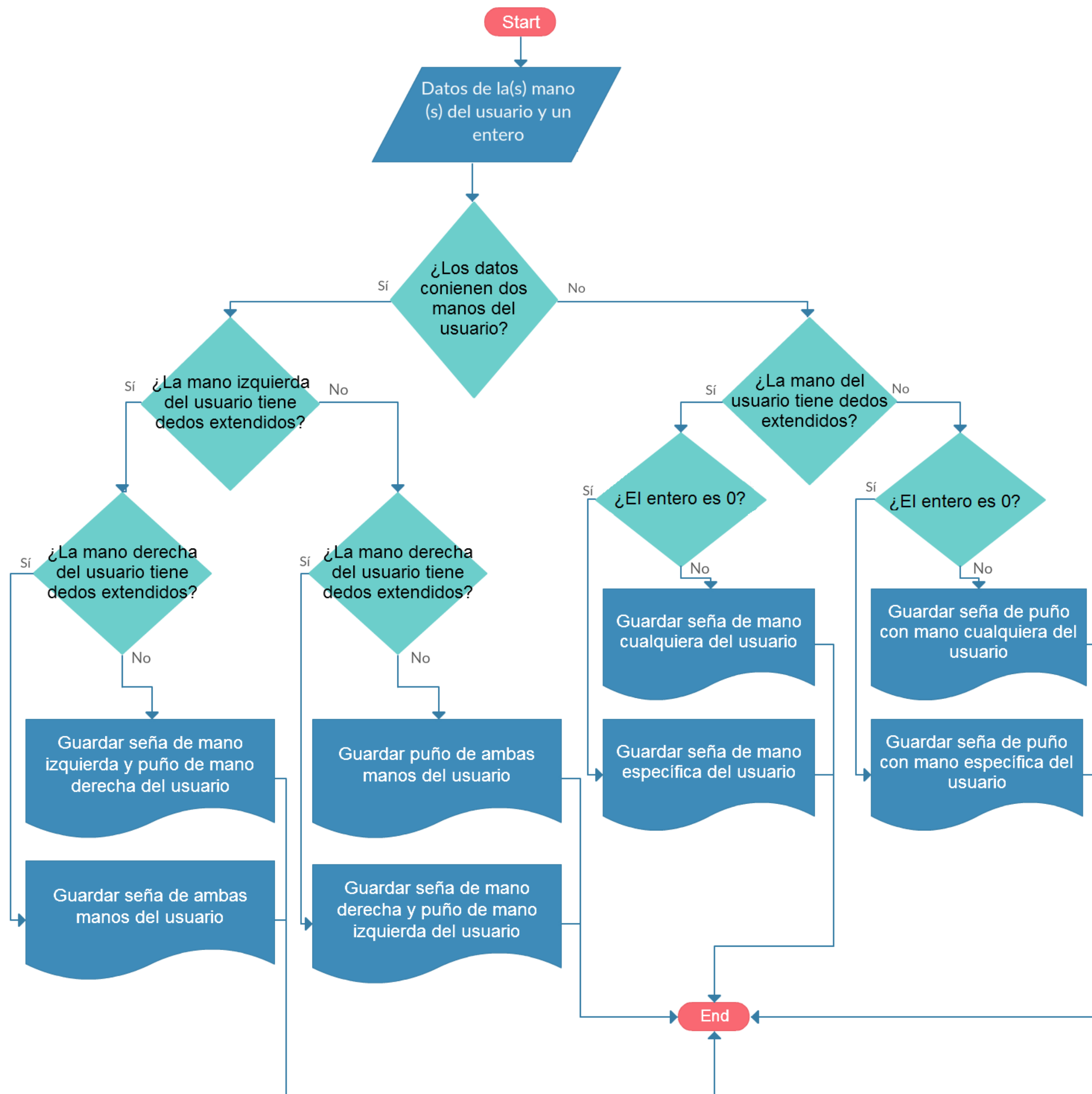


Figura 3.27: Almacenamiento en un archivo del código fuente en lenguaje C de una seña realizada por el usuario

En la Figura 3.28, se muestra el diagrama de flujo para validar la seña realizada, i.e, determinar si la seña realizada por el usuario coincide con la seña que se interpretó y guardó en código de lenguaje C.

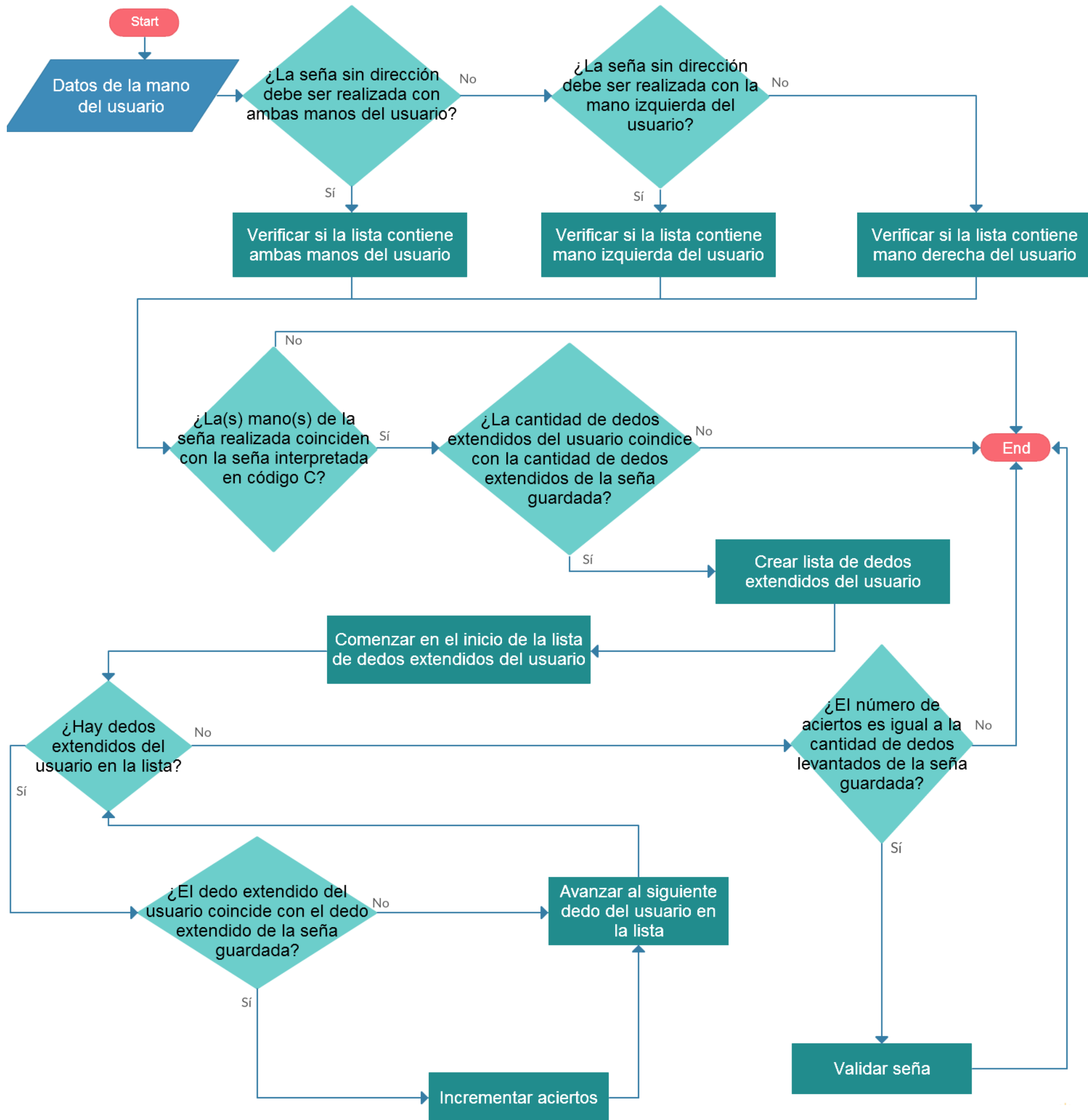


Figura 3.28: Validación de una seña realizada por el usuario

### 3.5.2. Función *generarInfoSenia*()

Esta función recibe una lista de una o ambas manos del usuario y un nombre (ver Interfaz 3.12). Guarda en un archivo el nombre real de los dedos extendidos (i.e., pulgar, indice, medio, anular y meñique), el vector de dirección de cada dedo extendido del usuario, la lista de los dedos extendidos del usuario con los números que Leap Motion asigna a cada dedo (0-pulgar, 1-indice, 2-medio, 3-anular y 4-meñique). Devuelve $false$ si hubo algún error al momento de crear o guardar los datos en el documento de texto y regresa $true$ si no hubo errores.

$$bool \quad generarInfoSenia(HandList \quad _hands, char * \quad _name) \tag{3.12}$$

En la Figura 3.29 se observa el flujo para obtener la información de una seña realizada por el usuario.

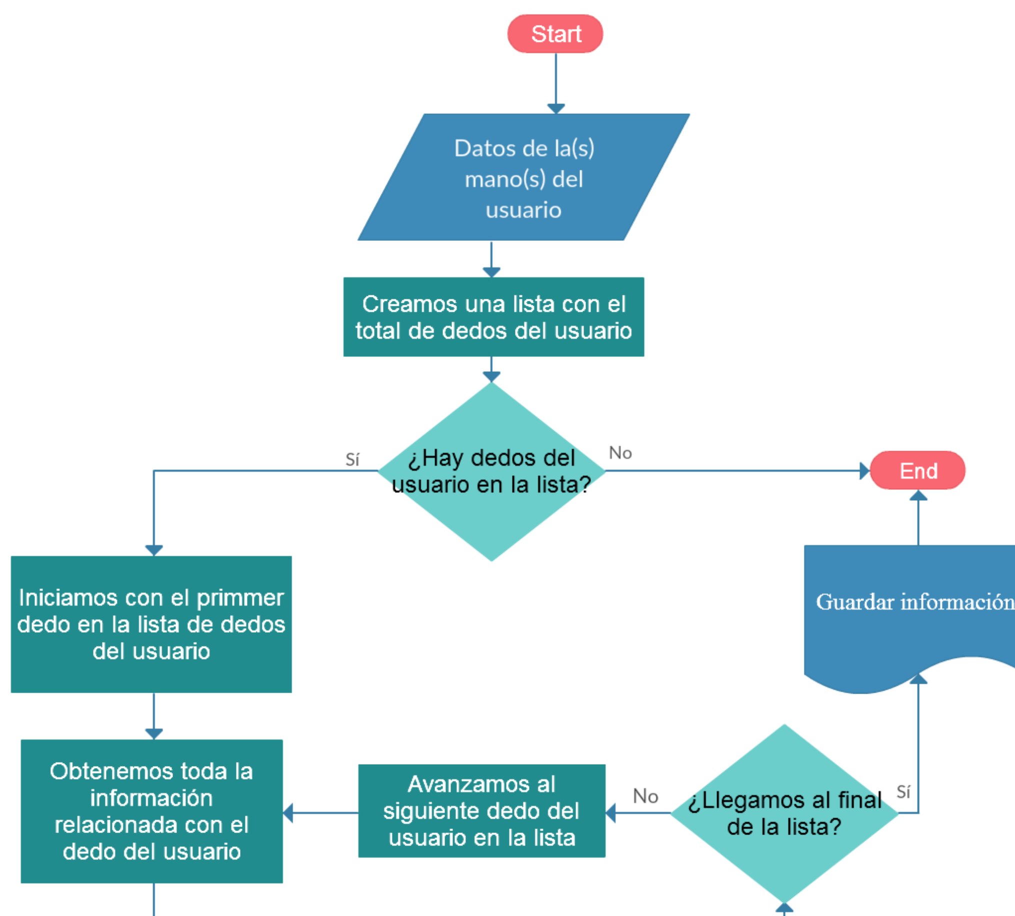


Figura 3.29: Almacenamiento en un archivo de la información de una seña realizada por el usuario

### 3.5.3. Función *generarSeniaConDir*()

Esta función realiza el mismo procedimiento que *generarSeniaConDir*() (ver Sección 3.5.1) con dos diferencias fundamentales. La primera es que una seña con dirección agrega la información del cuadrante al que apunta cada uno de los dedos extendidos del usuario, por lo tanto no existen puños, pues un puño apuntando hacia cualquier cuadrante de un mapa cartesiano siempre será un puño, por tanto la seña siempre será la misma sin importar la dirección. La segunda diferencia es que se debe guardar la información de cada dedo extendido del usuario en un arreglo (como en *generarSeniaSinDir*()) junto con el cuadrante hacia el cuál apunta. Para esta última tarea, la función *generarSeniaConDir*(), hace uso de la función de procesamiento *obtenerCuadranteDedo*() (ver Sección 3.4.3) para obtener los cuadrantes hacia los que apunta cada dedo extendido.

La función *generarSeniaConDir*() recibe una lista de una o ambas manos del usuario, un número y un nombre (ver Interfaz 3.13). La función devuelve *false* si hubo algún error al momento de crear o guardar los datos en el documento de texto y regresa *true* si no hubo errores. El número que recibe como parámetro es utilizado para determinar si la seña con dirección es con mano específica (‘0’) o con una mano cualquiera del usuario (‘1’). Si se utilizan ambas manos, el número no es tomado en cuenta.

La función creada por *generarSeniaConDir*() recibe una lista de una o ambas manos del usuario y es nombrada $< senia_x() >$, donde $x$ es el nombre que recibió *generarSeniaConDir*() como parámetro de entrada (ver Interfaz 3.11).

$$bool \quad generarSeniaConDir(HandList \quad _hands, int \quad _aux, char* \quad _name) \quad (3.13)$$

Como ejemplo se utiliza la seña realizada en la Figura 3.30, la cual será validada si cumple con los requisitos de los cuadrantes y los dedos extendidos. En la figura, el pulgar apunta al cuadrante dos y el resto apunta al cuadrante uno, sin importar la mano que se use (en este caso, la seña se realiza con la mano derecha, pero se puede voltear la mano izquierda y se validaría solo si la seña fue creada sin especificar con qué mano se debe validar).

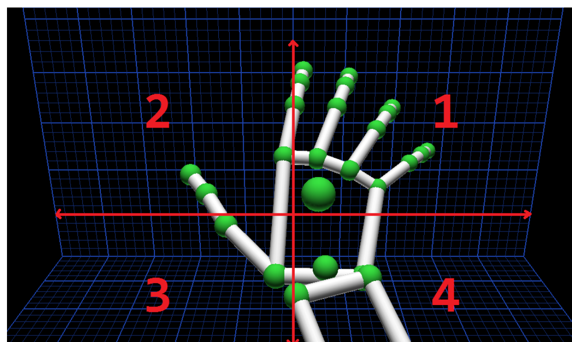


Figura 3.30: Seña realizada con una mano y los cinco dedos extendidos

En la Figura 3.31 se aprecia el funcionamiento de *generarSeniaConDir*() para crear el código en lenguaje C de una seña realizada por el usuario.

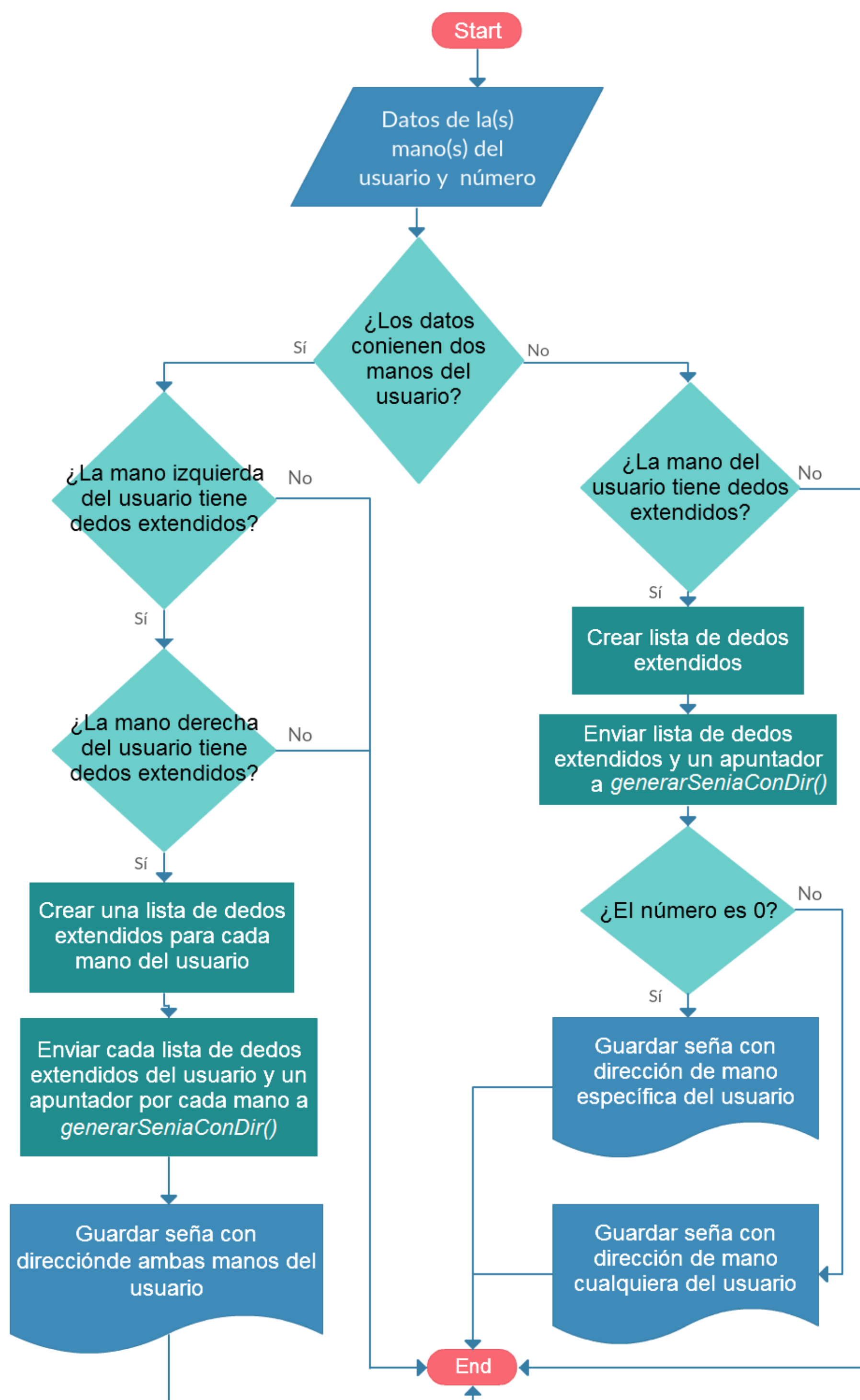


Figura 3.31: Almacenamiento en un archivo del código fuente en lenguaje C de una seña, empleando el cuadrante de cada dedo extendido del usuario

En la Figura 3.32, se observa el diagrama de flujo para validar la seña con dirección, i.e, determinar si la seña realizada por el usuario coincide con la seña que se interpretó y guardó en código de lenguaje C.

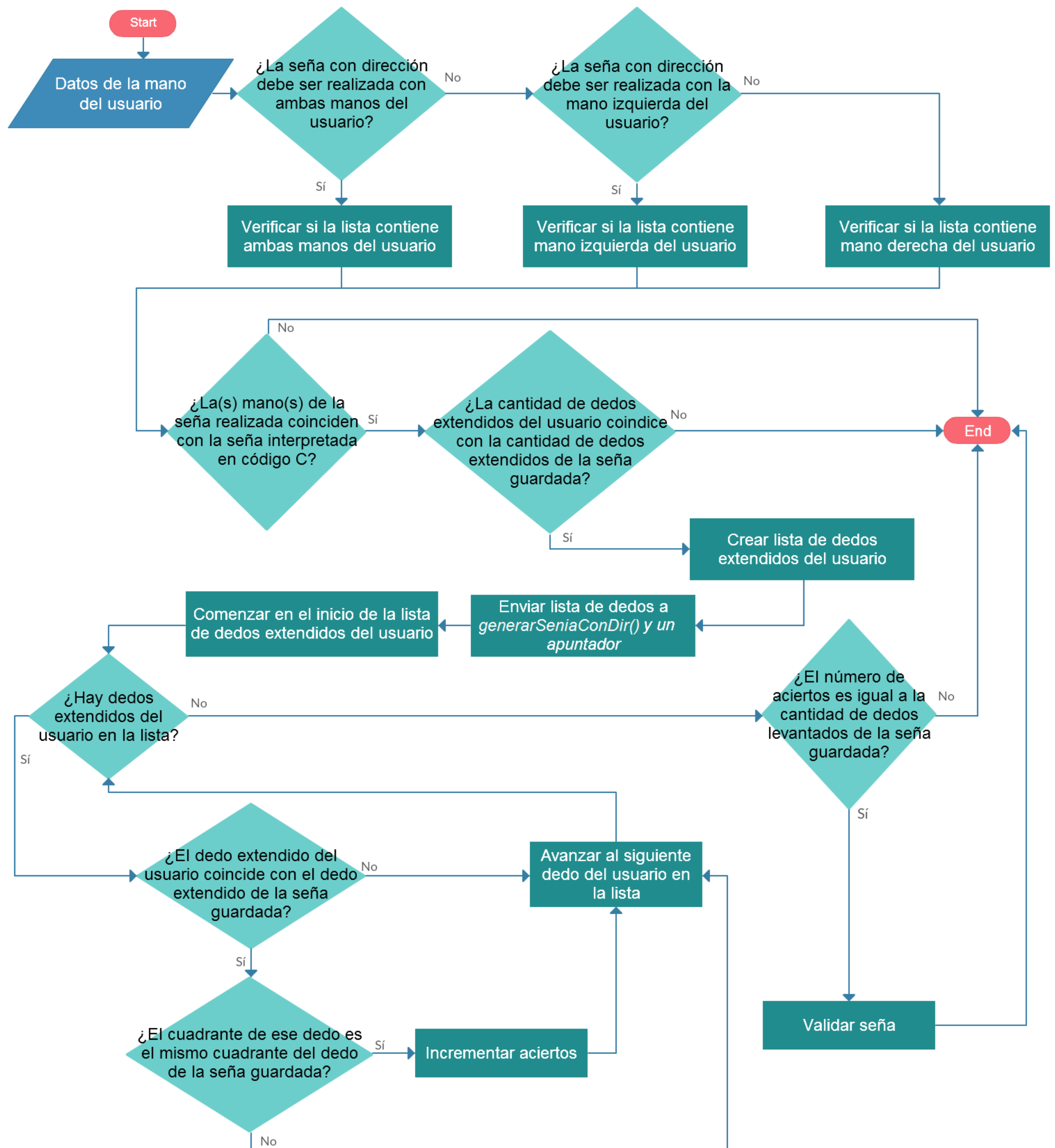


Figura 3.32: Validación de una seña con dirección realizada por el usuario

# Capítulo 4

# Implementación del *toolkit*

Este capítulo presenta la implementación de cada una de las APIs del *toolkit* desarrollado. En la Sección 4.1, se justifica nuestra elección del lenguaje de programación para implementar el *toolkit* y las aplicaciones de prueba. Así mismo, también se dan algunos consejos a considerar cuando un programador se enfrenta a la extensa información que se encuentra en la documentación del dispositivo, sobre todo si es la primera vez que programa aplicaciones que interaccionen con Leap Motion. En la Sección 4.2 se presentan los algoritmos relacionados con la API de control de usuarios. Posteriormente, en la Sección 4.3 se explica cada uno de los algoritmos de la API de procesamiento de dedos y vectores. Finalmente, en la Sección 4.4, se exponen los algoritmos relacionados con la API de interpretación de señas.

## 4.1. Lenguaje de programación

El lenguaje utilizado para la creación del *toolkit* y de las aplicaciones resultantes es C, por lo tanto el paradigma elegido para este trabajo de tesis es el de programación estructurada. Aunque Leap Motion utiliza el paradigma de programación orientado a objetos, se eligió la programación estructurada para alcanzar, de manera más rápida, un prototipo funcional.

Uno de los lenguajes con más documentación en el API de Leap Motion[1] es C++, el cual permite utilizar tanto el paradigma de programación orientada a objetos como el paradigma estructurado de C. El lenguaje C++ es utilizado por la API de Leap Motion, debido a que es un lenguaje estándar orientado a objetos con una potente capacidad de portabilidad y facilidad de comunicarse directamente a bajo nivel con el hardware [15].

Para la parte visual de las aplicaciones colaborativas se utilizó Allegro[2], que es una

[1] https://developer.leapmotion.com/documentation/cpp/index.html
[2] http://www.allegro.cc/

biblioteca para videojuegos escrita en lenguaje C, pero extendida a C++ y otros lenguajes de programación actuales. Contiene funciones que permiten añadir imágenes y audio, controlar rutinas de bajo nivel en videojuegos (e.g. temporizadores, teclado, ratón, audio), dibujar formas y calcular distancias.

Uno de los primeros problemas a enfrentarse con Leap Motion es el de desarrollar aplicaciones gráficas en lenguajes como C/C++, pues se debe tomar la decisión entre hacer una aplicación de consola o una aplicación gráfica. Si se decide por una aplicación de consola no habrá el problema de tener que experimentar con los diferentes *toolkits* de desarrollo gráfico que existen, tales como CINDER, OpenFrameworks, OpenGL y Allegro.

Para aquellos usuarios inexpertos en la programación orientada a objetos y en la programación de aplicaciones gráficas, se puede utilizar un mecanismo básico para comenzar, el cual consiste en el desarrollo de una aplicación en tres pasos:

1. Desarrollar una aplicación gráfica con cualquier *toolkit* gráfico, que funcione con teclado y/o ratón como dispositivos de entrada de la aplicación.
2. Desarrollar una aplicación de consola para Leap Motion que simule los dispositivos de entrada de la aplicación (teclado/ratón) pero con movimientos de las manos, usando impresiones para ayudarse, tales como arriba, abajo, derecha e izquierda.
3. Integrar los dispositivos de entrada de la aplicación de consola (creada para Leap Motion) en la aplicación gráfica.

Siguiendo estos pasos, se vuelve más sencillo entender cómo se integran las funciones o clases de Leap Motion dentro de un código de aplicación cualquiera.

## 4.2. Implementación de la API de control de usuarios

En esta sección de presentan los algoritmos de la API de control de usuarios, la cual contiene las siguientes funciones:

- $asignarId()$
- $obtenerId()$
- $borrarId()$
- $obtenerLista()$
- $contadorId()$
- $verificarLista()$

Estas funciones están explicadas de la Sección 4.2.1 a la Sección 4.2.6.

### 4.2.1. Función *asignarId*()

La función $asinarId()$ es utilizada por el programador para que un nuevo usuario sea agregado a la lista de usuarios conectados.

El primer paso del Algoritmo 1 para agregar un usuario a la lista, es crear el nodo (*_nodo*) donde se almacenarán los datos del usuario, como la dirección IP de su dispositivo y el identificador que le será asignado por la función $asignarId()$.

Posteriormente, se verifica si *_nodo* fue creado de manera exitosa; en caso de que no haya sido creado (i.e. no hay suficiente memoria secundaria) la función no puede continuar y regresa '-1' (ver líneas 1 y 2).

Si *_nodo* fue creado con éxito, entonces se guardan los datos del usuario, los cuales son recibidos como parámetros por la función y su apuntador *_posterior* se iguala a $NULL$ (ver líneas de la 3 a la 5).

El siguiente paso es acomodar el nuevo nodo en la lista. Para ello se necesita saber si *_nodo* es el primero en la lista o no, usando la función $verificarLista()$. Si esta función regresa **true**, entonces *_nodo* es el primero en la lista, así que su apuntador *_anterior* se actualiza a $NULL$ y los apuntadores auxiliares, que marcan el inicio y el final de la lista (*_inicio* y $_final$ respectivamente) se igualan a *_nodo* (ver líneas de la 6 a la 9).

En caso de que *_nodo* no sea el primero en la lista, su apuntador *_anterior* se actualiza al nodo apuntado por $_final$. Después, el apuntador *_posterior* del nodo, que era el final de la lista, se actualiza a *_nodo* y el apuntador auxiliar $_final$ se iguala a *_nodo* (ver líneas de la 10 a la 13).

Finalmente, se incrementa el contador (*_cont*) de usuarios conectados y se regresa el identificador asociado al nuevo usuario (ver líneas 14 y 15).

**Algorithm 1** Asignar identificador a un usuario nuevo en la lista de usuarios conectados

**Require:** Dirección IP de la máquina del nuevo usuario ($_dir$).
**Ensure:** Si hay memoria secundaria para crear el nodo, devuelve el identificador asociado al nuevo usuario o -1 en caso contrario.

1: **if** $_nodo = NULL$ **then**
2:    **return** -1
3: $_nodo \rightarrow _IP = _dir$
4: $_nodo \rightarrow _id = ++_ID$
5: $_nodo \rightarrow _posterior = NULL$
6: **if** $verificarLista()$ **then**
7:    $_nodo \rightarrow _anterior = NULL$
8:    $_inicio = _nodo$
9:    $_final = _nodo$
10: **else**
11:    $_nodo \rightarrow _anterior = _final$
12:    $_final \rightarrow _posterior = _nodo$
13:    $_final = _nodo$
14: $_cont++$
15: **return** $_ID$

### 4.2.2. Función *obtenerId*()

La función $obtenerId()$ se encarga de buscar en la lista de usuarios conectados una coincidencia entre la dirección IP asociada a un usuario en la lista y la dirección IP que recibe como parámetro.

El primer paso del Algoritmo 2, es crear un apuntador auxiliar ($_auxiliar$), el cual es apuntado a $_inicio$, i.e., se comienza desde el inicio de la lista para recorrer la lista de usuarios conectados (ver línea 1).

Posteriormente, se realiza la búsqueda comparando la dirección IP que se recibió como parámetro y la dirección IP almacenada en cada nodo, siempre que el apuntador $_auxiliar$ sea diferente del apuntador $_final$, i.e., mientras no se alcance el final de la lista de usuarios conectados (ver líneas de la 2 a la 6).

Si la dirección IP buscada coincide con la dirección IP del nodo visitado, se regresa el identificador asociado al usuario encontrado (ver líneas 3 y 4). En caso contrario, el apuntador $_auxiliar$ se posiciona en el siguiente nodo, i.e., avanzamos en la lista de usuarios conectados (ver línea 6). Si el apuntador $_auxiliar$ es igual a $_final$ y no encontró coincidencias, entonces regresa '0' (ver líneas 7 y 8).

**Algorithm 2** Buscar el identificador asociado a una dirección IP

**Require:** Dirección IP de la máquina del usuario que se busca ($_dir$).

**Ensure:** Si encuentra la coincidencia devuelve el identificador asociado al nodo que contiene la dirección IP y '0' si no la encuentra.

1: $_auxiliar = _inicio$
2: **while** $_auxiliar$ != $NULL$ **do**
3:    **if** $_auxiliar \rightarrow _IP = _dir$ **then**
4:       **return** $_auxiliar \rightarrow _id$;
5:    **else**
6:       $_auxiliar = _auxiliar \rightarrow _posterior$;
7: **if** $_auxiliar == NULL$ **then**
8:    **return** 0

### 4.2.3. Función *borrarId*()

Esta función permite al programador de la aplicación borrar a un usuario de la lista de usuarios conectados, mediante el identificador de dicho usuario.

El primer paso del Algoritmo 3 es crear un apuntador auxiliar ($_auxiliar$), el cual es igualado al apuntador $_inicio$, i.e., se comienza desde el inicio de la lista para recorrer la lista de usuarios conectados (ver línea 1).

Posteriormente, se realiza la búsqueda comparando el identificador que recibe como parámetro con el identificador almacenado en cada nodo, siempre que el apuntador $_auxiliar$ no alcance el final de la lista de usuarios conectados (ver líneas de la 2 a la 21).

Si el identificador buscado coincide con el identificador del nodo visitado, pueden ocurrir tres situaciones:

1. que el apuntador $_auxiliar$ sea igual al inicio de la lista de usuarios conectados, entonces se verifica si el apuntador $_posterior$ del nodo que apunta $_auxiliar$ es igual a $NULL$, i.e. el nodo a eliminar es el único nodo. Si lo es, entonces los apuntadores auxiliares $_inicio$ y $_final$ se igualan a $NULL$ (i.e., la lista queda vacía); en caso contrario, el apuntador $_anterior$ del nodo siguiente al nodo que apunta $_auxiliar$ se iguala a $NULL$ (i.e., el nodo siguiente se vuelve el primer nodo) y el apuntador $_inicio$ se iguala al nodo siguiente del nodo que apunta $_auxiliar$ (ver líneas de la 3 a la 10).

2. que el nodo que apunta $_auxiliar$ sea igual al final de la lista de usuarios conectados, entonces el apuntador $_posterior$ del nodo anterior al que apunta $_auxiliar$ se iguala a $NULL$ (i.e. el penúltimo nodo se convierte en el último nodo); posteriormente, $_final$ apunta al nodo anterior al que apunta $_auxiliar$ (ver lineas de la 11 a la 13).

3. que el nodo *_auxiliar* no sea ni el inicio ni el final de la lista, entonces el apuntador *_posterior* del nodo anterior al nodo *_auxiliar* se iguala al nodo siguiente del nodo apuntado por *_auxiliar*; posteriormente, el apuntador *_anterior* del nodo siguiente al nodo apuntado por *_auxiliar*, se iguala al nodo anterior del nodo que es apuntado por *_auxiliar* (ver líneas de la 14 a la 16).

Una vez identificada la posición del nodo apuntado por *_auxiliar*, éste se elimina. Después, se disminuye el número de usuarios conectados en la lista y la función regresa '1' (ver líneas de la 17 a la 19).

Si no se encuentra un identificador que coincida con el identificador que se quiere eliminar y no se ha alcanzado el final de la lista, entonces el nodo apuntado por *_auxiliar* apunta al nodo que está asociado al apuntador *_posterior* del nodo apuntado por *_auxiliar*, i.e., avanzamos en la lista de usuarios conectados (ver líneas 20 y 21).

Si se alcanza el final de la lista sin encontrar coincidencias, la función regresa '0' (ver línea 22).

**Algorithm 3** Borrar a un usuario de la lista de usuarios conectados)

**Require:** Identificador asociado al usuario a borrar (*_bid*).

**Ensure:** Devuelve '1' si encuentra la coincidencia entre el identificador contenido en el nodo y el identificador que recibe como parámetro, en caso de que no encuentre coincidencia devuelve '0'.

```
1: _auxiliar = _inicio
2: while _auxiliar != NULL do
3:    if _auxiliar → _id == _bid then
4:       if _auxiliar == _inicio then
5:          if _inicio → _posterior == NULL then
6:             _inicio = NULL
7:             _final = NULL
8:          else
9:             _inicio → _anterior = NULL;
10:            _inicio = _inicio → _posterior
11:      else if _auxiliar == _final then
12:         _final → _posterior = NULL
13:         _final = _auxiliar → _anterior
14:      else
15:         (_auxiliar → _anterior) → _posterior = _auxiliar → _posterior
16:         (_auxiliar → _posterior) → _anterior = _auxiliar → _anterior
17:      _cont − −
18:      return 1
19:   else
20:      _auxiliar = _auxiliar → _posterior
21: return 0
```

### 4.2.4. Función *obtenerLista*()

Esta función se encarga de obtener el contenido de la lista de usuarios conectados para que el programador pueda visualizar su contenido en pantalla.

El primer paso del Algoritmo 4 es crear un apuntador auxiliar (*_auxiliar*), el cual es apuntado a *_inicio*, i.e., se comienza desde el inicio de la lista, para recorrer la lista de usuarios conectados (ver línea 2).

Antes de recorrer la lista de usuarios conectados, se verifica que la lista no esté vacía, usando la función *verificarLista*(). Si esta función regresa **true** entonces la lista está vacía y *obtenerLista*() regresa '0' (ver líneas 2 y 3). En caso contrario, la función *obtenerLista*() recorre la lista de usuarios conectados, desplegando la información de cada nodo, hasta que el nodo al que apunta *_auxiliar* alcance el final de la lista (ver líneas de la 4 a la 7).

Si se alcanzó el final de la lista, la función regresa '1' (ver línea 8).

**Algorithm 4** Desplegar la información de la lista de usuarios conectados.

**Require:** No recibe ningún parámetro
**Ensure:** Devuelve '0' si la lista de usuarios conectados está vacía y '1' si la lista contiene al menos un usuario.

1: *_auxiliar* = *_inicio*
2: **if** *verificarLista*() **then**
3:    **return** 0
4: **else**
5:    **while** *_auxiliar* != *NULL* **do**
6:       **print** *_auxiliar* → *_id*
7:       **print** *_auxiliar* → *_IP*
8:    **return** 1

### 4.2.5. Función *contadorId*()

La función es utilizada para devolver el contenido de la variable de tipo global *_cont*, la cual lleva el registro del número de usuarios conectados.

Enseguida se presenta el Algoritmo 5, el cual regresa el número de usuarios conectados (ver línea 1).

**Algorithm 5** Conocer el número de usuarios conectados.

**Require:** No recibe ningún parámetro.
**Ensure:** Devuelve el número de usuarios conectados.

1: **return** *_cont*

### 4.2.6. Función *verificarLista*()

Esta función se usa para verificar si la lista de usuarios está vacía, i.e. la lista de usuarios conectados no contiene usuarios.

En el Algoritmo 6 se verifica si *_cont* es igual a '0'. Si lo es, la función regresa **true** (ver líneas 1 y 2) y, en caso contrario, regresa **false** (ver líneas 3 y 4).

**Algorithm 6** Verificar si lista de usuarios conectados se encuentra vacía.
**Require:** No recibe ningún parámetro.
**Ensure:** Devuelve **true** si la lista de usuarios conectados no contiene usuarios y **false** si la lista contiene al menos un usuario.

```
1: if _cont == 0 then
2:    return true
3: else
4:    return false
```

## 4.3. Implementación de la API de procesamiento de dedos y vectores

En esta sección se presentan las implementaciones de las funciones de procesamiento de dedos y vectores, las funciones son:

- *contarDedosExtendidos*()
- *calcularColision*()
- *obtenerCuadranteDedo*()

Los respectivos algoritmos de cada función y una breve explicación de su funcionamiento se encuentran distribuidos desde la Sección 4.2.1 a la Sección 4.2.3.

### 4.3.1. Función *contarDedosExtendidos*()

Esta función regresa el número total de dedos extendidos en una o ambas manos del usuario, utilizando tres métodos: 1) *fingers*() que obtiene una lista de dedos de una mano del usuario, 2) *extended*() para adquirir una lista de dedos extendidos a partir de una lista de dedos y 3) *count*() el cual cuenta los elementos de una lista cualquiera. Todos forman parte de la clase `FingerList`. Así como los métodos: 1) *begin*() para obtener la posición inicial en una lista y 2) *end*() para obtener la posición final de una lista, y forman parte de la clase `HandList` (ver Sección 2.2.1).

El primer paso del Algoritmo 7 es declarar la variable $_c$ que servirá de contador (ver línea 1). Después se recorre la lista de manos utilizando $begin()$ y $end()$ para obtener el objeto $_mano$ y posteriormente, adquirir el número total de dedos extendidos de dicho objeto para cada iteración (ver líneas 2 y 3). Finalmente, la función regresa el número de dedos extendidos de una o ambas manos del usuario (ver línea 4).

Los métodos que Leap Motion proporciona no permiten obtener de manera sencilla el número total de dedos extendidos, pues primero se crea una lista de ellos y, posteriormente, se cuentan los elementos de dicha lista para cada una de las manos del usuario. En cambio, *contarDedosExtendidos*() permite obtener el total de dedos extendidos para una o ambas manos del usuario, utilizando una lista de manos.

**Algorithm 7** Suma de dedos extendidos de una o ambas manos del usuario.

**Require:** Una lista de una o ambas manos del usuario ($_listaManos$).
**Ensure:** El total de dedos extendidos del usuario dentro de la lista de manos.
1: $_c = 0$.
2: **for all** $_mano$ en $_listaManos$ **do**
3: $_c$ += $_mano.fingers().extended().count()$
4: **return** $_c$

### 4.3.2. Función *calcularColision*()

Esta función permite calcular una colisión entre un punto (obtenido de un *vector*) y un rectángulo (formado por cuatro números $_a$, $_b$, $_c$ y $_d$) recibidos como parámetro.

En el Algoritmo 8 se muestra cómo calcular una colisión. El primer paso es verificar que la componente $x$ del *vector* sea mayor o igual que $_a$, pero menor que $_b$ (ver líneas 1 y 2). Después se verifica que la componente $y$ del *vector* sea mayor que $_c$, pero menor que $_d$ (ver líneas 3 y 4). De ser así, entonces la función $calcularColision()$ regresa **true** (ver línea 5). En caso contrario, regresa **false** (ver línea 6).

**Algorithm 8** Detectar la colisión de un punto dentro de un rectángulo

**Require:** Un vector ($_vector$) para formar un punto y cuatro enteros ($_a$, $_b$, $_c$, $_d$) para formar el rectángulo.
**Ensure:** Devuelve **true** si el punto se encuentra dentro del rectángulo y **false** si no se encuentra dentro.
1: **if** $_vector.x \geq _a$ **then**
2: **if** $_vector.x \leq _b$ **then**
3: **if** $_vector.y \geq _c$ **then**
4: **if** $_vector.y \leq _d$ **then**
5: **return true**
6: **return false**

### 4.3.3. Función *obtenerCuadranteDedo*()

Su propósito es calcular el cuadrante al que apunta cada dedo en la lista que recibe y almacenar en el arreglo apuntado por *_aux*, en dicho arreglo se almacena el número asignado por Leap Motion a cada dedo (0-pulgar, 1-indice, 2-medio, 3-anular y 4-meñique) y el cuadrante al que apunta dicho dedo (1, 2, 3, 4).

Esta función se utiliza métodos de las clases proporcionadas por la API de Leap Motion, como *direction*() de la clase `Vector`, que obtiene la dirección a la que apunta un dedo del usuario, así como el método *type*() de la clase `Finger`, que devuelve el número asignado por Leap Motion a cada dedo. Los métodos *begin*() y *end*() de la clase `FingerList` para obtener la posición inicial y final, respectivamente, en la lista de dedos (ver Sección 2.2.1).

El primer paso del Algoritmo 9 consiste en verificar si no hay dedos en la lista de dedos del usuario. Si es así, entonces la función *obtenerCuadranteDedo*(), regresa '0' (ver líneas 1 y 2).

En caso de que la lista contenga dedos (ver línea 3), se declara la variable *_j*, la cual servirá de índice para el arreglo apuntado por *_aux*. De esta forma, es posible guardar los datos cada dos posiciones en dicho arreglo (ver línea 4). Después se recorren uno a uno, utilizando los métodos *begin*() y *end*(). En cada iteración se obtiene el objeto *_dedo* (ver líneas de la 5 a la 18), el cual contiene el *vector _v* $(< x, y >)$ con la dirección a la que apunta y el número que Leap Motion asigna a cada dedo, este último se almacena en el arreglo apuntado por *_aux* (ver líneas 6 y 7).

Para obtener el cuadrante al que apunta cada dedo, existen cuatro casos:

1. si ambas componentes del *vector _v* son positivas, entonces se guarda el cuadrante 1 en el arreglo apuntado por *_aux* (ver líneas de la 8 a la 10).

2. si la componente $x$ del *vector_v* es positiva y la componente $y$ del *vector_v* es negativa, entonces se guarda el cuadrante 4 en el arreglo apuntado por *_aux* (ver líneas de la 11 a la 12).

3. si la componente $x$ del *vector_v* es negativa y la componente $y$ *vector_v* es positiva, entonces se guarda el cuadrante 2 en el arreglo apuntado por *_aux* (ver líneas de la 13 a la 15).

4. si ambas componentes del *vector_v* son negativas, se guarda el cuadrante 3 en el arreglo apuntado por *_aux* (ver líneas de la 16 a la 17).

El siguiente paso es aumentar en dos a la variable *_j* (ver línea 18) y la iteración termina hasta verificar el último dedo en la lista.

Finalmente la función regresa '1' si toda la operación fue exitosa (ver línea 19).

**Algorithm 9** Calcular el cuadrante al que apunta cada uno de los dedos del usuario

**Require:** Una lista de dedos del usuario ($_listaDedos$) y un apuntador a un arreglo ($_aux$).

**Ensure:** Si hay dedos en la lista, almacena en el arreglo apuntado, el número correspondiente al dedo junto con el cuadrante al que apunta, si no se encuentran dedos regresa '0'.

```
1: if no hay dedos en _listaDedos then
2:   return 0
3: else
4:   _j = 0
5:   for all _dedo en _listaDedos do
6:     _v = _dedo.direction()
7:     _aux[_j] = _tipo
8:     if _v.x ≥ 0 then
9:       if _v.y ≥ 0 then
10:        _aux[_j + 1] = 1
11:      else
12:        _aux[_j + 1] = 4
13:    if _v.x = 0 then
14:      if _v.y ≥ 0 then
15:        _aux[_j + 1] = 2
16:      else
17:        _aux[_j + 1] = 3
18:    _j += 2
19: return 1
```

## 4.4. Implementación de la API de interpretación de señas

En esta sección se presentan la API de interpretación de señas, la cual contiene las siguientes funciones:

- $generarSeniaSinDir()$
- $generarInfoSenia()$
- $generarSeniaConDir()$

Cada uno de los algoritmos que explican el funcionamiento de cada componente de la API de señas, se presentan de la Sección 4.4.1 a la Sección 4.4.3.

### 4.4.1. Función *generarSeniaSinDir*()

En el Algoritmo 10 se muestran los pasos para interpretar y generar el código en lenguaje C de una seña sin dirección realizada por el usuario, cuyo primer paso es determinar el número de manos que se recibe en la lista de manos del usuario (ver línea 1). Para ello se utiliza el método *count*() de la clase `HandList`. Si se recibe una mano, entonces se extrae una lista de dedos extendidos, se cuentan sus elementos y el total se almacena en la variable *_dedos*. Los métodos utilizados son: *begin*() de la clase `HandList`, así como *fingers*(), *extended*() y *count*() de la clase `FingerList` (ver Sección 4.3.1). Después se verifica si:

1. *_dedos* es igual a '0', se comprueba si la seña debe hacerse con una mano específica del usuario. Si es así, se almacena el código en lenguaje C del puño realizado con la mano específica del usuario. En caso de que no se necesite interpretar la seña con una mano específica, se guarda el código en lenguaje C del puño realizado con una mano cualquiera (ver líneas de la 2 a la 6).

2. *_dedos* es diferente de '0', se verifica si la seña debe ser realizada con una mano específica del usuario. Si es así, se almacena el código en lenguaje C de la seña realizada con la mano específica del usuario. En caso de que no se necesite interpretar la seña con una mano específica, se guarda el código en lenguaje C de la seña realizada con una mano cualquiera (ver líneas de la 7 a la 11).

En el documento de texto se almacena "int _dedosB = $a$", donde $a$ es el valor de la variable *_dedos*. Si la seña se realizó con mano específica, se almacena el texto "char _manoB = $'Z'$ ", donde $'Z'$ puede ser 'L' para la mano izquierda o 'R' para la mano derecha. Así mismo, se guarda el texto "int _dedos_ext[$b$] = $\{c\}$" donde $b$ es el valor de la variable *_dedos* y $c$ es un conjunto de números formado por el tipo de dedo que se obtiene con el método *type*() de la clase `Finger`.

Si se reciben ambas manos, se obtiene la mano izquierda y derecha, utilizando los métodos *leftmost*() y *rightmost*(), respectivamente, de la clase `HandList`, y se cuenta el número de dedos extendidos de cada mano, almacenando el total en las variables *_dedosL* y *_dedosR*, para la mano izquierda y derecha, respectivamente. Posteriormente, se comprueba si ocurre uno de los siguientes casos: (ver línea 12):

1. si las variables *_dedosL* y *_dedosR*, son igual a '0', i.e., si ambas manos son puños (ver línea 13).

2. si la variable *_dedosL* es igual a '0' y la variable *_dedosR* sea diferente de '0', i.e., si la mano izquierda es un puño (ver línea 15).

3. si la variable *_dedosL* es diferente de '0' y la variable *_dedosR* sea igual a '0', i.e., si la mano derecha es un puño (ver línea 17).

4. si las variables *_dedosL* y *_dedosR*, son diferentes de '0', i.e., si ninguna mano es un puño (ver línea 19).

Si alguno de los casos anteriores ocurre, entonces se guarda en un documento de texto el código para identificar la seña del caso que resultó ser verdadero (ver líneas 14, 16, 18 y 20). Además, se almacena, en el documento de texto, "int _dedosBL = a" e "int _dedosBR = b", donde *a* y *b* contienen el valor de las variables *_dedosL* y *_dedosR*, respectivamente. Posteriormente, se almacena el texto para identificar cada mano, "Hand _manoBL = _listaManos.leftmost()" y "Hand _manoBR = _listaManos.rightmost()", para obtener la mano izquierda (_manoBL) y derecha (_manoBR) durante la validación de una seña. También se almacena el texto "int _dedos_extBL[e] = $\{f\}$" y "int _dedos_extBR[g] = $\{h\}$", donde *e* y *g* son los valores de las variables *_dedosL* y *_dedosR*, respectivamente y *f* y *h* son conjuntos de números formados por el tipo de cada dedo que se obtiene con el método *type*().

La función *generarSeniaSinDir*() regresa **true** en caso que termine exitosamente (ver línea 21). Si existe un error al guardar el archivo, la función regresa **false** (ver líneas 4, 6, 9, 11, 14, 16, 18 y 20).

**Algorithm 10** Almacenar el código en C de una seña realizada por el usuario

**Require:** La lista de una o ambas manos del usuario (*_listaManos*), un número para identificar si se requiere una mano específica o una mano cualquiera del usuario (*_aux*) y un nombre para el documento y la función generada (*_nombre*).

**Ensure:** Devuelve **true** si se guarda el código de la seña interpretada en un documento de texto; en caso de existir un error al guardar, devuelve **false**

1: **if** *_listaManos.count*() == 1 **then**
2:   **if** *_dedos* == 0 **then**
3:     **if** *_aux* == 0 **then**
4:       Guardar en el documento *_nombre* el código para identificar la seña.
5:     **else**
6:       Guardar en el documento *_nombre* el código para identificar la seña.
7:   **else**
8:     **if** == 1 **then**
9:       Guardar en el documento *_nombre* el código para identificar la seña.
10:     **else**
11:       Guardar en el documento *_nombre* el código para identificar la seña.
12: **else**
13:   **if** *_dedosL* == 0 && *_dedosR* == 0 **then**
14:     Guardar en el documento *_nombre* el código para identificar la seña.
15:   **else if** *_dedosL* == 0 && *_dedosR* != 0 **then**
16:     Guardar en el documento *_nombre* el código para identificar la seña.
17:   **else if** *_dedosL* != 0 && *_dedosR* == 0 **then**
18:     Guardar en el documento *_nombre* el código para identificar la seña.
19:   **else if** *_dedosL* != 0 && *_dedosR* != 0 **then**
20:     Guardar en el documento *_nombre* el código para identificar la seña.
21: **return true**

El Algoritmo 11 representa los pasos para verificar si una seña realizada con una mano específica del usuario coincide con la seña que interpretada en código en lenguaje C.

El primer paso es declarar una variable $_c$ que servirá de contador (ver línea 1), la cual, se incrementará cada que exista una coincidencia entre los dedos extendidos de la mano del usuario ($_manoA$) que realiza la seña y la mano ($_manoB$) con la que fue interpretada la seña. Posteriormente, se verifica qué tipo de mano (izquierda o derecha) se recibe como parámetro (ver línea 2), utilizando el método $begin()$ para posicionarnos en el primer elemento de la lista de manos, de la clase `HandList`, así como el método $isLeft()$, el cual permite saber si la mano es izquierda, de la clase `Hand` (ver Sección 2.2.1), si el método $isLeft()$ regresa **true**, se almacena 'L' en la variable $_manoA$, si regresa **flase**, se almacena 'R' en dicha variable.

El siguiente paso es comparar $_manoA$ con el tipo de mano de la variable $_manoB$, el cual se almacenó al interpretar la seña (ver línea 3). Si ambas manos son del mismo tipo, entonces el siguiente paso es verificar que el número de dedos extendidos ($_dedosA$) de la mano del usuario que realiza la seña, sea el mismo número de dedos extendidos ($_dedosB$) de la seña interpretada (ver línea 4).

Por último, se comprueba el tipo de cada uno de los dedos extendidos de la mano del usuario que realiza la seña, con el tipo de cada uno de los dedos extendidos de la seña interpretada, utilizando el método $type()$ de la clase `Finger`. Por cada coincidencia, se incrementa la variable $_c$, en uno (ver líneas de la 5 a la 7) y se comprueba si la variable $_c$ es igual a la variable $_dedosB$ (ver línea 8). Si es así, la función regresa **true** (ver línea 9). En caso contrario o si alguno de los pasos anteriores resulta no ser verdadero, la función regresa **false** (ver línea 10).

**Algorithm 11** Validación de una seña realizada con una mano específica del usuario

**Require:** La lista de una o ambas manos del usuario ($_listaManos$)

**Ensure:** Devuelve **true** si la seña es validada, en caso contrario, devuelve **false**

```
 1: _c = 0
 2: _manoA = _listaManos.begin().isLeft() ? 'L' :' R'
 3: if  _manoA == _manoB then
 4:    if _dedosA == _dedosB then
 5:       for _i = 0 a _manoA do
 6:          if _listaManos.begin().fingers().extended()[_i].type() == _dedos_ext[_i]
            then
 7:             _c++
 8:       if _c == _dedosB then
 9:          return true
10: return false
```

El código generado para una seña realizada con una mano cualquiera es similar al del Algoritmo 11, con la diferencia de que la condicional $if$ de la línea 3 se omite para no comprobar la mano del usuario que se utilizó al interpretar la seña.

El Algoritmo 12 muestra los pasos para validar una seña realizada con ambas manos del usuario. A diferencia de la forma de obtener la mano de la lista de manos que recibe la función del Algoritmo 11 (línea 6), el Algoritmo 12 utiliza los métodos $leftmost()$ y $rightmost()$, para obtener la mano izquierda y derecha, respectivamente. Ambos métodos forman parte de la clase `HandList`.

El primer paso es declarar una variable $_c$ que servirá de contador (ver línea 1), cada que exista una coincidencia entre los dedos extendidos de la mano izquierda ($_manoAL$) del usuario que realiza la seña y la mano izquierda ($_manoBL$), así como una coincidencia entre la mano derecha ($_manoAR$) del usuario que realiza la seña y la mano derecha ($_manoBR$) de la seña que fue interpretada.

Posteriormente, se debe verificar si la lista de manos ($_listaManos$) que la función recibe como parámetro contiene ambas manos del usuario (ver línea 2).

Si $_listaManos$ contiene ambas manos del usuario, el siguiente paso es verificar si ocurren las siguientes condiciones:

- el número de dedos extendidos de la mano izquierda ($_dedosAL$) sea el mismo número de dedos extendidos ($_dedosBL$) (ver línea 3).

- el número de dedos extendidos de la mano derecha ($_dedosAR$) sea el mismo número de dedos extendidos ($_dedosBR$) (ver línea 4).

Si ambas condiciones se cumplen, significa que el número de dedos coincide para cada una de las manos con las manos de la seña interpretada, entonces se comprueba que coincida el tipo de cada uno de los dedos extendidos de la mano izquierda del usuario que realiza la seña, con el tipo de cada uno de los dedos extendidos de la mano izquierda de la seña interpretada, utilizando el método $type()$ de la clase `Finger`. Por cada coincidencia encontrada se incrementa la variable $_c$ en uno (ver líneas de la 5 a la 7).

Después, se verifica que coincida el tipo de cada uno de los dedos extendidos de la mano derecha del usuario que realiza la seña, con el tipo de cada uno de los dedos extendidos de la mano derecha de la seña interpretada, utilizando el método $type()$ de la clase `Finger`. Por cada coincidencia encontrada se incrementa la variable $_c$ en uno (ver líneas de la 8 a la 10).

Finalmente, se comprueba si la variable $_c$ es igual a la suma de las variables $_dedosBL$ y $_dedosBR$ (ver línea 11), si es así, la función regresa **true** (ver línea 12), en caso con-

trario o si alguno de los pasos anteriores resulta no ser verdadero, la función regresa **false** (ver líneas 14, 16, 18 y 20).

**Algorithm 12** Validación de una seña realizada con ambas manos del usuario

**Require:** La lista de una o ambas manos del usuario ($_listaManos$)
**Ensure:** Devuelve **true** si la seña es validada, en caso contrario, devuelve **false**

```
1:  _c = 0
2:  if _listaManos contiene más de una mano then
3:     if _dedosAL == _dedosBL then
4:        if _dedosAR == _dedosBR then
5:           for _i = 0 a _manoAL do
6:              if _listaManos.leftmost().fingers().extended()[_i].type() ==
                _dedos_extBL[_i] then
7:                 _c++
8:           for _i = 0 a _manoAR do
9:              if _listaManos.rightmost().fingers().extended()[_i].type() ==
                _dedos_extBR[_i] then
10:                _c++
11:          if _c == (_dedosBL + _dedosBR) then
12:             return true
13:          else
14:             return false
15:       else
16:          return false
17:    else
18:       return false
19: else
20:    return false
```

### 4.4.2. Función generarInfoSenia()

La función $generarInfoSenia()$ es utilizada para almacenar en un documento de texto la información de una seña realizada por el usuario.

En el Algoritmo 13 se presentan los pasos para generar la información de una seña interpretada por esta función.

El primer paso consiste en extraer el número de manos (ver línea 1) utilizadas para realizar la seña ($_manosTotal$) mediante el método $count()$ de la clase `Hand` (ver Sección 2.2.1) y el total de dedos extendidos (ver línea 2), empleando la función $contarDedosExtendidos()$ (ver Sección 4.3.1) de ambas manos ($_dedosTotal$).

Después, se recorre la lista de manos utilizando los métodos $begin()$ y $end()$ de la

clase `HandList`, para obtener el objeto *_mano* para extraer la información de una o ambas manos del usuario (ver línea 3),como cuántos dedos extendidos tiene esa mano en específico y almacenar ese número en una variable llamada *_dedosMano* (ver línea 4).

Posteriormente, se recorren los dedos de esa mano (ver línea 5). En cada iteración se obtiene el objeto dedo (*_dedo*) y se extrae su información, como el *vector* de dirección al que apunta cada dedo de esa mano (*_dirccion*) y el número (*_tipo*) asignado por Leap Motion para cada uno de ellos (ver línea 6 y 7).

Finalmente, la función almacena el contenido de las variables *_manosTotal*, *_dedosTotal*, *_dedosMano*, *_direccion* y *_tipo* en el documento de texto (ver línea 8). Si la función guarda exitosamente la información , regresa **true**(ver línea 9). Cuando no se puede guardar la información en el documento, se regresa **false** pero se omitió, para mantener un algoritmo claro.

**Algorithm 13** Almacenar la información de una seña realizada por el usuario

**Require:** La lista de una o ambas manos del usuario (*_listaManos*) y un nombre para el documento y la función generada (*_name*).

**Ensure:** Devuelve **true** si se guarda el código de la seña interpretada en un documento de texto, en caso de existir un error al guardar, devuelve **false**

1: $_manosTotal = _listaManos.count()$
2: $_dedosTotal = contarDedosExtendidos(_listaManos)$
3: **for all** *_mano* en *_listaManos* **do**
4: $\quad _dedosMano = _mano.fingers().count()$
5: $\quad$ **for all** *_dedo* en *_mano* **do**
6: $\quad\quad _direccion = _dedo.direction()$
7: $\quad\quad _tipo = _dedo.type()$
8: $\quad\quad$ Guardamos los datos extraídos en el documento de texto *_name*.
9: **return true**

### 4.4.3. Función generarSeniaConDir()

Esta función tiene el propósito de generar el código en lenguaje C de una seña con dirección realizada por el usuario (i.e. obtener cada cuadrante en un plano cartesiano de 2D al que apunta cada dedo extendido del usuario).

La función *generarSeniaConDir*() hace uso de los siguientes métodos (ver Sección 2.2.1):

- *fingers*() para obtener una lista de dedos de una mano, de la clase `Hand`

- *count*() para contar el número de elementos de una lista, de la clase `Hand` y `Finger`

- *extended*(), que permite crear una lista de los dedos extendidos de una mano a partir de una lista de dedos, de la clase `FingerList`

- *leftmost*(), que devuelve el número de la mano más a la izquierda de un marco capturado por Leap Motion de la clase `HandList`

- *rightmost*(), que devuelve el número de la mano más a la derecha de un marco, ambos métodos de la clase `HandList`

En el Algoritmo 14 se presentan los pasos para generar el código de una seña con dirección.

El primer paso es verificar el número de manos de la lista *_listaManos* que se recibe como parámetro, si la función recibe una mano en la lista de manos del usuario (ver línea 1), entonces:

1. se crea la variable *_h*, la cual representa la mano que se recibió como parámetro (ver línea 2).

2. se comprueba si *_h* tiene al menos un dedo extendido, de ser así, se crea una lista de dedos extendidos (*_flExt*), posteriormente, se crea un arreglo (*_maxFingers*) de tamaño 10 (pues se almacenan dos datos para cada dedo) y se envía *_flExt* y el apuntador al arreglo *_maxNumDedos* a la funcion *obtenerCuadranteDedo*() (ver líneas de la 3 a la 6).

3. se verifica si se requiere guardar el código de la seña con dirección con una mano específica del usuario, de ser así, entonces se guarda en el documento el código de la seña con dirección realizada, en caso contrario, se guarda el código de una seña con dirección realizada con una mano cualquiera (ver líneas de la 7 a la 11).

Si se reciben ambas manos del usuario (ver línea 12), entonces:

1. se crea la variable *_hL*, la cual representa la mano izquierda y *_hR*, que representa la mano derecha (ver líneas 13 y 14).

2. se comprueba si *_hL* y *_hR* tienen al menos un dedo extendido cada una, de ser así, se crea una lista de todos los dedos extendidos de la mano izquierda (*_flExtL*) y de la mano derecha (*_flExtR*) (ver líneas de la 15 a la 17).

3. se crean dos arreglos de tamaño 10 (*_maxNumDedosL* y *_maxNumDedosR*); y se envía a *obtenerCuadranteDedo*() la lista de dedos extendidos de la mano izquierda (*_flExtL*) y el apuntador al arreglo *_maxNumDedosL* (ver líneas 18 y 19).

4. se envía a *obtenerCuadranteDedo*() la lista de dedos extendidos de la mano derecha ($_flExtR$) y el apuntador al arreglo $_maxNumDedosR$ (ver líneas 20 y 21).

5. se guarda el código de la seña con dirección realizada con dos manos del usuario (ver línea 22).

Si la función guarda de manera exitosa el código de la seña con dirección, entonces regresa **true** (ver línea 23).

**Algorithm 14** Almacenar el código en C de una seña con dirección realizada por el usuario

**Require:** La lista de una o ambas manos del usuario ($_listaManos$), un número para identificar si se requiere mano específica o mano cualquiera del usuario ($_aux$) y un nombre para el documento y la función generada ($_nombre$).

**Ensure:** Devuelve **true** si se guarda el código de la seña interpretada en un documento de texto, en caso de existir un error al guardar, devuelve **false**

```
1:  if _listaManos.count() == 1 then
2:    _h = _listaManos.leftmost()
3:    if _h.fingers().extended().count() != 0 then
4:      _flExt = _h.fingers().extended()
5:      _maxNumDedos[10]
6:      obtenerCuadranteDedo(_flExt, *_maxNumDedos)
7:      if _aux == 0) then
8:        Guardar en el documento _nombre el código para identificar la mano es-
          pecífica y la seña.
9:      else
10:       Guardar en el documento _nombre el código para identificar la seña.
11:   return true
12: else
13:   _hL = _listaManos.leftmost()
14:   _hR = _listaManos.rightmost()
15:   if _hL.fingers().extended().count() != 0 &&_hR.fingers().extended().count()
      != 0 then
16:     _flExtL = _hL.fingers().extended()
17:     _flExtR = _hR.fingers().extended()
18:     _maxNumDedosL[10]
19:     _maxNumDedosR[10]
20:     obtenerCuadranteDedo(_flExtL, *_maxNumDedosL)
21:     obtenerCuadranteDedo(_flExtR, *_maxNumDedosR)
22:     Guardar en el documento _nombre el código para identificar la seña.
23:   return true
```

La función *generarSeniaConDir*() almacena las mismas variables que la función *generarSeniaSinDir*() de la sección 4.4.1. Con la excepción de cómo se almacenan

los datos en el arreglo "int _dedos_ext = {$c$}", donde $c$ es un conjunto de parejas de la forma *Número asignado por Leap Motion, cuadrante.*

En el Algoritmo 14 se eliminaron las condicionales de error (**false**) cuando se guarda en el archivo de texto (línea 8, 10 y 22), para mantener un pseudocódigo más claro.

Al igual que la función *generarSeniaSinDir*() (ver Sección 4.4.1), las señas con dirección se pueden realizar con una o ambas manos del usuario, con la particularidad de que ahora se obtienen los cuadrantes de cada dedo extendido.

En el Algoritmo 15 se presentan los pasos para validar una seña con dirección realizada con una mano específica del usuario.

El primer paso es declarar una variable *_c* que servirá de contador (ver línea 1). Dicha variable se incrementará cada que exista una coincidencia entre el cuadrante de cada dedo extendido de la mano del usuario que realiza la seña (*_manoA*) y el cuadrante de cada dedo extendido de la mano con la que fue interpretada la seña (*_manoB*).

Posteriormente, se debe verificar si la lista de manos contiene una sola mano del usuario, si es así, entonces se comprueba si la variable *_manoA* es igual a la variable *_manoB* (ver líneas 2, 3 y 4).

Si ambas manos son iguales, entonces el siguiente paso es verificar que el número de dedos extendidos (*_dedosA*), de la mano del usuario que realiza la seña, sea el mismo número de dedos extendidos (*_dedosB*), de la mano de la seña interpretada (ver línea 5).

Si el número de dedos coincide, se crea una lista de dedos extendidos (*_flExt*) y un arreglo (*_maxNumDedos*) de tamaño 10, y se envían como parámetros a la función *obtenerCuadranteDedo*() (ver líneas de la 6 a la 8).

Después, se comprueba que coincida el tipo y el cuadrante al que apunta cada uno de los dedos extendidos de la mano del usuario que realiza la seña, con el tipo y el cuadrante al que apunta cada uno de los dedos extendidos de la seña interpretada, utilizando el método *type*() de la clase `Finger` para obtener el tipo de dedo. Si se encuentra una coincidencia, entonces se incrementa la variable *_c* en uno (ver líneas de la 9 a la 12).

Finalmente, se comprueba si la variable *_c* es igual a *_dedosB*, si es así, la función regresa **true** (ver líneas 13 y 14), en caso contrario o si alguno de los pasos anteriores resulta no ser verdadero, la función regresa **false** (ver líneas 16, 18 y 20).

En caso de requerir la validación de una seña realizada con una mano cualquiera, los pasos para validarla son los mismos que el Algoritmo 15, con la particularidad de que se eliminan las condicionales **if** y **else** de las líneas cuatro y diecinueve, respectivamente.

---

**Algorithm 15** Validación de una seña con dirección realizada con una mano específica del usuario

---

**Require:** La lista de una o ambas manos del usuario ($_listaManos$)
**Ensure:** Devuelve **true** si la seña es validada, en caso de no validarse, devuelve **false**

1: $_c = 0$
2: **if** $_listaManos.count()$ != 0 **then**
3:    **return false**
4: **if** $_manoA == _manoB$ **then**
5:    **if** $_dedosA == _dedosB$ **then**
6:       $_flExt = _h.fingers().extended()$
7:       $_maxNumDedos[10]$
8:       $obtenerCuadranteDedo(_flExt, *_maxNumDedos)$
9:       **for** $_i = 0$ a $_manoA$ **do**
10:          **if** $_listaManos.begin().fingers().extended()[_i].type() == _dedos_ext[_i]$ **then**
11:             **if** el cuadrante del dedo extendido $_manoA$ es igual al cuadrante del dedo de $_manoB$ **then**
12:                $_c{+}{+}$
13:       **if** $_c == _dedosB$ **then**
14:          **return true**
15:       **else**
16:          **return false**
17:    **else**
18:       **return false**
19: **else**
20:    **return false**

---

La validación de una seña con dirección realizada con ambas manos del usuario (i.e. con la información del cuadrante al que apunta cada uno de los dedos extendidos de cada mano del usuario) se presenta en el Algoritmo 16.

Cuando la función $generarSeniaConDir()$ recibe dos manos, almacena variables para la mano izquierda "int _dedos_extL = $\{c\}$" y una para la mano derecha "int _dedos_extR = $\{d\}$", donde $c$ y $d$ son un conjunto de parejas de la forma *Número asignado por Leap Motion, cuadrante.*

El primer paso es declarar una variable $_c$ que servirá de contador (ver línea 1),

cada que exista una coincidencia entre, el número de dedos extendidos de ambas manos del usuario que realiza la seña y las manos de la seña interpretada; así como una coincidencia entre el cuadrante de cada uno de los dedos extendidos de ambas manos del usuario que realiza la seña y los cuadrantes de los dedos extendidos de ambas manos de la seña interpretada.
Posteriormente, se debe verificar si la lista de manos ($_listaManos$) que la función recibe como parámetro contiene ambas manos del usuario (ver línea 2).

Si $_listaManos$ contiene ambas manos del usuario, el siguiente paso es verificar si ocurren las siguientes condiciones:

- el número de dedos extendidos de la mano izquierda del usuario ($_dedosAL$) sea el mismo número de dedos extendidos ($_dedosBL$) de la mano de la seña interpretada (ver línea 3).

- el número de dedos extendidos de la mano derecha del usuario ($_dedosAR$) sea el mismo número de dedos extendidos ($_dedosBR$) de la mano de la seña interpretada (ver línea 4).

Si el número de dedos coincide, entonces se crea una lista de dedos extendidos ($_flExtL$) de la mano izquierda del usuario y una lista de dedos extendidos ($_flExtR$) para mano derecha del usuario (ver líneas 5 y 6).

Posteriormente se crean dos arreglos ($_maxNumDedosL$ y $_maxNumDedosR$) de tamaño 10 cada uno (ver líneas 7 y 8).

Después, se envían como parámetros a la función *obtenerCuadranteDedo*() la lista de dedos extendidos de la mano izquierda y el apuntador del arreglo $_maxNumDedosL$, y de la misma forma se envía la lista $_flExtR$ y el apuntador al arreglo $_maxNumDedosR$, en otra llamada a la función *obtenerCuadranteDedo*() (ver líneas 9 y 10).

Después, se comprueba que coincida el tipo y el cuadrante al que apunta cada uno de los dedos extendidos de la mano izquierda del usuario con el tipo y el cuadrante al que apunta cada uno de los dedos extendidos de la mano izquierda de la seña interpretada, utilizando el método *type*() de la clase `Finger` para obtener el tipo de dedo. Si el número que les asigna a Leap Motion y el cuadrante al que apuntan coincide con su homólogo, se incrementa a $_c$ en uno (ver líneas de la 11 a la 14). Por otro lado, se hace el mismo procedimiento anterior, pero para la mano derecha del usuario (ver líneas de la 15 a la 18).

Finalmente, se comprueba si $_c$ es igual a la suma de $_dedosAL$ y $_dedosAR$ (ver línea 19), si es así, la función regresa **true** (ver línea 20), en caso contrario o si alguno de los pasos anteriores resulta no ser verdadero, la función regresa **false** (ver líneas 22, 24, 26 y 28).

**Algorithm 16** Validación de la una seña con dirección realizada con ambas manos del usuario

**Require:** La lista de una o ambas manos del usuario ($_listaManos$)
**Ensure:** Devuelve **true** si la seña es validada, en caso de no validarse, devuelve **false**

```
1: _c = 0
2: if la lista contiene más de una mano then
3:    if _dedosAL == _dedosBL then
4:       if _dedosAR == _dedosBR then
5:          _flExtL = _hL.fingers().extended()
6:          _flExtR = _hR.fingers().extended()
7:          _maxNumDedosL[10]
8:          _maxNumDedosR[10]
9:          obtenerCuadranteDedo(_flExtL, *_maxNumDedosL)
10:         obtenerCuadranteDedo(_flExtR, *_maxNumDedosR)
11:         for _i = 0 a _manoA do
12:            if _flExtL[_i].type() == _dedos_extBL[_i] then
13:               if el cuadrante del dedo extendido _manoAL es igual al cuadrante del
                  dedo de _manoB then
14:                  _c++
15:         for _i = 0 a _manoA do
16:            if _flExtR[_i].type() == _dedos_extBR[_i] then
17:               if el cuadrante del dedo extendido _manoA es igual al cuadrante del
                  dedo de _manoB then
18:                  _c++
19:         if _c == (_dedosBL + _dedosBR then
20:            return true
21:         else
22:            return false
23:      else
24:         return false
25:   else
26:      return false
27: else
28:    return false
```

# Capítulo 5

# Implementación de las aplicaciones de prueba

Este capítulo está dividido en cuatro secciones, las cuales son: 1) Sección 5.1, donde se explican las aplicaciones mono-usuario que se desarrollaron para comprobar la flexibilidad del *toolkit* al ser aplicable tanto a aplicaciones multi-usuario como mono-usuario, 2) Sección 5.2, en la que se exponen las características de las aplicaciones multi-usuario creadas, para comprobar la efectividad del *toolkit* desarrollado, 3) la Sección 5.3, contiene una tabla comparativa, la cual presenta las características generales de las aplicaciones y, finalmente, 4) la Sección 5.4, que presenta las pruebas hechas a diferentes usuarios de las aplicaciones mono-usuario y multi-usuario.

## 5.1. Aplicaciones mono-usuario

En esta sección se presentan las aplicaciones mono-usuario cuyo, propósito es comprobar la flexibilidad de nuestro *toolkit*. Aunque el objetivo de este trabajo de investigación es desarrollar un *toolkit* para aplicaciones multi-usuario, no se limita a éstas, pues permite utilizar las funciones para implementar aplicaciones mono-usuario, haciendo del *toolkit* desarrollado una herramienta flexible y efectiva.

La Sección 5.1.1 presenta una aplicación de consola denominada Señas, cuyo propósito es permitir al programador crear funciones que validen señas con o sin dirección, utilizando la información de una mano específica (izquierda o derecha) para posteriormente validar dicha seña sí, y solo sí, se utiliza la mano elegida. De la misma forma, permite crear señas sin utilizar dicha información de las manos para validar la seña realizada.

Por otra parte, la aplicación también permite crear funciones de señas que utilicen ambas manos. Así mismo, permite interpretar una seña realizada por el usuario y guardar la información de dicha seña en un documento de texto. En la Sección 5.1.2,

se describen las características de la aplicación Ratón, la cual tiene como propósito permitir al usuario controlar el puntero del ratón, del sistema operativo Windows, mediante el dispositivo Leap Motion y un conjunto de señas. Posteriormente, en la Sección 5.1.3, se exponen las singularidades de la aplicación Rompecabezas, que utiliza la aplicación Ratón (ver Sección 5.1.2) para controlar el puntero y, de esta forma, desplazar las diferentes secciones de la imagen del rompecabezas, mediante la selección de un trozo de imagen. La aplicación Rompecabezas, utiliza la función *calcularColision*() (ver Sección 4.3.2) para realizar la selección de un trozo de imagen (i.e. detectar si existe una colisión entre un trozo de imagen y el punto generado por el centro de la palma del usuario).

### 5.1.1. Crear señas para Leap Motion

Esta aplicación que permite al usuario crear funciones en lenguaje C de señas personalizadas. Así mismo permite nombrarlas y guardarlas en documentos de texto, para posteriormente, ser utilizadas en cualquier aplicación que utilice la API de Leap Motion.

En la Figura 5.1 se muestra un ejemplo de la ejecución de la aplicación, cuyo menú consta de dos niveles, que son: 1)el primer nivel permite al usuario interpretar una seña, guardar la información de dicha seña y salir de la aplicación y 2) el segundo nivel pertenece a la opción de interpretar señas, donde el usuario puede crear el número de señas que desee. En el segundo nivel, el usuario puede crear funciones de señas sin dirección utilizando la información de una mano específica del usuario (izquierda o derecha) para posteriormente ser validada, utilizando dicha información (ver nivel 2.1). Así mismo, ofrece la opción de interpretar señas sin dirección que no utilicen la información de una mano en específico para poder ser validadas (ver nivel 2.2). Por otro lado, el usuario puede elegir interpretar señas sin dirección que utilicen las dos manos. Finalmente, la aplicación cuenta con una sección $A$, que es el resultado de cualquier elección que el usuario realice (exceptuando *salir*) incluida la segunda opción del primer nivel. Esta sección permite al usuario nombrar su seña, así como, el documento donde se almacenará código para validar la seña del usuario. Como se mencionó al inicio de esta sección, la función generada se puede copiar y pegar en cualquier aplicación que utilice Leap Motion.

La aplicación Señas, utiliza la API de procesamiento de dedos y vectores, así como la API de interpretación de señas.

```
Nivel 1   1.-Crear una sena
          2.-almacenar informacion de una sena
          3.-Salir

Nivel 2      1.-Crear la sena con una mano especifica
             2.-Crear la sena con una mano cualquiera
             3.-Crear la sena con dos manos
             4.-Crear una sena con direccion
                hacia los cuadrantes (2D) / cualquier mano
             5.-Crear una sena con direccion
                hacia los cuadrantes (2D) / mano especifica
             6.-Crear una sena con direccion
                hacia los cuadrantes (2D) / dos manos
             7.-Salir

  A        ┐Que nombre desea ponerle a la sena?
           tijeras
           A continuacion realice su sena y espere 10 segundos
           para que se guarde
```

Figura 5.1: Menú para crear señas para el dispositivo Leap Motion

### 5.1.2. Ratón

Esta aplicación que utiliza una seña específica para permitir que el usuario interaccione con el sistema operativo, como se muestra en la Figura 5.3. Para que el usuario pueda mover el puntero del ratón, basta con mover su mano hacia arriba, abajo, izquierda o derecha.

La aplicación Ratón fue diseñada con el propósito de poner a prueba la validación de una seña específica creada por el usuario y enviar el evento de clic al sistema operativo.

Esta aplicación puede ser usada para controlar completamente el sistema operativo $Windows$, debido a que la aplicación es flexible, pues se podría modificar el código fuente de la misma para cambiar el número de señas que valida y de esta forma generar más eventos y expandir su funcionamiento.

El diagrama de casos de uso de la aplicación Ratón se muestra en la Figura 5.2

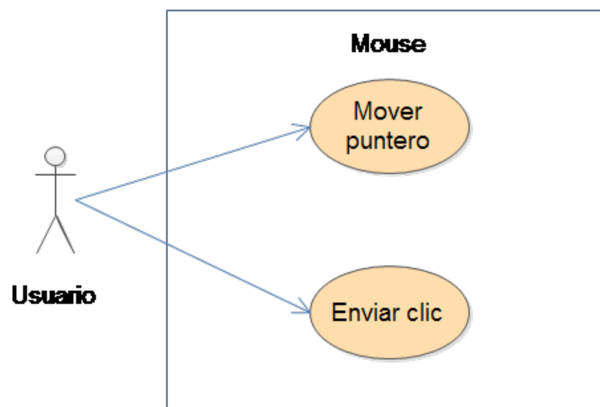


Figura 5.2: Diagrama de casos de uso de la aplicación Ratón

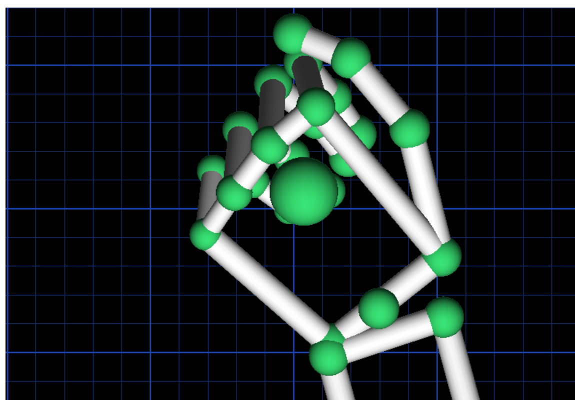

Figura 5.3: Seña para emular el clic del ratón

| CASO DE USO: | Mover puntero |
|---|---|
| ACTOR: | Usuario |
| DESCRIPCIÓN: | Permite mover el puntero del ratón |
| PRECONDICIONES: | |
| CURSO NORMAL: | ALTERNATIVAS: |
| 1) El usuario puede mover el puntero del ratón moviendo una de sus manos (izquierda o derecha) hacia arriba, abajo, derecha, izquierda y/o diagonales. | |

Cuadro 5.1: Caso de uso para mover el ratón utilizando una mano cualquiera del usuario

| CASO DE USO: | Enviar clic |
|---|---|
| ACTOR: | Usuario |
| DESCRIPCIÓN: | Permite enviar el evento de clic al sistema operativo |
| PRECONDICIONES: | |
| CURSO NORMAL: | ALTERNATIVAS: |
| 1) El usuario debe realizar la seña de clic para enviar el evento al sistema operativo. | Dejar de realizar la seña para dejar de enviar evento de clic. |

Cuadro 5.2: Caso de uso para enviar el evento de clic al sistema operativo utilizando una seña en particular

### 5.1.3. Rompecabezas

Utiliza la aplicación Ratón (Sección 5.2) para seleccionar y mover trozos de una imagen en cualquier dirección. Esta aplicación fue desarrollada para probar la aplicación Ratón,la validación de señas y la función de detectar colisiones del *toolkit* creado. funciona con la misma seña para enviar evento de clic de Ratón y una seña (ver Figura 5.5) para salir de la aplicación.

En la Figura 5.4 se muestran los casos de uso de la aplicación Rompecabezas.

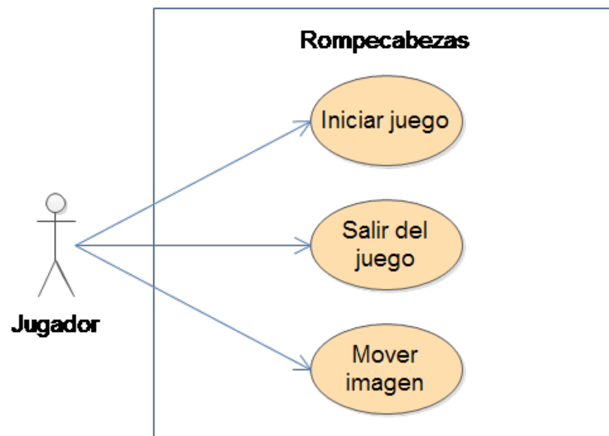


Figura 5.4: Diagrama de casos de uso de la aplicación *Rompecabezas*

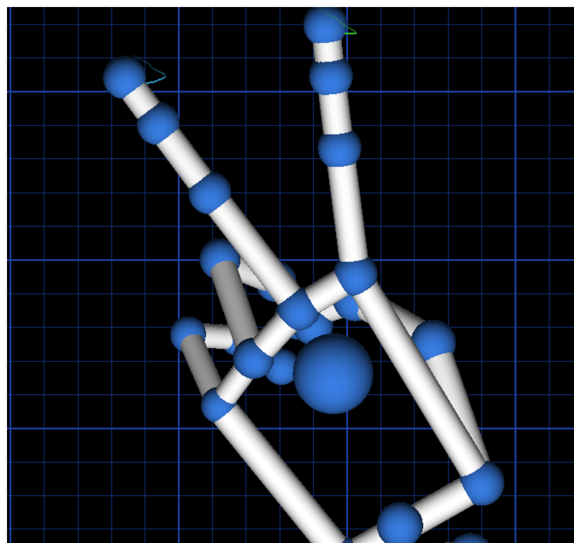

Figura 5.5: Seña para finalizar la ejecución de la aplicación Rompecabezas

| Caso de uso: | Iniciar juego |
|---|---|
| Actor: | Jugador |
| Descripción: | Permite comenzar el juego |
| Precondiciones: | |
| Curso normal: | Alternativas: |
| 1) Poder mover las imágenes. | |
| 2) Salir del juego. | |

Cuadro 5.3: Caso de uso para comenzar el juego de *Rompecabezas*

| Caso de uso: | Mover imagen |
|---|---|
| Actor: | Jugador |
| Descripción: | Permite mover cualquier trozo de la imagen principal |
| Precondiciones: | Haber iniciado el juego |
| Curso normal: | Alternativas: |
| 1) El usuario debe realizar la seña de clic sobre cualquier imagen. | Dejar de realizar la seña para dejar de enviar evento de clic y soltar la imagen. |

Cuadro 5.5: Caso de uso para mover una imagen del *Rompecabezas*

| Caso de uso: | Salir del juego |
|---|---|
| Actor: | Jugador |
| Descripción: | Permite a cualquier jugador abandonar el juego. |
| Precondiciones: | Haber iniciado el juego |
| Curso normal: | Alternativas: |
| 1) Realizar la seña que sirve para salir del juego. | |

Cuadro 5.4: Caso de uso para salir del juego utilizando una seña

## 5.2. Aplicaciones multi-usuario

Los entornos multi-usuario elegidos para este documento de tesis tienen como finalidad comprobar la eficiencia del *toolkit* de disminuir el trabajo de desarrollar técnicas o mecánicas que permitan al programador controlar el uso de la aplicación colaborativa.

Se eligieron juegos multi-usuario con jugadores físicamente distantes puesto que Leap Motion tiene un mayor uso en el campo de los videojuegos y además, estas aplicaciones colaborativas tienen el propósito de permitir a dos usuarios interaccionar tanto con la aplicación como con un Leap Motion conectado a una computadora, cuyo trabajo es transmitir los datos de la aplicación sobre una red hacía la computadora del otro usuario, estableciendo una comunicación cliente/servidor.

Las aplicaciones que fueron utilizadas para poner a prueba las funciones del *toolkit* creado son, en su mayoría, juegos pues Leap Motion es mayormente explotado en este tipo de aplicaciones, además de que el usuario podrá disfrutar mientras interacciona con las aplicaciones y con los dispositivos Leap Motion.

En seguida se presenta el algoritmo 17, el cual sirve para explicar el bucle de ejecución del cliente, para las aplicaciones PingPong (sección 5.2.1) y Zombis invasores (sección 5.2.2).

El primer paso es conectarnos al servidor y mientras el cliente no salga del bucle (ver líneas 1 y 2), se verifica si el jugador realizó un movimiento hacía abajo, en el juego de PingPong, o izquierda en el juego de Zombis invasores (ver línea 3). Si es así, entonces se almacena la letra 'D' en una variable *char* de tipo global denominada *_estadoC*, la cual alacena el estado de juego del cliente y envía el contenido de dicha variable al servidor (ver líneas 4 y 5).

Por otro lado, se comprueba si el cliente realizó un movimiento hacia arriba, en el juego de PingPong, o derecha para el juego de Zombis invasores (ver línea 6). De ser cierto, entonces se almacena en la variable *_estadoC* la letra 'U' y se envía el contenido de dicha variable al servidor (ver líneas 7 y 8).

En caso de que ningún movimiento sea realizado (ver línea 9), entonces se almacena en la variable *_estadoC* la letra 'I' y se envía al servidor (ver líneas 10 y 11 ). Finalmente, recibimos los datos de la respuesta del servidor y mostramos los objetos en pantalla (ver líneas 12 y 13).
A continuación se expone el algoritmo 18, el cual detalla el bucle de ejecución del servidor, para las aplicaciones PingPong (ver Sección 5.2) y Zombis invasores (ver Sección 5.3).

**Algorithm 17** Bucle de ejecución del cliente

1: Conectar con el servidor
2: **while** !*salir* **do**
3: **if** se ha movido el jugador hacia abajo/izquierda **then**
4: $_estadoC$ = 'D'
5: Enviar $_estadoC$ al servidor
6: **else if** se ha movido el jugador hacia arriba/derecha **then**
7: $_estadoC$ = 'U'
8: Enviar $_estadoC$ al servidor
9: **else**
10: $_estadoC$ = 'I'
11: Enviar $_estadoC$ al servidor
12: $_estadoS$ = dato recibido
13: Mostrar objetos en pantalla

El primer paso es esperar a que el cliente se conecte al servidor (ver línea 1). Mientras el servidor no salga del bucle (ver línea 2), se verifican los movimientos del servidor. Si éste realizó un movimiento hacía abajo, en el juego de PingPong, o izquierda en el juego de Zombis invasores (ver línea 3). Si es así, entonces se almacena la letra 'D' en una variable *char* de tipo global denominada $_estadoServidor$ (ver línea 4).

Por otro lado, se comprueba si el servidor realizó un movimiento hacia arriba, en el juego de PingPong, o derecha para el juego de Zombis invasores (ver línea 5). De ser cierto, entonces se almacena en la variable $_estadoServidor$ la letra 'U' (ver línea 6).

En caso de que ningún movimiento sea realizado (ver línea 7), entonces se almacena en la variable $_estadoServidor$ la letra 'I' (ver línea 8 ).

Posteriormente, espera la llegada de un dato del cliente (ver línea 9) y se almacena en una variable *char* de tipo global llamada $_estadoCliente$ (ver línea 10). El siguiente paso es comprobar si $_estadoCliente$ contiene un movimiento hacia arriba (i.e. contiene la letra 'U'), en el juego de PingPong, o derecha para el juego de Zombis invasores (ver línea 11). Si es así, entonces el servidor mueve el personaje del cliente en el juego hacia arriba o hacia la derecha y envía el contenido de la variable $_estadoServidor$ al cliente (ver líneas 12 y 13).

Por otra parte, si el servidor recibió un movimiento hacía abajo (i.e. contiene la letra 'D'), en el juego de PingPong, o izquierda en el juego de Zombis invasores (ver línea 14), entonces el servidor mueve el personaje del cliente en el juego hacia abajo o hacia la izquierda y envia el contenido de la variable $_estadoServidor$ al cliente (ver líneas 15 y 16).

Finalmente, si la variable *_estadoCliente* contiene un movimiento nulo (i.e. contiene la letra 'I'), entonces, el servidor envía el contenido de la variable *_estadoServidor* al cliente (ver líneas 17 y 19).

**Algorithm 18** Bucle de ejecución del servidor

```
 1: Esperar conexión con el cliente
 2: while !salir do
 3:    if se ha movido el jugador hacia abajo/izquierda then
 4:       _estadoServidor = 'D'
 5:    else if se ha movido el jugador hacia arriba/derecha then
 6:       _estadoServidor = 'U'
 7:    else
 8:       _estadoServidor = 'I'
 9:    Esperar dato del cliente.
10:    _estadoCliente = dato recibido
11:    if _estadoCliente == 'U' then
12:       Mostrar objetos en pantalla
13:       Enviar _estadoServidor al cliente
14:    else if _estadoCliente == 'D' then
15:       Mostrar objetos en pantalla
16:       Enviar _estadoServidor al cliente
17:    else
18:       Enviar _estadoServidor al cliente
```

### 5.2.1. PingPong

Esta aplicación está hecha para que dos usuarios remotos compitan entre ellos y fue realizada para explotar las funciones de manejo de usuarios utilizando listas para controlar el momento en que un usuario se agrega, elimina y/o busca en una lista de usuarios conectados. En la Figura 4.1 se precia una vista de la aplicación *PingPong*. El juego está pensado para utilizar la comunicación cliente/servidor con *sockets* de *Windows* y soporta la interacción de dos usuarios mediante dispositivos Leap Motion conectados a una computadora por separado. La aplicación *PingPong* permite que un usuario (jugador 1) desempeñe el papel de *host* (servidor) y que otro usuario (jugador 2) realice el papel de cliente.

El usuario puede controlar la raqueta para jugar con el simple movimiento de subir o bajar cualquiera de sus manos (izquierda o derecha) para que la raqueta suba o baje respectivamente.

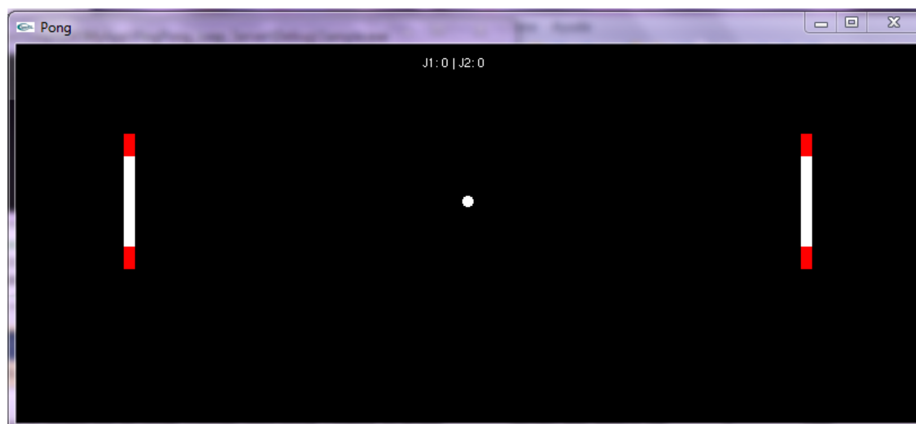


Figura 5.6: El clásico juego *PingPong*

A continuación se presenta el diagrama de casos uso del juego *PingPong* en la Figura 4.3.

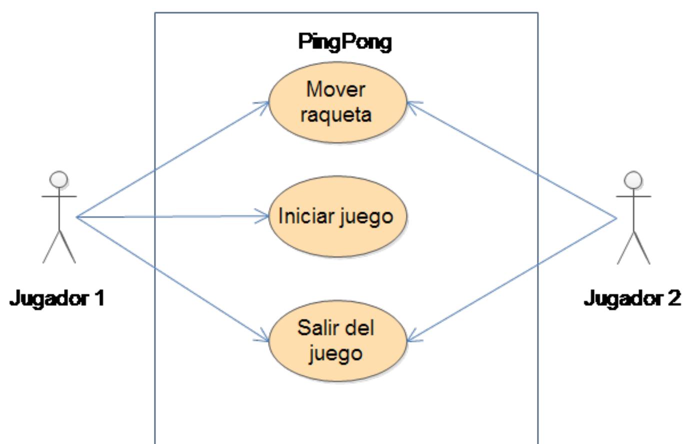


Figura 5.7: Caso de uso de la aplicación *PingPong*

### 5.2.2. Zombis invasores

Este juego permite que dos usuarios físicamente distantes colaboren en la interacción con la aplicación y los dispositivos Leap Motion, para alcanzar la meta del juego que es destruir a todas las naves enemigas. Esta aplicación fue realizada para explotar las funciones de manejo de usuarios, de igual forma que lo hace PingPong (sección 5.2.1) y funciones de validación de señas con una mano cualquiera del usuario.

En la Figura 5.9 se observa el funcionamiento del juego, donde la nave de cualquier usuario puede moverse a la derecha de la pantalla levantando la mano derecha o a la

| CASO DE USO: | Mover raqueta |
|---|---|
| ACTOR: | Jugador 1, jugador 2 |
| DESCRIPCIÓN: | Modifica la posición de las raquetas en la pantalla |
| PRECONDICIONES: | Haber iniciado el juego |
| CURSO NORMAL: | ALTERNATIVAS: |
| 1) Alzar cualquiera de las manos (izquierda o derecha) para mover la raqueta hacia arriba. | |
| 2) Bajar cualquiera de las manos para mover la raqueta hacia abajo | |

Cuadro 5.6: Caso de uso para mover las raquetas del juego utilizando Leap Motion

| CASO DE USO: | Iniciar juego |
|---|---|
| ACTOR: | Jugador 1 |
| DESCRIPCIÓN: | Permite comenzar el juego |
| PRECONDICIONES: | |
| CURSO NORMAL: | ALTERNATIVAS: |
| 1) Poder mover las raquetas. | |
| 2) Salir del juego. | |

Cuadro 5.7: Caso de uso para comenzar el juego de *PingPong*

| CASO DE USO: | Salir del juego |
|---|---|
| ACTOR: | Jugador 1, jugador 2 |
| DESCRIPCIÓN: | Permite a cualquier jugador abandonar el juego |
| PRECONDICIONES: | Haber iniciado el juego |
| CURSO NORMAL: | ALTERNATIVAS: |
| 1) Realizar la seña que sirve para salir del juego. | |

Cuadro 5.8: Caso de uso para salir del juego utilizando una seña

izquierda con la otra mano. A demás, el usuario puede elegir disparar utilizando la seña de disparo (ver Figura 5.8).

En seguida, se muestra el diagrama de casos de uso de la aplicación *Zombis invasores* (Figura 5.10).

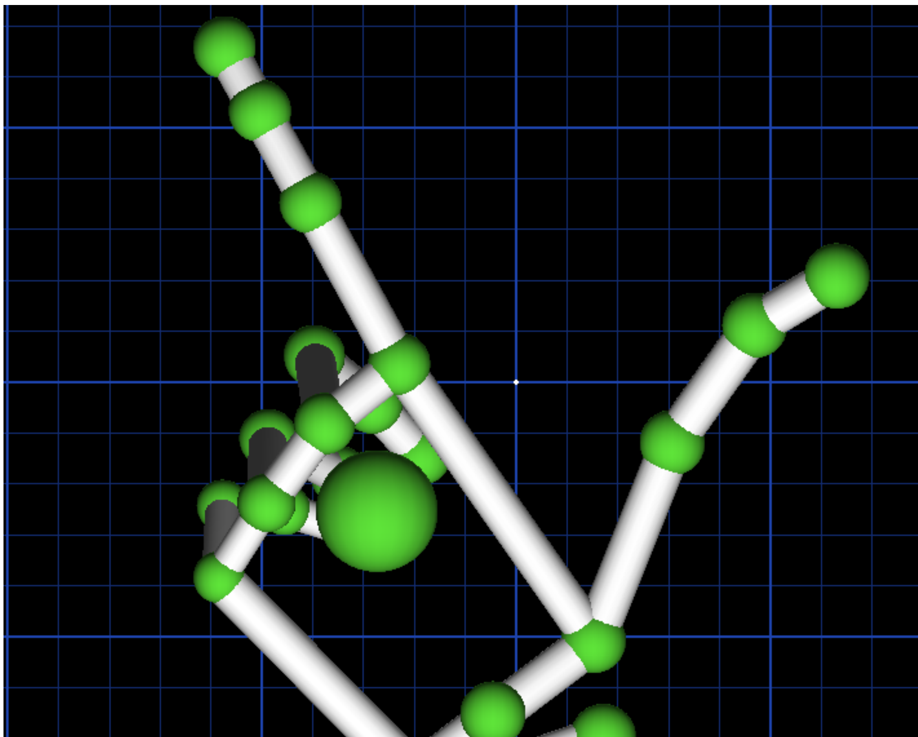

Figura 5.8: Seña para realizar un disparo

Figura 5.9: Pantalla de juego

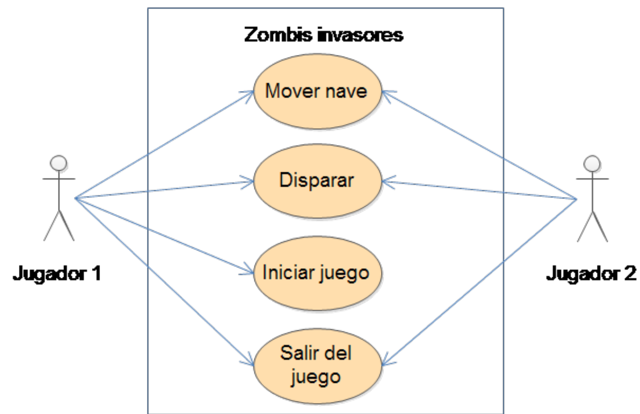


Figura 5.10: Diagrama de casos de uso de la aplicación *Zombis invasores*

| Caso de uso: | Mover nave |
|---|---|
| Actor: | Jugador 1, jugador 2 |
| Descripción: | Permite a cualquier jugador mover su nave |
| Precondiciones: | Haber iniciado el juego |
| Curso normal: | Alternativas: |
| 1) El jugador 1 y 2 pueden mover su nave utilizando la mano derecha para ir a la derecha del juego. | Utilizar mano izquierda para ir a la izquierda del juego |

Cuadro 5.9: Caso de uso para mover las naves del juego utilizando una mano especifica de cada usuario

| Caso de uso: | Disparar |
|---|---|
| Actor: | Jugador 1, jugador 2 |
| Descripción: | Permite a cualquier jugador disparar el láser |
| Precondiciones: | Haber iniciado el juego |
| Curso normal: | Alternativas: |
| 1) Realizar la seña seleccionada para disparar láser | |

Cuadro 5.10: Caso de uso para disparar láser utilizando una seña

| Caso de uso: | Iniciar juego |
|---|---|
| Actor: | Jugador 1 |
| Descripción: | Permite comenzar el juego |
| Precondiciones: | |
| Curso normal: | Alternativas: |
| 1) Poder mover las naves. | |
| 2) Salir del juego. | |

Cuadro 5.11: Caso de uso para comenzar el juego de *Zombis invasores*

| Caso de uso: | Salir del juego |
|---|---|
| Actor: | Jugador 1, jugador 2 |
| Descripción: | Permite a cualquier jugador abandonar el juego. |
| Precondiciones: | Haber iniciado el juego |
| Curso normal: | Alternativas: |
| 1) Realizar la seña que sirve para salir del juego. | |

Cuadro 5.12: Caso de uso para salir del juego utilizando una seña

## 5.3. Características generales de las aplicaciones

A continuación se presenta un cuadro con las características generales de las aplicaciones, así como las funciones utilizadas para poner a prueba la eficiencia del *toolkit*.

| Aplicación | ¿Es multi-usuario? | Funciones derivadas del *toolkit* |
|---|---|---|
| *PingPong* | Sí | ▪ $verificarLista()$ ▪ $contarId()$ ▪ $obtenerLista()$ ▪ $borrarId()$ ▪ $obtenerId()$ ▪ $asignarId()$ |
| *Zombis invasores* | Sí | ▪ $verificarLista()$ ▪ $contarId()$ ▪ $obtenerLista()$ ▪ $borrarId()$ ▪ $obtenerId()$ ▪ $asignarId()$ ▪ Funciones creadas a partir $generarSeniaSinDir()$ |
| Ratón | No | ▪ Funciones creadas a partir $generarSeniaSinDir()$ |
| *Rompecabezas* | No | ▪ Funciones creadas a partir $generarSeniaSinDir()$ ▪ $calcularColision()$ |
| *Crear señas para Leap Motion* | No | ▪ $contarDedosExtendidos()$ ▪ $generarSeniaSinDir()$ ▪ $generarInfoSenia()$ |

Cuadro 5.13: Características de aplicaciones creadas

## 5.4. Pruebas con usuarios finales

Las aplicaciones anteriormente descritas en las Secciones 5.1 y 5.2, sirvieron para poner aprueba el *toolkit* desarrollado. El conjunto de personas que fueron seleccionadas tienen conocimientos tanto básicos como avanzados de programación, así como del funcionamiento de las aplicaciones multi-usuario e Interacción Humano Computadora.

El cuestionario utilizado para las pruebas es el proporcionado por la NASA denominado Task Load Index (NASA TLX por sus siglas en inglés), el cual tiene el único propósito de evaluar la carga de trabajo al realizar ciertas tareas, en este caso, la carga que los usuarios tienen al momento de interactuar con las aplicaciones y los dispositivos Leap Motion. Dicho cuestionario está dirigido a las principales actividades de las aplicaciones. Por ejemplo, para todas las aplicaciones se evalúan tres tareas, las cuales son: 1) realización de una seña con una mano, 2) realización de una seña con ambas manos y 3) la interacción del usuario con la aplicación mediante el dispositivo Leap Motion. El primer cuestionario para medir la carga de trabajo al realizar una seña con una mano del usuario está dividido en seis subescalas y son:

1. Exigencia de tipo mental: ¿cuánta actividad mental y perceptiva fue necesaria para comprender la forma en que se realiza una seña?

2. Exigencia de tipo físico: ¿cuánta actividad física fue necesaria para realizar la seña?

3. Exigencia temporal ¿cuánta presión de tiempo sintió, debido al ritmo al cual se realizaba la seña?

4. Esfuerzo ¿en qué medida ha tenido que trabajar (física o mentalmente) para alcanzar el nivel de desempeño requerido?

5. Rendimiento ¿hasta qué punto cree que ha tenido éxito al crear la seña y cuál es su grado de satisfacción con su desempeño?)

6. Nivel de frustración ¿cuál fue su nivel de frustración al realizar la seña?

La tarea del usuario es asignar un puntaje a cada subescala de 0 a 100, de los cuales se determina la carga de trabajo total que los usuarios presentan al momento de interactuar con la aplicación y el dispositivo.

Posteriormente, se presenta un conjunto de pares para determinar el peso que representa la carga más pesada de la tarea y son:

- Física-Mental
- Temporal-Física
- Temporal-Frustración
- Temporal-Mental
- Rendimiento-Física
- Temporal-Esfuerzo
- Rendimiento-Mental
- Frustración-Física
- Rendimiento-Frustración
- Frustración-Mental
- Esfuerzo-Física
- Rendimiento-Esfuerzo
- Esfuerzo-Mental
- Temporal-Rendimiento
- Esfuerzo-Frustración

El peso se determina a partir de la elección de pares y el rango es [0,5], donde 5 es el miembro del par más elegido y 0 el que no se eligió. La puntuación final se obtiene mediante el calculo de la Fórmula 5.1.

$$Puntuación \quad ponderada \quad = \quad puntaje \quad x \quad peso \tag{5.1}$$

El resultado puede observarse en la Figura 5.7 en la cual se aprecia la cantidad asignada por cada usuario a cada una se las subclases.

Enseguida se presenta en Cuadro 5.14 donde se exponen los datos resultantes de la aplicación del cuestionario, así como el promedio $\overline{x}$ y la desviación estándar $\sigma$ para conocer la dispersión de los datos.

| Usuario | Exigencia Mental | Exigencia Física | Exigencia Temporal | Esfuerzo | Rendimiento | Frustración |
|---|---|---|---|---|---|---|
| 1 | 40 | 180 | 0 | 110 | 360 | 45 |
| 2 | 30 | 15 | 25 | 60 | 480 | 45 |
| 3 | 20 | 15 | 10 | 15 | 380 | 0 |
| 4 | 20 | 30 | 15 | 15 | 270 | 0 |
| 5 | 25 | 80 | 25 | 40 | 460 | 30 |
| $\overline{x}$ | 27 | 64 | 15 | 48 | 390 | 24 |
| $\sigma$ | 7,48 | 62,72 | 9,48 | 36,60 | 75,36 | 20,34 |

Cuadro 5.14: Resultados del cuestionario de carga de trabajo para la realización de una seña con una mano del usuario

A continuación se analizan los resultados obtenidos por cada subescala:

- Exigencia Mental: los usuarios entendieron de manera rápida el objetivo de la realización de señas, pues los puntajes presentados son muy bajos.

- Exigencia Física: la mayoría de los usuarios concluye en que la exigencia física para realizar las señas es muy poca.

- Exigencia Temporal: al igual que la exigencia mental, los usuarios no presentaron demasiada presión con respecto al tiempo.

- Esfuerzo: la mayoría de los usuarios infiere en que no es necesario realizar un gran esfuerzo para llevar a cabo una seña.

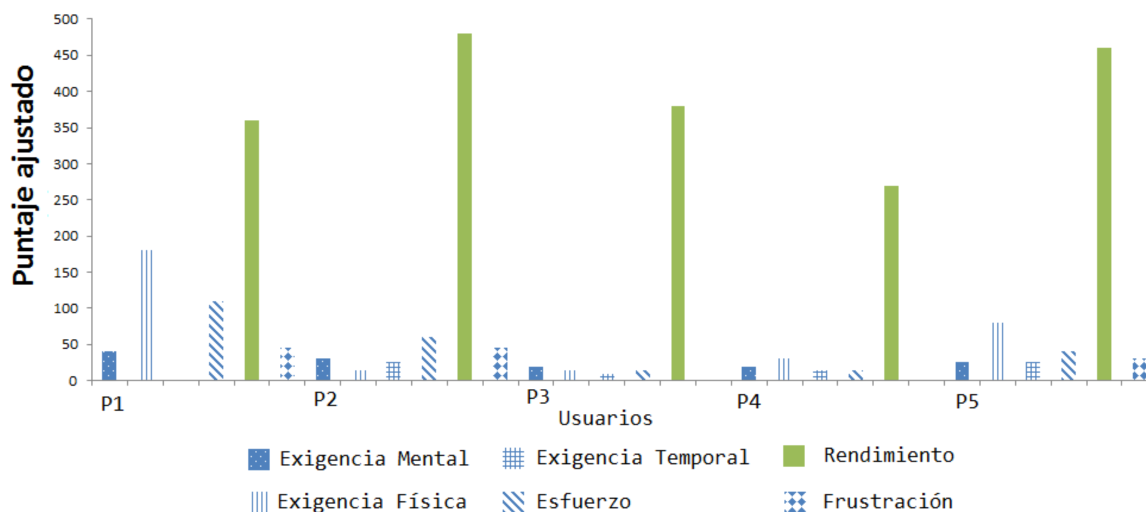


Figura 5.11: Puntaje ajustado de cada subescala para la realización de una seña con una mano del usuario

- Rendimiento: todos los usuarios se sintieron satisfechos con su desempeño al realizar las señas con una sola mano.

- Frustración: Algunos usuarios sintieron frustración un poco elevada debido a que no habían realizado señas o estaban acostumbrados a usar gestos.

En el segundo cuestionario, se evaluó la carga de trabajo para la realización de una seña con ambas manos del usuario. Para el cual se presentan las subescalas utilizadas, que son:

1. Exigencia de tipo mental: ¿cuánta actividad mental y perceptiva fue necesaria para comprender la forma en que se realiza una seña con ambas manos?

2. Exigencia de tipo físico: ¿cuánta actividad física fue necesaria para realizar la seña con ambas manos?

3. Exigencia temporal ¿cuánta presión de tiempo sintió, debido al ritmo al cual se realizaba la seña con ambas manos?

4. Esfuerzo ¿en qué medida ha tenido que trabajar (física o mentalmente) para alcanzar el nivel de desempeño requerido?

5. Rendimiento ¿hasta qué punto cree que ha tenido éxito al crear la seña con ambas manos y cuál es su grado de satisfacción con su desempeño?)

6. Nivel de frustración ¿cuál fue su nivel de frustración al realizar la seña con ambas manos?

El resultado del cuestionario para evaluar las señas realizadas con ambas manos del usuario puede observarse en la Figura 5.8 en la cual se aprecia la cantidad asignada por cada usuario a cada una se las subclases.

Enseguida se muestra el Cuadro 5.15 donde se exponen los datos resultantes de la aplicación del cuestionario, así como el promedio $\overline{x}$ y la desviación estándar $\sigma$ para conocer la dispersión de los datos.

| Usuario | Exigencia Mental | Exigencia Física | Exigencia Temporal | Esfuerzo | Rendimiento | Frustración |
|---|---|---|---|---|---|---|
| 1 | 70 | 180 | 35 | 50 | 280 | 35 |
| 2 | 35 | 35 | 30 | 40 | 360 | 45 |
| 3 | 30 | 15 | 10 | 10 | 380 | 20 |
| 4 | 55 | 30 | 15 | 10 | 270 | 10 |
| 5 | 35 | 120 | 20 | 0 | 450 | 30 |
| $\overline{x}$ | 39 | 76 | 22 | 22 | 348 | 28 |
| $\sigma$ | 16.31 | 63.67 | 9,27 | 16.71 | 66.75 | 12,08 |

Cuadro 5.15: Resultados del cuestionario de carga de trabajo para la realización de una seña con ambas manos del usuario

A continuación se analizan los resultados obtenidos por cada subescala:

- Exigencia Mental: los usuarios entendieron de manera rápida el objetivo de la realización de señas con ambas manos, aunque de manera más lenta comparada con la realización de una seña con una mano.

- Exigencia Física: la mayoría de los usuarios concluye que la exigencia física para realizar la seña con ambas manos es poca, pero superior a realizar una seña con una mano.

- Exigencia Temporal: algunos usuarios aseguraron haber sentido más presión al realizar las señas con ambas manos, debido a tenían que checar que ambas manos estuvieran bien posicionadas arriba del dispositivo y realizar la seña dentro del tiempo. Aún así la cantidad de presión causada por el tiempo tiene valores muy bajos.

- Esfuerzo: la mayoría de los usuarios infiere en que no es necesario realizar un gran esfuerzo para llevar a cabo una seña con ambas manos, pero el esfuerzo realizado para una seña de ambas manos es visiblemente mayor que para realizar una seña de una mano.

- Rendimiento: todos los usuarios se sintieron satisfechos con su desempeño al realizar las señas con ambas manos, pero el rendimiento y el éxito de la realización se ven disminuidos comparados con las señas de una mano.

- Frustración: Algunos usuarios sintieron mayor frustración al realizar señas con ambas manos debido a que ya se habían acostumbrado a realizarlas con una mano y se les dificultaba controlar dos señas al mismo tiempo. Aún así los valores obtenidos no son altos.

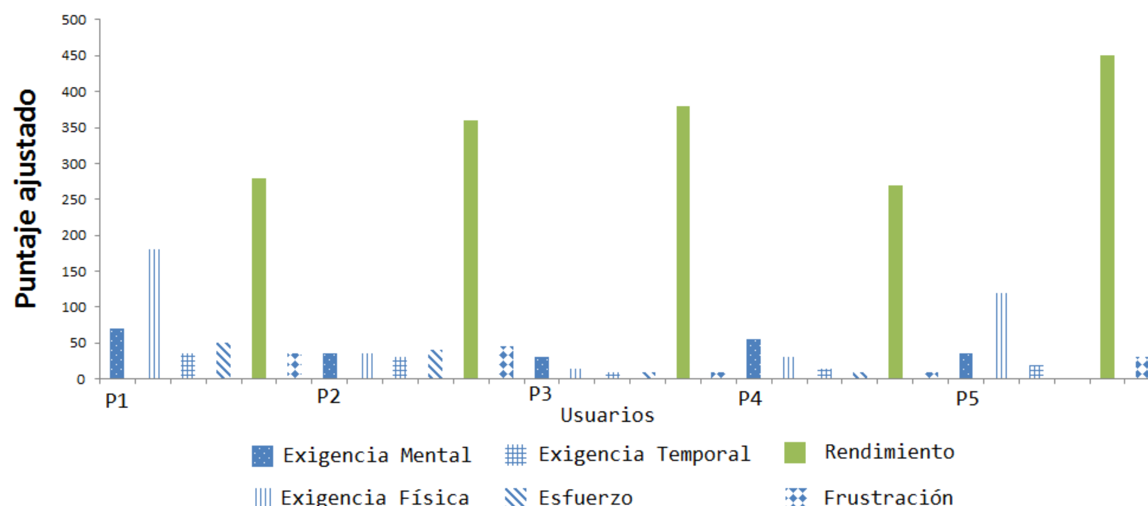


Figura 5.12: Puntaje ajustado de cada subescala para la realización de una seña con ambas manos del usuario

En el tercer cuestionario, se evaluó la carga de trabajo para la interacción de la aplicación mediante el dispositivo Leap Motion (i.e. jugar con la aplicación que se le presenta al usuario mediante el dispositivo Leap Motion). Para el cual se presentan las subescalas utilizadas, que son:

1. Exigencia de tipo mental: ¿cuánta actividad mental y perceptiva fue necesaria para comprender la forma en que se juega?

2. Exigencia de tipo físico: ¿cuánta actividad física fue necesaria para jugar?
3. Exigencia temporal ¿cuánta presión de tiempo sintió, debido al ritmo al cual se jugaba?
4. Esfuerzo ¿en qué medida ha tenido que trabajar (física o mentalmente) para alcanzar el nivel de desempeño requerido?
5. Rendimiento ¿hasta qué punto cree que ha tenido éxito al jugar y cuál es su grado de satisfacción con su desempeño?)
6. Nivel de frustración ¿cuál fue su nivel de frustración al jugar?

El resultado del cuestionario para evaluar la carga mental al jugar se muestra en la Figura 5.9 en la cual se aprecia la cantidad asignada por cada usuario a cada una se las subclases.

A continuación se presenta el Cuadro 5.16 donde se exponen los datos resultantes de la aplicación del cuestionario, así como el promedio $\overline{x}$ y la desviación estándar $\sigma$ para conocer la dispersión de los datos.

| Usuario | Exigencia Mental | Exigencia Física | Exigencia Temporal | Esfuerzo | Rendimiento | Frustración |
|---|---|---|---|---|---|---|
| 1 | 30 | 170 | 60 | 160 | 360 | 55 |
| 2 | 25 | 40 | 55 | 90 | 270 | 30 |
| 3 | 30 | 15 | 10 | 10 | 380 | 10 |
| 4 | 30 | 20 | 10 | 40 | 280 | 25 |
| 5 | 25 | 100 | 25 | 30 | 450 | 20 |
| $\overline{x}$ | 28 | 69 | 32 | 66 | 348 | 28 |
| $\sigma$ | 2,45 | 58,85 | 21,58 | 53,88 | 66,75 | 15,03 |

Cuadro 5.16: Resultados del cuestionario de carga de trabajo experimentada al interactuar con las aplicaciones

Enseguida se analizan los resultados obtenidos por cada subescala:

- Exigencia Mental: los usuarios entendieron de manera rápida el objetivo de la interacción con las diferentes aplicaciones.
- Exigencia Física: la mayoría de los usuarios concluye que la exigencia física para interactuar con las aplicaciones es poca.

- Exigencia Temporal: algunos usuarios concluyeron que la interacción con las aplicaciones requería más tiempo, debido al manejo de las señas, por tanto, algunos usuarios dijeron haber sentido más presión al realizar las señas. Aún así la cantidad de presión causada por el tiempo tiene valores muy bajos.

- Esfuerzo: la mayoría de los usuarios infiere en que no es necesario realizar un gran esfuerzo para llevar a cabo la tarea de interactuar con las aplicaciones y el dispositivo Leap Motion.

- Rendimiento: todos los usuarios se sintieron satisfechos con su desempeño al realizar la interacción, pero el rendimiento y el éxito de la interacción se ven disminuidos debido a que la mayoría sintió más esfuerzo al jugar y combinar las señas.

- Frustración: Algunos usuarios sintieron mayor frustración al interactuar con las aplicaciones, pero los valores obtenidos no son altos.

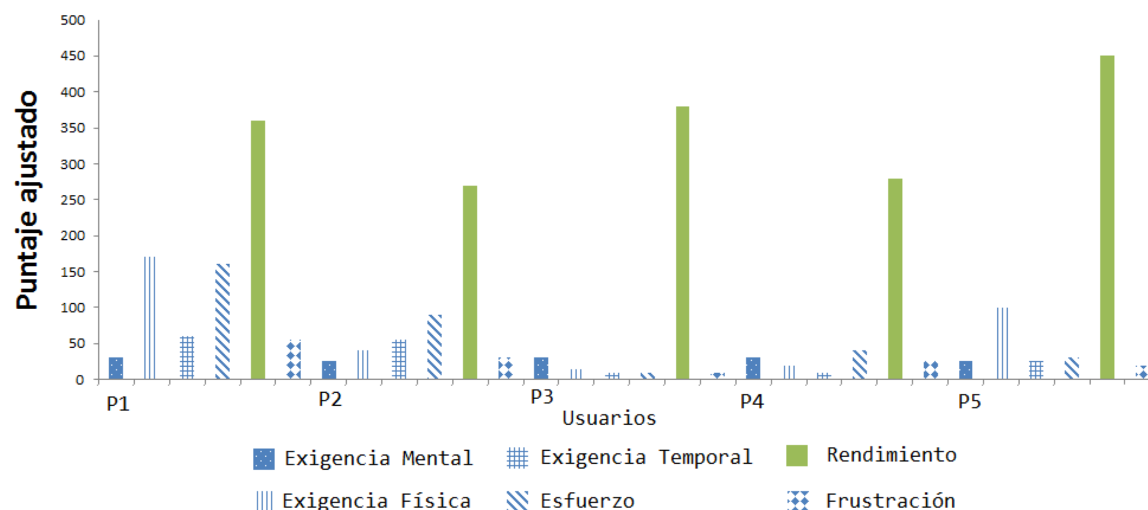


Figura 5.13: Puntaje ajustado de cada subescala para la interacción con las aplicaciones y el dispositivo Leap Motion

# Capítulo 6

# Conclusiones y trabajo a futuro

En este capítulo se presenta una recapitulación del problema (ver Sección 6.1) para presentar las conclusiones del trabajo realizado, así como las contribuciones hechas con el soporte de interacción realizado (ver Sección 6.2). Posteriormente, se exponen las limitaciones del toolkit desarrollado (ver Sección 6.3). Finalmente, se muestran algunos trabajos a futuro, presentados como actualizaciones de funcionalidades del toolkit, así como aplicaciones que pueden realizarse (ver Sección 6.4).

## 6.1. Recapitulación del problema

Como se mencionó en la Sección 1.4, actualmente existen trabajos de investigación que realizan avances en el uso del dispositivo Leap Motion en una computadora con un solo dispositivo para permitir que un usuario interactúe en una aplicación mono-usuario. Es por esta razón que este trabajo de tesis prevé el uso del dispositivo en una computadora para permitir que un usuario interaccione con una aplicación multi-usuario y de esta forma, muchos usuarios concurrirían en la realización de un trabajo (i.e. explorar el uso del dispositivo en entornos multi-usuario).

El problema que se presenta al momento de desarrollar aplicaciones multi-usuario para el dispositivo Leap Motion, es la correcta interacción del usuario con ésta mediante dicho dispositivo. Debido a que no se tiene un control de los usuarios, así como la falta de mecanismos, como señas, que permitan controlar el espacio de trabajo de la aplicación.

## 6.2. Conclusiones y contribuciones

A lo largo de la realización de este trabajo y del desarrollo del *toolkit* se presentaron diversas funciones cuyo propósito es permitir el control de los usuarios para mantener una correcta comunicación entre ellos, otorgar al programador herramientas necesarias para controlar el espacio de trabajo de la aplicación mediante señas y

entregar funciones que faciliten el procesamiento de los datos de los usuarios.

En un principio, se tenía pensado entregarle al programador señas predefinidas, así como la API de Leap Motion proporciona cuatro gestos (uno de los inconvenientes de la API del dispositivo, pues no permite crear gestos propios), pero resultó más interesante el hecho de permitir que el programador realice sus propias señas. Es por eso que las funciones de creación señas entregan el código fuente de dicha seña realizada, para que el programador la utilice a su conveniencia. Por otra parte, las funciones de señas utilizan otras funciones para realizar el trabajo de la interpretación, las cuales pertenecen a la API de procesamiento de dedos y vectores (ver Sección 3.4). Las funciones de procesamiento de dedos y vectores, tienen la finalidad de ahorrar tiempo de escritura de código al programador, como es el caso de $contarDedosExtendidos()$ (ver Sección 3.4.1), que permite conocer el total de dedos extendidos de una o ambas manos del usuario, algo que es muy extenso de codificar con la API de Leap Motion. Otra de las funciones realizadas es $calcularColision()$ (ver Sección 3.4.2), que sirve para calcular si existe o no una colisión dentro de alguna figura u objeto 2D de una aplicación. De igual forma, la función $obtenerCuadranteDedo()$ (ver Sección 3.4.3) posibilita al programador obtener el cuadrante al que apunta a cada dedo del usuario, algo que es esencial al momento de requerir señas con dirección y/o sentido.

En el ámbito de las aplicaciones de prueba, se presentó una aplicación mono-usuario que permite controlar el movimiento y la simulación del clic de un ratón de una manera distinta a como algunos programadores de aplicaciones para el dispositivo Leap Motion lo han resuelto. La manera común en que dichos programadores resuelven el problema de simular un “clic” es mediante uno de los gestos que la API de Leap Motion proporciona denominado $ScreenTap$ (ver Figura 2.6 de la Sección 2.1.2), el cual consiste en presionar con un dedo una pantalla vertical invisible. El problema de usar este gesto para el ratón es que la posición del puntero del mismo está dirigida por la punta del dedo que usa para hacer el gesto de $ScreenTap$, lo cual resulta inconveniente, pues el usuario, de manera natural, mueve puntero con el dedo en posición extendida y al momento de realizar el gesto de clic, el puntero se mueve y no permite dar clic en el área deseada. La solución que presenta en este trabajo es permitir al usuario mover el puntero del ratón mediante el centro de su palma y simular el clic con una seña de puño, así como la liberación del clic cuando el usuario deje de realizar dicha seña. Dando una sensación natural de “agarre”, lo cual es interpretado por el usuario como hacer clic, una seleccionar o como tomar un objeto virtual.

## 6.3. Limitaciones

Una de las principales limitaciones del trabajo es que una computadora no puede procesar el flujo de datos de dos o más dispositivos Leap Motion conectados. Por lo tanto, dos o más usuarios no pueden concurrir en una misma máquina para realizar un trabajo.

## 6.4. Trabajo a futuro

A continuación se presentan algunos de los trabajos a futuro, así como algunas de las aplicaciones que se pueden implementar utilizando las funciones del *toolkit* desarrollado:

- Como se mencionó en la Sección 6.3 de limitaciones, una computadora no puede soportar de dos o más dispositivos Leap Motion. Una de las propuestas para este problema, es la modificación del flujo de datos recibido por los periféricos, en este caso los dispositivos Leap Motion, así como modificar algunas de las rutinas del sistema operativo para distinguir cada flujo de dato.
- Actualización el *toolkit* con funciones predefinidas de señas, tal como el API de Leap Motion entrega al programador los cuatro gestos principales.
- Programación de una aplicación que utilice las señas interpretadas para realizar capturas de pantalla, dibujar círculos o figuras geométricas que indiquen el espacio de trabajo que está utilizando el usuario.
- Realización de un estudio de cuántas señas serían necesarias para controlar un sistema operativo completo.
- Implementación un lenguaje de señas para personas hipoacúsicas, con el código en lenguaje C de cada letra, para utilizarse en aplicaciones que trabajen con personas que tienen este impedimento.
- Aunque el *toolkit* desarrollado utiliza señas, uno de los trabajos a futuro sería la interpretación de gestos y generación de código en lenguaje C. La interpretación de los gestos necesitaría el uso de varios marcos, la duración del gesto realizado, la dirección, el sentido y en algunos casos, un filtro para suavizar el movimiento del gesto.
- Implementación del *toolkit* en otro lenguaje de programación soportado por el API.
- Modificación del *toolkit* para que permita exportar el código de la seña en uno de los lenguajes que utilice la API de Leap Motion, sin importar el código fuente del *toolkit* desarrollado.

# Bibliografía


[1] C. Ellis, S. Gibbs, and G. Rein, Groupware: Some Issues and Experiences, Communication of the ACM, ACM Press, Vol. 34, No.1, pp. 39 - 58, 1991.

[2] A. Prakash, H. S. Shim, and J. H. Lee, Data Management Issues and Trade-offs in CSCW Systems, IEEE Transactions on Knowledge and Data Engineering, Vol. 11, No. 1, pp. 213 - 227, January/February 1999.

[3] John M. Carroll, Encyclopedia chapter on Human Computer Interaction (HCI), 2nd edition, The Interaction Design Foundation, 2014.

[4] Ch. Truong, D. Nguyen-Huynh, M. Tran and A. Duong, Collaborative Smart Virtual Keyboard with Word Predicting Function, HCI International 2013, M. Kurosu (Ed.), Springer, pp. 513522, Las Vegas, NV, USA, julio 21-26, 2013.

[5] Donald A. Norman y Stephen W. Draper, User Centered System Design: New Perspectives on Human-computer Interaction, 1ra edición, CRC Press, 1986, ISBN 0898597811 .

[6] F. Weichert, D. Bachmann, B. Rudak and D. Fisseler, Analysis of the Accuracy and Robustness of the Leap Motion Controller, Sensors, Vol. 13, No. 5, pp. 6380-6393, MDPI, mayo 2013.

[7] Barry W. Boehm, A spiral model of software development and enhancement, In ACM SIGSOFT Software Engineering Notes, Vol. 11, No. 4, pp. 14-24, ACM, 1986.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, 1st edition, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995, ISBN 0-201-63361-2.

[9] J. Dumas, and J. Redish, A practical guide to usability testing, 2nd edition, Intellect Books, USA, 1994, ISBN 1-84150-020-8.

[10] Grudin, Why CSCW Applications Fail: Problems in the Design and Evaluation of Organization of Organizational Interfaces, In Proceedings of the 1988 ACM Conference on Computer Supported Cooperative Work, ACM Press, pp. 85 - 93, Portland OR (USA), 1988.

[11] Lauralee Alben, Quality of experience: defining the criteria for effective interaction design, Interactions, ACM, Vol. 3, No.3, pp. 11-15.

[12] H. Mitake, T. Harano, S. Fujinaga, S. Matsuyama, S. Shibata, M. Ezoe and S. Hasegawa, SprBlender : Creation Environment for Touchable Characters, SIGGRAPH 2014, ACM, Article 41, pp. 1, Vancouver, Canada, agosto 10 14, 2014.

[13] F. Tecchia, L. Alemy , W. Huang, 3D Helping Hands: a Gesture Based MR System for Remote Collaboration, VRCAI 2012, ACM, pp. 323-328, Singapore, diciembre 2 4, 2012.

[14] N. Marquardt, R. Diaz-Marino, S. Boring, S. Greenberg, The Proximity Toolkit: Prototyping Proxemic Interactions in Ubiquitous Computing Ecologies, UIST' 11, ACM, pp. 315-325, Santa Barbara, CA (EE. UU.), octubre 16-19, 2011.

[15] B. Stroustrup, The C++ Programming Language, 4th Edition, Addison-Wesley, ISBN 978-0321563842, May 2013.

[16] A. Genest, C. Gutwin, A. Tang, M. Kalyn, and Z. Ivkovic, KinectArms: A Toolkit for Capturing and Displaying Arm Embodiments in Distributed Tabletop Groupware, CSCW '13, ACM, pp. 157-166, San Antonio, TX, USA, February 2327, 2013

[17] Bekker, Mathilde M. and Olson, Judith S. and Olson, Gary M., Analysis of Gestures in Face-to-face Design Teams Provides Guidance for How to Use Groupware in Design, DIS '95, ACM, pp. 157-166, New York, NY, USA, 1995

[18] J. Li, A. Wessels, L. Alem, C. Stitzlein, Exploring Interface with Representation of Gesture for Remote Collaboration, OZCHI '07, ACM, pp. 179-182, New York, NY, USA, 2007

[19] J. Segen, S. Kumar,Human-computer interaction using gesture recognition and 3D hand tracking.,Image Processing 1998, IEEE, Vol. 3, pp. 188 - 192, Chicago, IL , October 04-07, 1998.