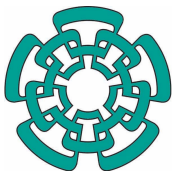


Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

Unidad Zacatenco
Departamento de Computación

Interacción aumentada con un objeto deformable

Tesis que presenta

Michel Torres Alonso

Para obtener el grado de

Maestro en Ciencias en Computación

Director de la Tesis

Dr. Luis Gerardo de la Fraga

Ciudad de México Septiembre, 2019

Resumen

La realidad aumentada es una área interdisciplinaria donde se crean aplicaciones que mezclan la realidad con elementos virtuales, y que buscan encontrar nuevas formas de interacción más intuitivas y sencillas para los usuarios. En esta tesis presentamos un sistema de realidad aumentada donde el usuario puede interactuar tanto con objetos rígidos como con objetos deformables simplemente utilizando sus manos. El sistema en su conjunto conlleva la detección del marcador que forma la referencia para insertar los objetos virtuales, la detección de las manos, la detección de colisiones, la creación de objetos virtuales y la definición de su comportamiento basado en física, y finaLeap Motionente la visualización. La posición de las manos se obtiene mediante el uso del dispositivo Leap Motion. Los objetos virtuales son dibujados encima de un marcador fiducial, que sirve como referencia del mundo real. La detección de colisiones y el comportamiento basado en física de los objetos es proporcionado por la biblioteca Bullet Physics. La visualización de los objetos virtuales se logra a través de una aplicación, donde se muestran imágenes de la escena real con la cámara Playstation Eye, la cual se ejecuta a una frecuencia de 60 Hz. Se realizaron algunas pruebas al sistema utilizando un cubo como el objeto rígido y una esfera formada por triángulos como el objeto deformable, con los cuales se probaron distintas formas para que el usuario pudiera manipularlos de manera intuitiva, con las cuales exponemos las ventajas, desventajas y limitaciones del sistema y de los elementos que lo componen. Ventajas como que el sistema es sencillo de montar y su uso es muy intuitivo gracias al uso del Leap Motion. Desventajas como que el sistema ofrece una perspectiva fija que limita la visión del usuario y el hecho que el usuario tenga que mantener su mirada en un monitor en lugar de en la escena real. Por último, limitaciones del sistema como que solo se modelan la punta de los dedos índice y pulgar de la mano derecha y el tamaño de la escena está limitado por el rango de visión del Leap Motion y de la cámara.

Abstract

Augmented reality is an interdisciplinary area where the applications are developed by mixing reality with virtual elements. This applications are always looking for new easy and intuitive ways for user interaction. We present an augmented reality system in which the user can interact in real time with solid and deformable objects. The system have to resolve several tasks in real time: marker detection, hands detection, collision detection, virtual object creation and the object's behavior based in physics, and finally the object visualization. The fingers positions are tracked by the Leap Motion device. The virtual objects are draw with respect to a fiducial marker. The collision detection and the object's behavior based in physics are given by the Bullet Physics library. The object visualization is achieved through a PC application, showing us video standard taking from the Playstation Eye camera, that run at 60 Hz. We proved several ways to interact with objects using a cube as rigid object and a sphere of triangles as deformable object, then we present the system's advantages, disadvantages and limitation. Advantages such as the system is simple to assemble and intuitive to use. Disadvantages such as the user has a limit vision by the fix camera and the fact that the user have to watch the monitor instead of the real scene. Finally, some system's limitations such as the user can use only two fingers of the right hand and not the whole hand and the size of the scene limited by the range of Leap Motion and the camera.

Agradecimientos

Agradezco al Centro de Investigación y de Estudios Avanzados (CINVESTAV), en particular al Departamento de Computación, por la oportunidad que me dió al aceptarme en su programa de maestría en ciencias en computación.

Agradezco al Consejo Nacional de Ciencia y Tecnología (CONACYT) por la beca económica que me proporcionó durante estos dos años de maestría.

Agradezco a mi asesor el Dr. Luis Gerardo de la Fraga por el tiempo que dedicó a mi tesis y por el conocimiento que compartió conmigo.

Agradezco a mis revisores el Dr. Cuauhtemoc Mancillas y la Dra. Sonia por su tiempo y por sus comentarios que han mejorado mi trabajo.

Agradezco a mis padres el Sr. Manuel Torres y la Sra. Andrea Alonso por el apoyo incondicional que me han dado siempre a lo largo de vi mida, en especial en este periodo en el que decidí cursar una maestría. Al igual que agradezco a mis hermanos Sandra, Manuel, Daniel y Leticia por motivarme y apoyarme en todo momento. Por último, dedico este trabajo a mis sobrinos Gael, Leonel, Jacqueline, Valeria y Sebastian por alegrar cada uno de mis días.

Agradezco a mis compañeros de la maestría por el apoyo que me dieron durante la maestría y por los buenos momentos que compartimos juntos. En especial a mis amigos Fernanda, Isaac y Amín.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

La realidad aumentada (RA) consiste en un conjunto de tecnologías y técnicas que permiten la visualización de elementos virtuales, generados por computadora, superpuestos sobre el mundo real. Los elementos, a pesar de ser virtuales, son integrados en el ambiente de forma que el usuario los perciba como cualquier otro objeto real. La RA representa un poderoso paradigma en la interacción humano-computadora, proporcionando al usuario una interface más cercana a la realidad física. Para entender mejor la RA debemos tener en cuenta que existen otro tipo de técnicas, que no deben ser confundidas con ésta, como lo es la realidad virtual (RV).

El concepto de continuo de la virtualidad, introducido por Paul Milgram y Fumio [22] y mostrado en la figura 1.1, describe una escala continua donde se puede distinguir entre lo que es completamente virtual y la realidad. Dentro de este concepto, la realidad está ubicada en uno de los extremos, mientras que al otro extremo se encuentra la RV. Entre estos dos conceptos tenemos la RA la cual define cualquier caso en el que un ambiente real es 'aumentado' o complementado con elementos virtuales, siempre tratando que los elementos virtuales tengan un comportamiento muy similares a los reales y se vean como parte del mundo real. Por lo tanto, la RA es un concepto que está más cerca de la realidad.

1.1. Motivación

Dentro de los sistemas de RA se desea que el usuario tenga una experiencia lo más realista posible, intentando imitar el comportamiento de los objetos y la forma en la que el usuario interactúa con ellos. Por ello, las nuevas interfaces de interacción exploran nuevas tecnologías y técnicas que permitan acercar al usuario a una interacción 'natural'.

En los últimos años, se han introducido algunas tecnologías con un costo accesible que pueden ayudar a mejorar la interacción con elementos virtuales; sensores de profundidad como Microsoft Kinect [6] y dispositivos de seguimiento del movimiento de las manos como el Leap

Figura 1.1: El concepto continuo de la virtualidad [22]

Motion(LM) [20] han comenzado a ser utilizados para explorar nuevas técnicas para lograr esto. En el caso del LM, existen trabajos que han explorado el uso del dispositivo como medio de entrada para detectar los gestos del usuario [18,31,32]. Ninguna de las técnicas hasta el momento contempla la respuesta que tendrían los objetos deformables con el uso de este dispositivo. Además, hay un número reducido de gestos implementados utilizando la información del LM, por lo que aún queda trabajo por hacer para mejorar la interacción del usuario en un ambiente de RA con el uso del LM.

1.2. Planteamiento del problema

En la interacción con objetos virtuales mediante el uso de las manos se tienen varios problemas, siendo el principal conocer la posición y orientación de las manos en tiempo real, además de la definición de distintos gestos para interactuar con el objeto los cuales generarán respuestas distintas al momento de interactuar. También, es importante conocer el momento en el que la interacción ocurre, es decir, tenemos que detectar la colisión de los objetos virtuales con las manos mediante un algoritmo de detección de colisiones. Además, se debe dotar a los objetos virtuales con un comportamiento similar a los reales, principalmente los objetos deben ser afectados por la fuerza de gravedad y reaccionar a la colisión con otros objetos.

El comportamiento de los objetos deformables es un tema complejo, dado que estos son modelados usando resortes, que son representados en una malla con la forma del objeto que se desea modelar. La aplicación de una fuerza sobre cualquier resorte se propaga hacia los demás resortes provocando la deformación del objeto. Esto necesita de la resolución de un conjunto de ecuaciones que deben ser resueltas en tiempo real si se desean implementar en un sistema de RA.

1.2.1. Estado actual del problema

En la actualidad existe un gran número de sistemas de RA con distintas aplicaciones en entrenamiento médico [29, 31], en tratamientos médicos [9] y en entretenimiento [8, 23]. Cada uno de ellos presenta diferentes formas de interacción entre el usuario y los objetos virtuales, dependiendo del tipo de tecnología que utilicen. Algunos de estos sistemas abordan el problema de obtener la posición y orientación de las manos utilizando marcadores de RA. Volkert Buchmann *et al.* presentan una técnica de interacción basada en la detección de las yemas de los dedos para ambientes de RA [7]. Esta técnica hace uso de un guante con marcadores que el usuario debe utilizar para interactuar dentro del sistema. Esto permite la detección de los gestos del usuario, mediante la detección de los marcadores. En la figura 1.2, se muestra la forma de ejecutar el gesto para presionar un botón.

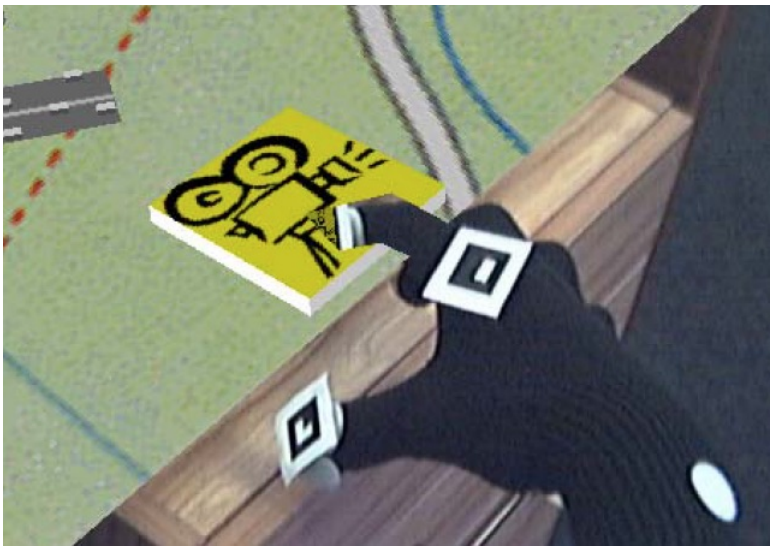


Figura 1.2: FingARtips: gesto para presionar un botón [7].

Hoy en día se busca que la interacción en un ambiente de RA sea más real. Los sensores de profundidad y sensores de movimiento han ayudado a crear técnicas que permitan al usuario tener mayor libertad dentro del ambiente de RA, y por ende ha acercado al usuario a una experiencia más cercana a la realidad. Tales son los casos del dispositivo Microsoft Kinect y del LM, los cuales han sido utilizados en la creación de sistemas de RA que pretenden mejorar la interacción con el usuario.

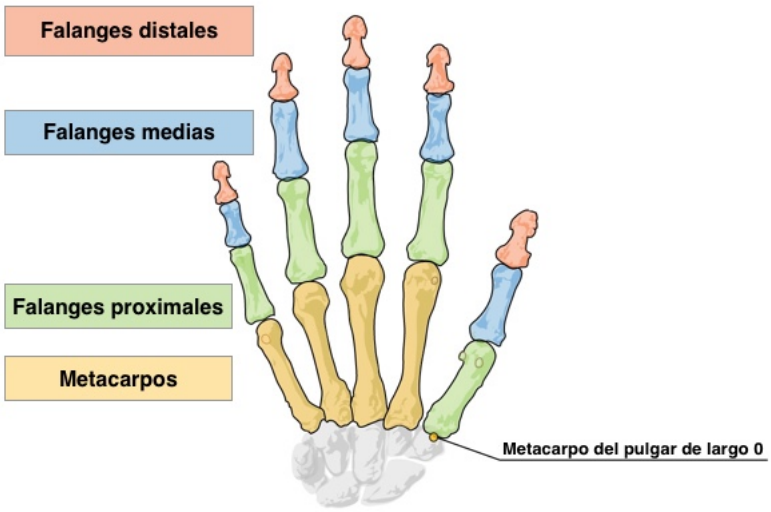
Trabajos como el de Sam Corbett-Davies *et al.* utilizan el dispositivo Microsoft Kinect [9]. Ellos proponen una nueva técnica de interacción para sistemas de RA y presentan una aplicación para el tratamiento contra la aracnofobia, la cual se muestra en la figura 1.3. Mediante el uso de los datos que proporciona el dispositivo Kinect detectan y modelan objetos del mundo real en un ambiente de RA. Los objetos reales son detectados en 3D utilizando el algoritmo IPC (Interactivo del Punto más Cercano), un algoritmo comúnmente usado para reconstruir superficies 2D y 3D. Los objetos reales no necesitan ser marcados de una manera en particular, por lo que facilita la interacción natural con los objetos. Además, existen pocas restricciones en cuanto al tipo de

objetos que se pueden usar. Para demostrar la forma en que la técnica funciona se propone un sistema de RA como tratamiento a la aracnofobia, en donde las arañas son elementos virtuales que pueden caminar sobre, alrededor y debajo de objetos reales. También, pueden ser sujetadas, transportadas y ocluidas por el usuario. Este tipo de técnicas ofrecen la facilidad de interactuar con objetos sin el uso de dispositivos invasivos como el guante con marcadores del sistema de la figura 1.2.

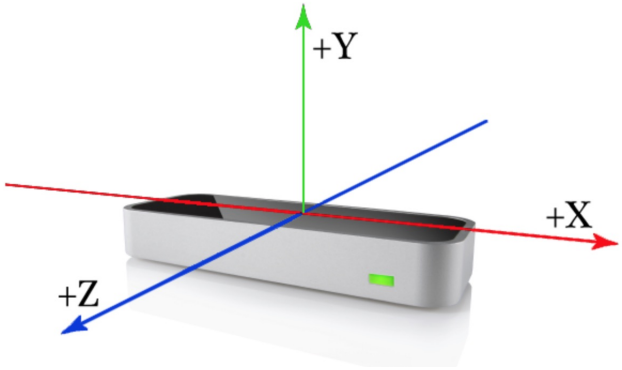
Figura 1.3: Sistema de RA para el tratamiento contra la aracnofobia utilizando Kinect [9].

En los últimos años, el uso del dispositivo Leap Motion ha sido recurrente en los sistemas de RA y RV. Este dispositivo hace el seguimiento del movimiento de las manos, ofreciendo información bastante completa sobre su posición y orientación. El dispositivo nos entrega la posición del inicio y fin de cada uno de los huesos que componen la mano humana. En la figura 1.4(a), se puede ver la mano humana y los huesos que la componen. Estas posiciones son coordenadas del sistema de coordenadas que define el propio dispositivo, mostrado en la figura 1.4(b). El hecho de que el LM nos proporcione la información de la posición de cada hueso de la mano nos permite saber cuando alguno de los dedos está flexionado. Por lo tanto, el número de gestos que el dispositivo reconoce es bastante amplio. Con esta información se puede generar una imagen virtual de ambas manos, que puede servir como entrada en los sistemas de RA.

Valentini presenta una metodología de RA basado en el uso del LM [32]. Su sistema permite al usuario visualizar la escena a través de unas gafas de RV y se puede interactuar con los objetos virtuales simplemente utilizando las manos, como se ve en la figura 1.5. Este sistema solo implementa la interacción con objetos sólidos y puede detectar 3 gestos del usuario: uno usado para sujetar objetos esféricos, uno para sujetar objetos cilíndricos y un último para sujetarlos usando sólo el dedo índice y pulgar.



(a) Modelo del esqueleto de la mano humana, con los huesos que la componen y sus nombres [20].



(b) Sistema de coordenadas del LM [20]

Figura 1.4: Información que ofrece el dispositivo LM y su sistema de coordenadas.

1.2.2. Solución propuesta

Para crear el sistema de RA que permita al usuario interactuar con objetos deformables, usamos el dispositivo LM para obtener el seguimiento del movimiento de las manos. Se utilizó un motor de simulación basado en física, llamado Bullet Physics [10], para proporcionar a los objetos virtuales un comportamiento similar a los reales. Además, este también proporciona los algoritmos para detectar colisiones entre los objetos que conforman la escena, incluidas las manos del usuario. Los objetos deformables son representados por cuerpos blandos proporcionados por Bullet Physics, utilizando una forma simple como una esfera como modelo de prueba.

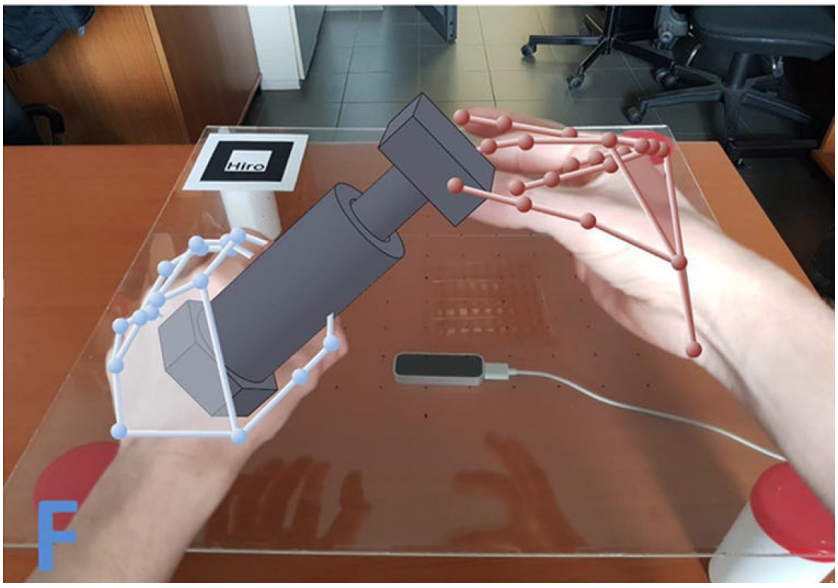


Figura 1.5: Interfaz natural para montaje virtual interactivo en RA utilizando el dispositivo LM [32].

1.2.2.1. Configuración del sistema

El sistema se compone principalmente de la cámara Playstation Eye y el dispositivo LM. El dispositivo LM es una pequeña caja de $8cm \times 3cm \times 1.1cm$, que se conecta por medio del puerto USB a una computadora. El dispositivo cuenta con 3 LEDS infrarrojos y dos cámaras. Su principio de seguimiento se basa en la estereoscopía. Esto es que la luz reflejada desde los LEDs se ve desde dos puntos de vista diferentes y la distancia a la que se encuentran las manos del sensor se calcula de acuerdo con estos datos.

LM tiene un campo de visión de 150°, el cual tiene la forma de una pirámide invertida, como se ve en la figura 1.6. El dispositivo tiene la capacidad de trabajar a frecuencias entre 20 Hz y 200 Hz, dependiendo de la configuración y del poder de computo disponible.

Se utiliza un marcador de RA para obtener la referencia del plano sobre el cual se presentan los elementos virtuales al usuario. El marcador se encuentra sobre una superficie de madera junto con el dispositivo LM, para mantener la distancia entre ambos fija. Esto ayudará a relacionar los sistemas de coordenadas de ambos elementos y hacer que se encuentren en un mismo marco de referencia.

La cámara del Playstation Eye es utilizada para tomar imágenes y poder procesarlas para detectar el marcador de RA. Esto nos permite presentar los elementos virtuales en la escena y que sean visibles para el usuario. La cámara tiene la capacidad de capturar video estándar a una velocidad de 60 Hz en resolución 640×480 px y a 120 Hz en resolución 320×240. El sistema completo puede verse en la figura 1.7.

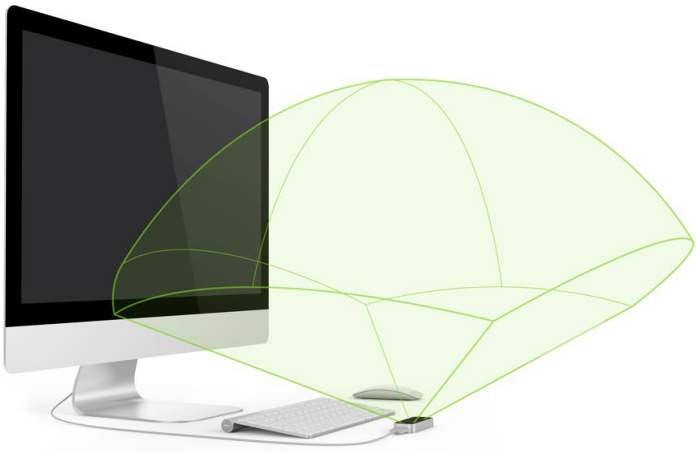


Figura 1.6: Campo de visión de LM.

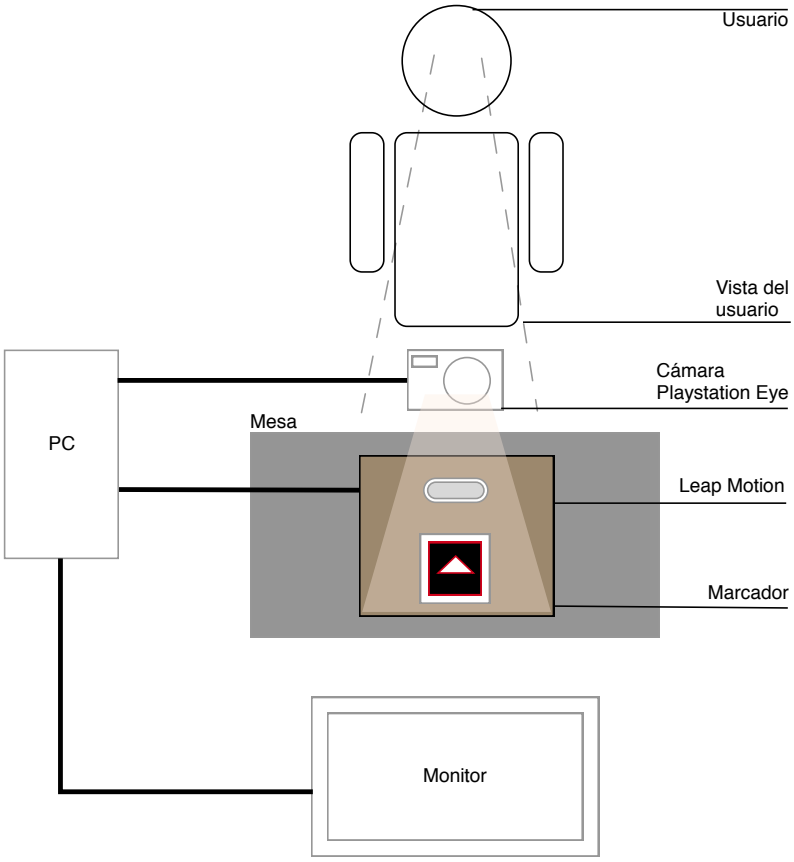


Figura 1.7: Configuración del sistema de RA.

1.3. Contribuciones

- Sistema de RA que permita al usuario interactuar con objetos deformables.
- Estudio y prueba de la biblioteca Bullet Physics para conocer su alcance como un motor de detección de colisiones.
- Prueba de la biblioteca Bullet Physics para conocer su implementación de objetos deformables y cómo interactuar con ellos.
- Uso de los marcadores basados en tipo de orden [12, 13] para construir un sistema de RA.

1.4. Organización de la Tesis

Esta tesis consta de 6 capítulos. Este primer capítulo nos presenta la introducción de la tesis. En ésta se presentan antecedentes básicos sobre la RA, se plantean el problema de la tesis y la motivación de la misma. Finalmente, se da un esbozo de la solución propuesta y listado las contribuciones de este trabajo.

El capítulo dos presenta los conceptos necesarios para entender el funcionamiento del sistema. Conceptos como el modelo de la cámara obscura, que representa la relación que existe entre una escena 3D y una fotografía tomada por la cámara. La calibración de la cámara el cual es un proceso que nos permite conocer los parámetros de la cámara y con ellos eliminar la distorsión de las fotografías que se capturan. La detección del marcador que es un proceso muy importante para poder presentarle los objetos virtuales al usuario. Se presentan los algoritmos para la detección de colisiones y su funcionamiento que nos permite lograr la interacción del usuario con el sistema. Finalmente, se presenta la forma en que se modelan los objetos deformables y como son implementados por la biblioteca Bullet Physics.

El capítulo tres presenta todos los detalles acerca del diseño del sistema de RA. Se introducen cada uno de los elementos de hardware que componen el sistema y se especifica el rol que cumplen dentro del mismo. También, se describen las herramientas de software utilizadas para el desarrollo del sistema.

El capítulo cuatro presenta la implementación del sistema. En este capítulo se muestra como se resolvieron las diversas tareas que el sistema debe desempeñar para lograr la interacción. Se describe como se utilizan los datos del dispositivo LM y cómo se procesa la información para que exista una respuesta por parte de los objetos al entrar en contacto con el usuario.

El capítulo cinco presenta las pruebas aplicadas al sistema y sus resultados.

Finalmente, el capítulo seis presenta la conclusiones obtenidas del desarrollo de esta tesis.

Presentando también los trabajos futuros que surgen del desarrollo de esta tesis.

Capítulo 2

Marco teórico

En este capítulo se revisarán los conceptos necesarios para entender cómo se desarrolló el sistema de RA propuesto. Dentro de estos conceptos se encuentran el modelo de la cámara obscura, el cálculo de la homografía y la calibración de la cámara. Estos conceptos nos ayudarán a comprender cómo se realiza la detección del marcador, cómo obtener la transformación del modelo del marcador para visualizarlo como se ve en la imagen y cómo usar esta información para visualizar los objetos virtuales en la escena real del sistema de RA.

Además, se presentan los objetos deformables, describiendo la forma en la que son modelados y cómo se consigue el comportamiento característico de este tipo de objetos. Por último, hablaremos del detector de colisiones, que es fundamental para lograr la interacción del usuario con los objetos virtuales.

2.1. Modelo de la cámara obscura

El modelo de una cámara es la expresión matemática que relaciona los puntos tridimensionales de la escena, que ve la cámara, a los puntos bidimensionales en la imagen, que es la fotografía que toma la cámara [15, 17]. Se usó el modelo más simple, que se conoce como modelo de la cámara obscura. Este modelo no usa lentes.

Considerando la proyección central de un conjunto de puntos en un plano sobre el espacio. Siendo el centro de proyección el mismo origen del sistema de coordenadas euclidiano y considerando el plano $z = -f$, llamado el plano de proyección. Dentro del modelo de la cámara obscura, un punto tridimensional $\mathbf{X} = [x, y, z]^{\mathrm{T}}$ es asignado al punto en el plano de proyección donde una línea que une el punto $\mathbf{X}$ con el centro de proyección intersecta el plano de proyección [17]. Esto se puede apreciar en la figura 2.1.

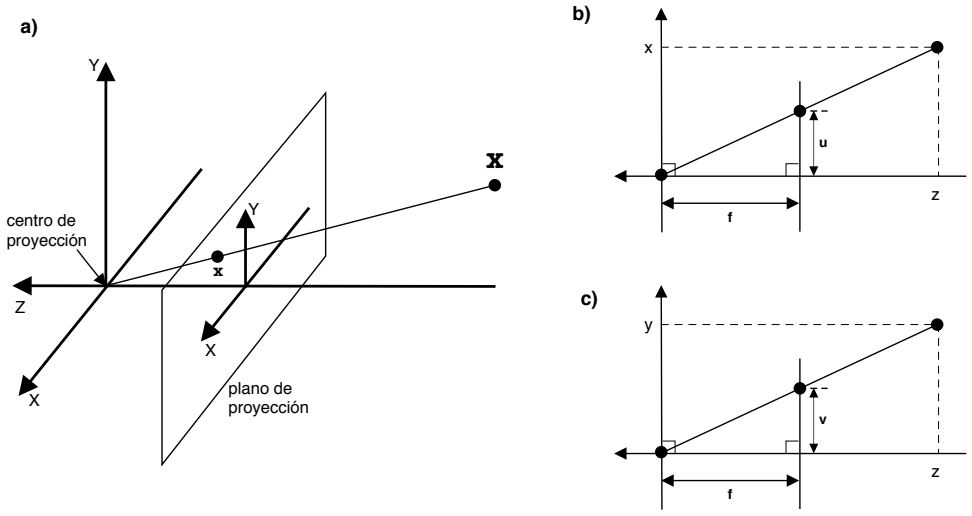


Figura 2.1: El sistema de coordenadas para deducir las ecuaciones de la proyección en perspectiva que definen al modelo de la cámara obscura. El punto 3D $\mathbf{X}$ se proyecta en el plano de proyección como el punto $\mathbf{x}$ en 2D. [17].

De la figura 2.1(b), por triángulos semejantes se tiene:

$$\begin{aligned} \frac{u}{-f} &= \frac{x}{z}, \\ u &= \frac{-fx}{z}, \text{ para } -f < 0, \text{ y } z < 0. \end{aligned} \tag{2.1}$$

De la figura 2.1(c), también por triángulos semejantes se tiene:

$$\begin{aligned} \frac{v}{-f} &= \frac{y}{z}, \\ v &= \frac{-fy}{z}, \text{ para } -f < 0, \text{ y } z < 0. \end{aligned} \tag{2.2}$$

La restricción $-f < 0$ en las ecuaciones (2.1) y (2.2) significa que el plano de proyección debe estar adelante de la cámara y con $z < 0$ significa que necesariamente los puntos tridimensionales también deben de estar adelante de la cámara. El punto $[u, v]^{\mathrm{T}}$ es el punto proyectado sobre la imagen.

Utilizando coordenadas homogéneas, las ecuaciones (2.1) y (2.2) pueden representarse en una sola matriz como:

$$\lambda \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} -fx \\ -fy \\ z \end{bmatrix} = \begin{bmatrix} -f & 0 & 0 & 0 \\ 0 & -f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}. \tag{2.3}$$

Abusando de la notación, se usará $\mathbf{x} = [u, v, 1]^{\mathrm{T}}$ para representar también un punto bidimensional homogéneo, y $\mathbf{X}$ para representar un punto tridimensional homogéneo. Así, el modelo de

la cámara obscura en (2.3) puede representarse como:

$$\lambda \mathbf{x} = P\mathbf{X} = K[I|0]\mathbf{X}, \tag{2.4}$$

donde la matriz K es la matriz que se conoce como *matriz de parámetros intrínsecos* de la cámara o también como la *matriz de calibracion* de la cámara.

El origen de las coordenadas en la imagen se le conoce como *punto principal*. Si representamos el punto principal como (u_0, v_0), la matriz K será ahora:

$$K = \begin{bmatrix} -f & 0 & u_0 \\ 0 & -f & v_0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{2.5}$$

Finalmente, si el cuerpo tridimensional $\mathbf{X}$ es rotado por una matriz de rotación R y translado según el vector $\mathbf{t}$, la ecuación (2.4) queda como:

$$\lambda \mathbf{x} = K[R|\mathbf{t}]\mathbf{X}. \tag{2.6}$$

Una forma alternativa, tal vez más intuitiva de usar, a la ecuación (2.6) se logra extrayendo R de la matriz aumentada $[R|\mathbf{t}]$ así:

$$\begin{aligned} \lambda \mathbf{x} &= KR[I|R^{\mathrm{T}}\mathbf{t}]\mathbf{X}, \\ \lambda \mathbf{x} &= KR[I|-\mathbf{c}]\mathbf{X}, \end{aligned} \tag{2.7}$$

donde el vector $\mathbf{c}$ representa la posición de la cámara, que es igual a $\mathbf{c} = -R^{\mathrm{T}}\mathbf{t}$. Nótese que en este modelo en (2.7) el cuerpo tridimensional $\mathbf{X}$ primero es transladado según $-\mathbf{c}$ y luego rotado por R y proyectado según la matriz K, para una cámara que está en el origen de coordenadas viendo hacia el eje z negativo.

La matriz R en (2.7) puede obtenerse a partir de la posición de la cámara $\mathbf{c}$, que apunta al origen de coordenadas y con un vector $\mathbf{a}$ que indica la dirección hacia arriba para la cámara. De esta forma se tiene:

$$R = \begin{bmatrix} \mathbf{x}^{\mathrm{T}} \\ \mathbf{y}^{\mathrm{T}} \\ \mathbf{z}^{\mathrm{T}} \end{bmatrix}, \tag{2.8}$$

donde $\mathbf{z} = \frac{c}{\|c\|}$, $\mathbf{x} = \frac{\mathbf{a}\times\mathbf{z}}{\|\mathbf{a}\times\mathbf{z}\|}$, y $\mathbf{y} = \mathbf{z} \times \mathbf{x}$.

2.1.1. Calibración de la cámara

La calibración de la cámara es un proceso el cual nos permite conocer los parámetros intrínsecos y extrínsecos de la cámara. En la práctica, el mundo y el sistema de coordenadas de la cámara

están relacionados por un conjunto de parámetros físicos, como la distancia focal, el tamaño de los píxeles, la posición del punto principal y la posición y orientación de la cámara. Nosotros distinguiremos los parámetros intrínsecos, que relacionan el sistema de coordenadas de la cámara con un sistema de coordenadas ideal, de los parámetros extrínsecos, que relacionan el sistema de coordenadas de la cámara con un sistema de coordenadas fijo del mundo y especifica su posición y orientación en el espacio [15]. El cálculo de los parámetros intrínsecos es un proceso importante para el sistema de RA, nos entrega la matriz que realiza la transformación en perspectiva de la cámara y se podría eliminar la distorsión del lente que usa la cámara.

La cámara que utilizamos la cámara Playstation Eye, que se muestra en la figura 2.2. Esta cámara es capaz de capturar video estándar con frecuencias de 60 Hz en una resolución de 640×480 píxeles y 120 Hz en 320×240 píxeles.

Figura 2.2: Cámara Playstation Eye.

Nosotros hicimos uso de una aplicación de OpenCV llamada *Interactive camera calibration application* [4] para realizar este proceso. Se hizo uso de un tablero con marcadores chAruco como el de la figura 2.3.

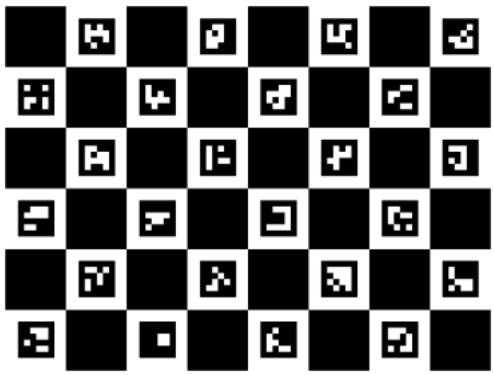


Figura 2.3: Tablero con marcadores chAruco utilizado para la calibración de la cámara [4].

El proceso es bastante simple, solo debe ejecutarse el comando en la terminal de la aplicación y escribir los parámetros que la aplicación necesita sobre el tablero. Debemos indicarle el tamaño

del tablero, el tamaño de un lado de cualquier cuadrado negro del tablero y el tipo de marcadores utilizados.

```
opencv_interactive-calibration -h=8 -w=6 --sz=.016 --t=charuco
```

Debemos colocar el tablero enfrente de la cámara donde sea completamente visible. Mientras la aplicación está en ejecución debemos mover el tablero y colocarlo en distintas posiciones tratando de cubrir el espacio que ve la cámara. El programa tomará distintas imágenes y hará el cálculo de los parámetros. Hasta que finalmente, el programa despliegue el mensaje de que la calibración ha terminado. Así podremos guardar los parámetros calculados en un archivo. Y con ello habremos concluido el proceso de calibración de la cámara.

Del archivo D.1 del apéndice D el parámetro `cameraMatrix` representa la matriz de calibración de la cámara, de dónde podemos obtener los valores $f = 548.22$, $u_0 = 333.26$ y $v_0 = 257.91$. Por lo que, la matriz de calibración de la cámara es:

$$K = \begin{bmatrix} -548.22 & 0 & 333.26 \\ 0 & -548.22 & 257.91 \\ 0 & 0 & 1 \end{bmatrix}. \tag{2.9}$$

Para corregir la distorsión de la lente, OpenCV toma en cuenta sus factores radiales y tangenciales [2]. Para los factores radiales se utiliza las siguientes expresiones:

$$\begin{aligned} x_{corregida} &= x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \\ y_{corregida} &= y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \end{aligned} \tag{2.10}$$

Para un punto $[x, y]^{\mathrm{T}}$ en la imagen de entrada tenemos que su posición corregida en la imagen de salida será $[x_{corregida}, y_{corregida}]^{\mathrm{T}}$. La distorsión radial se presenta en forma de 'efecto de pez' en la imagen capturada por la cámara. En el caso de la distorsión tangencial, ésta se presenta porque la lente de la cámara no es perfectamente perpendicular al plano de imagen [2]. Esta distorsión puede ser corregida utilizando las expresiones:

$$\begin{aligned} x_{corregida} &= x + [2p_1 xy + p_2(r^2 + 2x^2)] \\ y_{corregida} &= y + [p_1(r^2 + 2y^2) + 2p_2 xy] \end{aligned} \tag{2.11}$$

En el archivo D.1 del apéndice D, el parámetro `dist_coeffs` contiene la matriz de coeficientes de distorsión. Esta matriz tiene una sola fila con 5 columnas y está definida como se muestra en la expresión (2.12). Con los coeficientes podemos aplicar la corrección de las distorsiones radiales y tangenciales utilizando las expresiones (2.10) y (2.11), respectivamente.

$$\text{Coeficientes de distorsión} = \begin{bmatrix} k_1 & k_2 & p_1 & p_2 & k_3 \end{bmatrix} \tag{2.12}$$

Tomando los valores del archivo de parámetros tenemos que la matriz de coeficientes de distorsión es:

$$\text{Coeficientes de distorsión} = \begin{bmatrix} -0.1247 & 0.2442 & 0 & 0 & -0.0681 \end{bmatrix}$$

Por último, el archivo de parámetros nos indica el error de reproyección promedio `avg_reprojection_error`. El error de reproyección es una medida geométrica que nos proporciona el error en una imagen entre la distancia de un punto proyectado y uno medido.

2.1.2. Cálculo de una homografía

En el desarrollo del sistema es necesario estimar la posición de ciertos elementos en las imágenes que se van a procesar. En especifico nos interesa estimar la posición del marcador dentro del sistema. La homografía es una transformación que relaciona dos planos y se representa con una matriz no singular de tamaño 3×3 [17].

Suponiendo que $\mathbf{x}$ son los puntos de una imagen y $\mathbf{X}$ son los puntos de la escena. Considerando que la escena está sobre un plano, entonces podemos relacionarlos por medio de una homografía.

$$\lambda \mathbf{x} = H\mathbf{X},$$

donde H es una matriz de homografía de la forma:

$$H = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix}.$$

La homografía tiene 8 grados de libertad y no 9 debido al factor de escala λ. Podemos fijar el valor de $h_{33} = 1$. Así que para calcular la homografía tenemos un sistema de ecuaciones:

$$\lambda \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}.$$

$$\begin{aligned} \lambda u &= h_{11}x + h_{12}y + h_{13} \\ \lambda v &= h_{21}x + h_{22}y + h_{23} \\ \lambda &= h_{31}x + h_{32}y + 1. \end{aligned}$$

Sustituyendo la última ecuación en las dos primeras tenemos que

$$u = h_{11}x + h_{12}y + h_{13} - h_{31}ux - h_{32}uy,$$
$$v = h_{21}x + h_{22}y + h_{23} - h_{31}vx - h_{32}vy.$$

Con esto obtenemos que una correspondencia de puntos $\mathbf{x}$ y $\mathbf{X}$ nos genera dos ecuaciones, y necesitamos al menos cuatro correspondencias de puntos para generar las ocho ecuaciones necesarias para poder encontrar las ocho incógnitas. Así que para poder calcular la homografía podemos generar un sistema de ecuaciones de la forma:

$$A\mathbf{h} = \mathbf{b}.$$

Para calcular la homografía se tiene un sistema de ecuaciones de tamaño $2n \times 8$, para n correspondencias de puntos.

$$A \in \mathbb{R}^{2n\times 8}, \text{para } n \text{ correspondencias.}$$

Una forma simple de calcular $\mathbf{h}$ es utilizando la descomposición QR.

$$A = QR,$$

donde Q es una matriz ortogonal y R es una matriz triangular superior

Aplicando esta idea tenemos que

$$\begin{aligned} A\mathbf{h} &= \mathbf{b} \\ (QR)\mathbf{h} &= \mathbf{b} \\ (QR)^T(QR)\mathbf{h} &= (QR)^T\mathbf{b} \\ R^TQ^TQR\mathbf{h} &= R^TQ^T\mathbf{b} \\ R\mathbf{h} &= Q^T\mathbf{b}. \end{aligned}$$

Considerando la matriz aumentada B:

$$B = [A \mid \mathbf{b}].$$

Y calculando su descomposición QR.

$$[QR \mid \mathbf{b}] = Q[R \mid Q^T\mathbf{b}].$$

Encontramos que para obtener la homografía no tenemos que calcular $Q^{\mathrm{T}}\mathbf{b}$, sino construir la matriz aumentada $[A|\mathbf{b}]$ y calcular la descomposición QR de esta matriz aumentada. En la última columna aparecerá el resultado de $Q^{\mathrm{T}}\mathbf{b}$. Aquí finalmente se tiene que aplicar una retrosustitución para obtener los valores de las incógnitas.

Hay que considerar que calcular la homografía a partir directamente de los puntos es inestable numéricamente. Por lo tanto, se deben de transformar los puntos a una media cero y varianza 1 antes de realizar el cálculo de la homografía de la forma siguiente:

$$\lambda T_1\mathbf{x} = H'T_2\mathbf{X}, \tag{2.13}$$

donde T_1 y T_2 son transformaciones de la forma:

$$T = \begin{bmatrix} \frac{1}{\sigma_x} & 0 & -\frac{\bar{x}}{\sigma_x} \\ 0 & \frac{1}{\sigma_y} & -\frac{\bar{y}}{\sigma_y} \\ 0 & 0 & 1 \end{bmatrix}, \tag{2.14}$$

donde $\bar{x}$, σ_x, $\bar{y}$ y σ_y son las medias y desviaciones estándar de las coordenadas x y y, respectivamente, de los puntos.

La homografía se calcula finalmente a partir de la ecuación (2.13):

$$\lambda\mathbf{x} = T_1^{-1}H'T_2\mathbf{X} = H\mathbf{X}. \tag{2.15}$$

La homografía es entonces calculada como $H = T_1^{-1}H'T_2$.

2.1.3. Estimación de la pose a partir de una homografía

Se tiene el modelo del marcador, $\mathbf{X}$, y una imagen tomada del marcador, $\mathbf{x}$. La estimación de la pose significa estimar la matriz R y el vector de translación $\mathbf{t}$ de la ecuación del modelo de la cámara obscura en (2.6).

La homografía relaciona los puntos en ambos planos $\mathbf{x}$ y $\mathbf{X}$ como:

$$\lambda\mathbf{x} = H\mathbf{X}, \tag{2.16}$$

donde $\mathbf{X} = [x, y, 1]^{\mathrm{T}}$ son puntos en el plano del marcador, que se representarán como $\mathbf{X}$ para dejar claro que estos puntos pertenecen a la escena real. Usando esta idea en el modelo de la cámara oscura en (2.6) se tiene:

$$\lambda \mathbf{x} = K[\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3, \mathbf{t}] \begin{bmatrix} x \\ y \\ 0 \\ 1 \end{bmatrix} = K[\mathbf{r}_1, \mathbf{r}_2, \mathbf{t}] \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{2.17}$$

En la última expresión en (2.17), la matriz $K[\mathbf{r}_1, \mathbf{r}_2, \mathbf{t}]$ es una matriz de tamaño 3×3 que relaciona los puntos en la imagen $\mathbf{x}$ con los del modelo $\mathbf{X}$, por lo tanto también es una homografía y se tiene:

$$K[\mathbf{r}_1, \mathbf{r}_2, \mathbf{t}] = \lambda H. \tag{2.18}$$

Si se tiene la matriz de calibración, entonces es fácil la obtención de R y $\mathbf{t}$ (la pose) del marcador con los siguientes pasos:

1. Calcular: $B = \lambda K^{-1} H$.

2. El factor de escala λ puede recuperarse aquí dado que los vectores $\mathbf{r}_1$ y $\mathbf{r}_2$ tienen que ser unitarios. Dado que H incorpora también ruido en las posiciones de los puntos, λ se puede recuperar como:
$$\lambda_1 = \|\mathbf{b}_1\|, \lambda_2 = \|\mathbf{b}_2\|, \text{ y } \lambda = \frac{\lambda_1 + \lambda_2}{2}$$

3. Conociendo el factor de escala podemos calcular ahora:
$$[\mathbf{r}_1, \mathbf{r}_2, \mathbf{t}] = \frac{1}{\lambda} B,$$
donde $\mathbf{t} = \frac{1}{\lambda}\mathbf{b}_3$.

4. La matriz de rotación se calcula como:
$$R' = [\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_1 \times \mathbf{r}_2].$$
Pero como la homografía contiene también ruido, esta matriz R' no es ortogonal. Se tiene que ortogonalizar usando su descomposición en valores singulares:
$$R' = UDV^{\mathrm{T}}.$$
Finalmente, la matriz de rotación es $R = UV^{\mathrm{T}}$.

2.2. Detección del marcador

Para realizar la detección del marcador se hizo uso de un algoritmo basado en el tipo de orden (TO) [13]. En general, los algoritmos de detección de marcadores desempeñan 3 tareas principales:

la identificación del marcador, la corrección de la imagen del marcador y la decodificación del marcador [13]. El primero de estos pasos consiste en identificar los puntos que formar parte de la figura del marcador. El segundo paso se realiza calculando la homografía entre los puntos del marcador en la imagen capturada y los puntos del modelo original del marcador. El último paso consiste en decodificar la imagen procesándola bit por bit, haciendo su correspondiente verificación y corrección. Usando el TO se tiene la ventaja de que solo se requiere un paso: la identificación del marcador. Las otras dos tareas, la corrección de la imagen del marcador y su decodificación, no son necesarias y quedan implícitas en el primer paso.

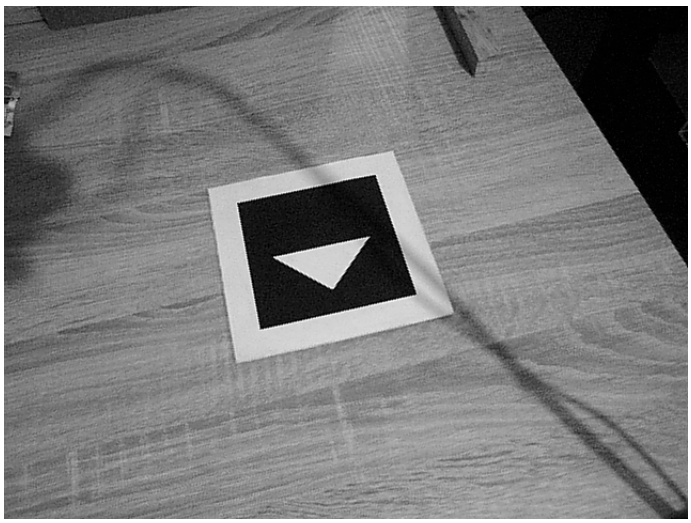


Figura 2.4: Imagen en tonos de gris del marcador tomada con la cámara PS Eye.

2.2.1. Estimar la posición de los vértices del triángulo

El marcador que se utiliza para el sistema de RA tiene un triángulo blanco, como se muestra en la figura 2.4. Una vez que el marcador es detectado, como la figura más obscura en la imagen que tiene cuatro esquinas, es necesario conocer los vértices del triángulo. Se deben realizar las siguientes tareas:

1. Marcar su perímetro.

2. Extraer todos las posiciones de los pixeles en un arreglo. El algoritmo de extracción garantiza que los píxeles del perímetro se extraen en el sentido antihorario.

3. Detectar el primer vértice como el primer pixel buscado (en la secuencia de la imagen, de arriba hacia abajo y de izquierda a derecha).

4. El segundo vértice es el pixel más alejado del encontrado en el punto anterior.

5. El tercer vértice es el pixel más alejado de la línea que une los dos vértices encontrados anteriormente [26].

De esta forma obtenemos los vértices del triángulo de forma cruda [26]. Para refinar esas posiciones se realiza el ajuste de los puntos de cada arista a una línea. Luego, se toma la intersección de las líneas como los vértices refinados.

El ajuste de las líneas se hace por medio de los componentes principales [26]. Para cada conjunto de puntos se debe realizar lo siguiente:

1. Calcular la media de las coordenadas x y y.

2. Substraer a cada punto su media. Sea este punto de la forma $\mathbf{p'} = [x', y']^T$.

3. Calcular la matriz:
$$\sum \mathbf{p'}\mathbf{p}^T = \begin{bmatrix} \sum (x')^2 & \sum x'y' \\ \sum x'y' & \sum (y')^2 \end{bmatrix}$$

4. La dirección de la línea será el eigenvalor asociado al eigenvalor más grande de la matriz 2×2 del paso anterior [26].

Una vez calculadas las tres líneas de cada lado del triángulo se calculan las intersecciones de las mismas. Las intersecciones de las líneas serán los vértices refinados del triángulo. Este mismo proceso de refinamiento se realiza con las esquinas del cuadrado del marcador. Así, al final tenemos las posiciones de los vértices del cuadrado y del triángulo conocidas con precisión subpíxel.

2.2.2. Emparejamiento de los puntos utilizando el tipo de orden

Para calibrar la cámara y encontrar la pose del marcador es necesario emparejar los puntos del marcador y los puntos detectados en la imagen [26], como se muestra en la figura 2.5.

Los puntos del modelo del marcador de la figura 2.5 se reproducen en la tabla 2.1, en el orden circular para generar la matriz lambda mínima lexicográfica.

Tabla 2.1: Puntos del modelo del marcador de la figura 2.5.

Orden	x (en mm)	y (en mm)
0	-55.0	-55.0
1	55.0	-55.0
2	0.0	-37.5
3	37.5	0.0
4	55.0	55.0
5	-37.5	0.0
6	-55.0	55.0

Si almacenamos los puntos de la tabla 2.1 en un archivo llamado `p1`, multiplicamos los puntos

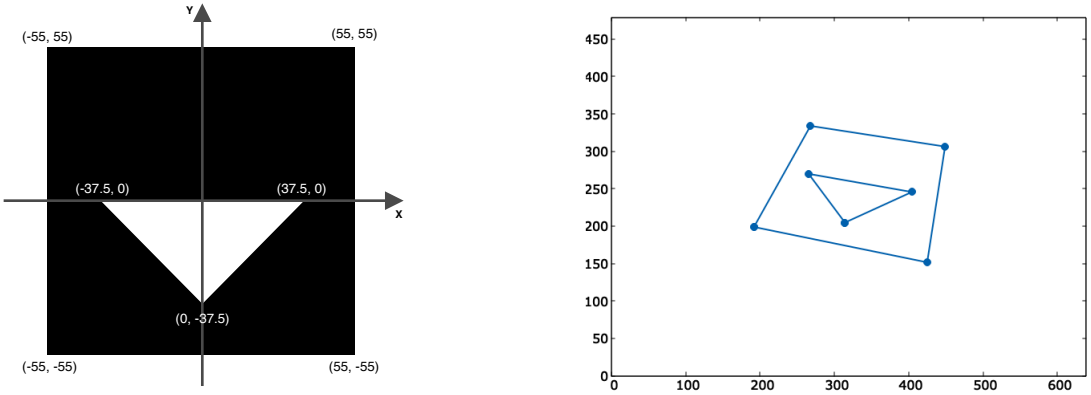


Figura 2.5: Puntos del modelo del marcador y una proyección del mismo calculada en la posición de la cámara $[50, -200, 200]^{\mathrm{T}}$. La unidades del marcador a la izquierda y de la posición de la cámara están en milímetros. Las unidades de la imagen a la derecha son en píxeles. Se usó la matriz de calibración en la expresión (2.9), en la pág. 15

por 10, para tener coordenadas enteras, y calculamos su matriz lambda con el script disponible en [13][1], obtenemos:

```
$ ./Scripts/ComputesLambdaMatrix/computesLambdaMatrix p1 1
Labels: 0 1 2 3 4 5 6
145234312434451
```

Aquí lo importante es ver que los puntos en la tabla 2.1 ya están en el orden circular para generar la matriz lambda mínima lexicográfica.

Una vez que se obtienen los puntos del marcador en la imagen, estos deben ser ordenados según la secuencia que obtuvimos del cálculo de la matriz lambda. Realizando la proyección de los puntos del modelo del marcador según la matriz de calibración (2.9) con la cámara en la posición $[50, -200, 200]^{\mathrm{T}}$, utilizando el script del apéndice B, obtenemos los puntos mostrados en la tabla 2.2.

Tabla 2.2: Puntos proyectados con la matriz de calibración (2.9) con la cámara en la posición $[50, -200, 200]^{\mathrm{T}}$.

Orden	x (píxels)	y (píxels)
0	267.79	333.88
1	448.98	306.26
2	265.36	269.72
3	404.31	245.54
4	314.16	204.72
5	191.81	198.81
6	424.91	151.53

Los puntos en la tabla 2.2 están en el orden en que son reconocidos en la imagen. Para ordenar-

[1] O directamente en `http://cs.cinvestav.mx/~fraga/OTT/Data.zip`

los en la misma correspondencia de los puntos del modelo del marcador en la tabla 2.1, aplicamos de nuevo el script `computesLambdaMatrix` (los puntos están en el archivo p2 y multiplicados por 100) y obtenemos:

```
$ ./Scripts/ComputesLambdaMatrix/computesLambdaMatrix p2 1
Labels: 5 6 4 3 1 2 0
145234312434451
```

La secuencia `5 6 4 3 1 2 0` nos indica el orden en que deben tomarse según la secuencia de la tabla 2.2. Finalmente los puntos quedan como se muestra en la tabla 2.3. Vea que en la tabla 2.3, en la primera columna precisamente está en forma vertical la secuencia `5 6 4 3 1 2 0`, y cada número en esta columna corresponde al orden de los puntos en la primera columna de la tabla 2.2.

Tabla 2.3: Los puntos de la tabla 2.2 ordenados según el tipo de orden.

Orden	x (píxels)	y (píxels)
5	191.81	198.81
6	424.91	151.53
4	314.16	204.72
3	404.31	245.54
1	448.98	306.26
2	265.36	269.72
0	267.79	333.88

Ya que los puntos estén ordenados se puede realizar el cálculo de la homografía y la estimación de la pose del marcador, como se indican en la secciones 2.1.2 y 2.1.3 respectivamente.

2.3. Objetos deformables

Los objetos deformables son objetos virtuales que modelan cuerpos que se deforman al aplicarse una fuerza sobre ellos [28]. Estos objetos son modelados tomando como base la deformación de un resorte. Un resorte se deforma con base en la ley de elasticidad de Hooke, como se observa en la figura 2.6.

El resorte de la figura 2.6 se representa matemáticamente por la expresión (2.19).

$$f = kx \tag{2.19}$$

Los objetos deformables se forman mediante una malla de simplejos. En cada arista de la malla se tiene un sistema masa-resorte-amortiguador (MRA), como el de la figura 2.7. Este sistema da a la malla la propiedad de deformarse mediante la aplicación de una fuerza. El sistema MRA es resuelto por medio de las ecuaciones de movimiento de Newton [28].

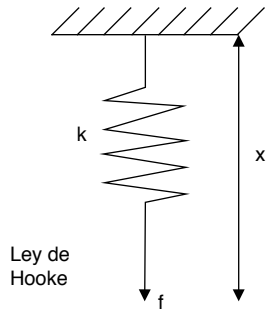


Figura 2.6: Ley de Hooke, donde k es el coeficiente de elasticidad del resorte, x el alargamiento del resorte y f es la fuerza.

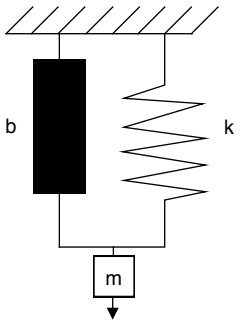


Figura 2.7: Sistema resorte-masa-amortiguador.

El sistema MRA es representado por la ecuación diferencial de segundo orden de la expresión (2.20).

$$f = m\ddot{x} + b\dot{x} + kx \tag{2.20}$$

En el sistema MRA la masa, m, hace que el sistema se comporte según la segunda ley de Newton, y el amortiguador, b, se encarga de disipar la energía [28]. Sin el amortiguador, el resorte y la masa oscilarían eternamente. El sistema MRA se resuelve mediante la ecuación diferencial de segundo orden de la expresión (2.20). Esta ecuación tiene tres tipos de soluciones, como se puede ver en la figura 2.8.

Lo que se busca es que el objeto tenga una respuesta rápida, por lo que la solución subamortiguada es la de mayor interés para el modelado de los objetos deformables.

La ecuación de segundo orden del sistema MRA puede resolverse mediante un método numérico. Se puede resolver utilizado el método de diferencias finitas [30].

Discretizando el tiempo t puede cuantificarse la evolución del sistema de la ecuación (2.20) bajo las leyes de movimiento las cuales definen la velocidad $\dot{x}$ y la aceleración $\ddot{x}$ como:

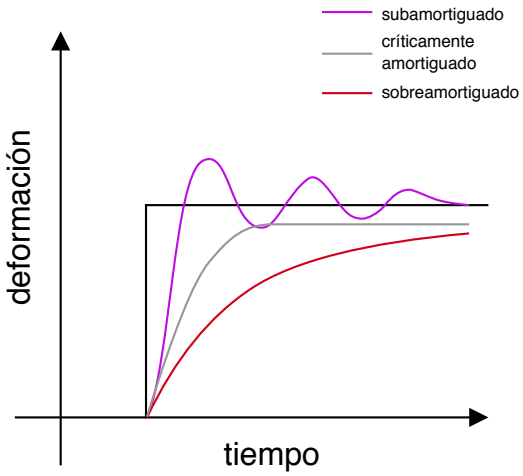


Figura 2.8: Soluciones de la ecuación del sistema MRA

$$\dot{x} = \frac{x_t - x_{t-1}}{\Delta_t}$$

$$\ddot{x} = \frac{\dfrac{x_{t+1} - x_t}{\Delta_t} - \dfrac{x_t - x_{t-1}}{\Delta_t}}{\Delta_t}$$

Sustituyendo estas dos expresiones en la ecuación (2.20).

$$\begin{aligned} m\ddot{x} + b\dot{x} + kx &= f \\ \ddot{x} + \frac{b}{m}\dot{x} + \frac{k}{m}x &= \frac{f}{m} \\ \frac{\dfrac{x_{t+1} - x_t}{\Delta_t} - \dfrac{x_t - x_{t-1}}{\Delta_t}}{\Delta_t} + \frac{b}{m}\frac{x_t - x_{t-1}}{\Delta_t} + \frac{k}{m}x &= \frac{f}{m} \end{aligned} \tag{2.21}$$

Desarrollando la expresión anterior tenemos que:

$$\begin{aligned} x_{t+1} &= \underbrace{\left(2 - \frac{\Delta_t b}{m} - \frac{(\Delta_t)^2 k}{m}\right)}_{k_1} x_t + \underbrace{\left(-1 + \frac{\Delta_t b}{m}\right)}_{k_2} x_{t-1} + \underbrace{\frac{(\Delta_t)^2}{m}}_{k_3} f \\ x_{t+1} &= k_1 x_t + k_2 x_{t-1} + k_3 f \end{aligned} \tag{2.22}$$

Con la expresión (2.22) podemos calcular el estado x_{t+1} a partir del estado actual y el estado anterior.

Para la implementación de los objetos deformables en nuestro sistema de RA se utilizaron los cuerpos blandos (*Soft Body Dynamics*) de la biblioteca Bullet Physics [14]. La simulación de cuerpos blandos en la biblioteca Bullet Physics se realiza usando la dinámica basada en la posición [19,24]. El enfoque para simulaciones de objetos deformables utilizando el sistema MRA están basados en fuerzas. Fuerzas internas y externas son acumuladas para calcular la aceleración de los cuerpos con base en la segunda ley de movimiento de Newton. Luego se utiliza un método para actualizar su velocidad y finalmente la posición del objeto. A diferencia de este, el enfoque de la dinámica basada en la posición omite la capa que calcula la velocidad y se enfoca en el cálculo de la posición [19,24].

2.3.1. Dinámica basada en la posición

En la dinámica basada en la posición, los objetos a ser simulados son representados por un conjunto de N partículas y un conjunto de M restricciones [25]. Cada partícula cuenta con los atributos mencionados en la tabla 2.5.

Tabla 2.4: Atributos de una partícula [25].

m_i	masa
x_i	posición
v_i	velocidad

Mientras que una restricción cuenta con 5 atributos, los cuales pueden verse en la tabla 2.5.

Tabla 2.5: Atributos de una restricción [25].

n_i	cardinalidad
$C_j : \mathbb{R}^{3n_j} \rightarrow \mathbb{R}$	función de restricción escalar
$\{i_1, \ldots, i_{n_j}\}, i_k \in [1, \ldots, N]$	conjunto de índices
$k_i \in [0 \ldots 1]$	parámetro de rigidez
unilateral o bilateral	tipo

El atributo tipo bilateral se satisface si $C_j(x_i, \ldots, x_{i_{n_j}}) = 0$ y el tipo unilateral se satisface si $C_j(x_i, \ldots, x_{i_{n_j}}) \geq 0$. El parámetro de rigidez k_i define la fuerza de la restricción en un rango de cero a uno.

Con está información a cada paso de tiempo Δ_t, la simulación realiza lo siguiente:

```
1 forall particulas i
2   inicializamos x_i = x_i^0 , v_i = v_i^0 , w_i = 1/m_i
3 endfor
4 loop
5   forall particulas i do v_i ← v_i + Δ_t w_i f_ext(x_i)
6   forall particulas i do p_i ← x_i + Δ_t v_i
```

```
7   forall particulas i do generateCollisionContraint(x_i → p_i)
8   loop solverIterations times
9     projectConstraint(C_1, ..., C_{M+M_coll}, p_1, ..., p_N)
10  endloop
11  forall particular i
12    v_i ← (p_i − x_i)/Δ_t
13    x_i ← p_i
14  endfor
15 endloop
```

Ya que el algoritmo simula un sistema de segundo orden en el tiempo, tanto la posición como la velocidad de las partículas deben ser especificadas en las líneas (1)-(3) antes de que comience el ciclo de la simulación. Las líneas (5)-(6) realizan un paso de integración de Euler hacia delante explícito en las velocidades y posiciones. La nueva ubicación p_i no es asignada a la posición directamente sino que es usada como predicción. Las restricciones externas no permanentes, así como las restricciones de colisión, se generan al comienzo de cada paso de tiempo en la línea (7). Aquí la posición original y la posición predicha son usadas para la detección continua de colisiones. El solucionador (8)-(10) corrige de forma iterativa las posiciones predichas de modo que satisfagan las restricciones externas M_{coll} como internas M. Finalmente, las posiciones p_i corregidas son utilizadas para actualizar la velocidad y posición. Es esencial actualizar la velocidad junto con la posición, ya que si no se realiza de esta forma, la simulación no produce el comportamiento correcto de un sistema de segundo orden [25].

2.4. Detección de colisiones

Para saber cuando el usuario está interactuando con un objeto virtual es necesario detectar el momento en que estos entran en contacto. Por lo que necesitamos un algoritmo de detección de colisiones para saber cuándo la interacción entre el usuario y los objetos ocurre.

La biblioteca Bullet Physics es un motor de simulación basado en física que nos permite hacer simulaciones del comportamiento físico de los objetos virtuales [14]. La biblioteca también nos proporciona la detección de colisiones.

Cada tarea que desempeña el motor de simulación basado en física tiene su propio componente, lo cual permite que la biblioteca sea altamente ajustable. El componente de Bullet Physics que se encarga de la detección de colisiones se conoce como fase amplia.

2.4.1. Fase amplia

Una simulación basada en física se ejecuta en tiempo real, pero lo hace en pasos discretos de tiempo. Por lo que, este componente se encarga de verificar si existen colisiones entre los objetos en la simulación en cada paso de tiempo [14].

El proceso conocido como *detector de colisiones de fase amplia* tiene como propósito eliminar los pares de objetos que tienen poca o ninguna posibilidad de colisionar y hacer una lista de los objetos que tienen mayor posibilidad de colisionar [14]. Esto con el fin de evitar todo el procesamiento que produciría verificar si existe colisión de un objeto con el resto de objetos en la escena (método de fuerza bruta) y reducir la complejidad del proceso de detección de colisiones.

Una vez que se tiene la lista de objetos que pueden colisionar, se pasan a otro componente del motor de simulación basado en física llamado *detección de colisión de fase angosta*, el cual verifica la lista para detectar colisiones utilizando técnicas matemáticas precisas [14].

Bullet Physics utiliza una técnica de fase amplia basada en volúmenes de límite dinámico. Este algoritmo crea volúmenes de espacio que envuelven pares de objetos en un árbol jerárquico usando *cajas envolventes alineadas con los ejes (CEAEs)*. Estas CEAEs rodean al objeto con la caja más pequeña posible, que es alineada con cada eje, como se observa en la figura 2.9.

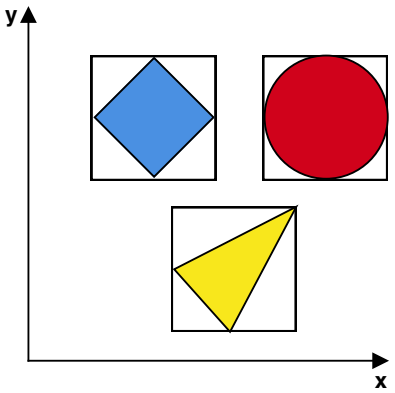


Figura 2.9: CEAEs rodeando distintos objetos [14].

Se organizan pares de CEAEs en una estructura de árbol para que la verificación de colisiones sea menos costosa. Se organizan los objetos por la distancia, separando los objetos mas alejados, como se muestra en la figura 2.10.

Finalmente, se utiliza un simple árbol binario para que los objetos sean procesados por el detector de colisiones. Además, existe un procesamiento para mantener la organización del árbol de la figura 2.10, que es menos costoso que verificar cada par de CEAEs.

2.4.2. Fase angosta

Las técnicas utilizadas en la fase angosta para detectar la colisión entre dos objetos depende de la figura de colisión de cada objeto. Bullet Physics define una gran variedad de tipos de figuras de colisión, incluso ofrece la posibilidad de definir una figura de colisión [11]. Las primitivas que ofrece como figuras de colisión la biblioteca son:

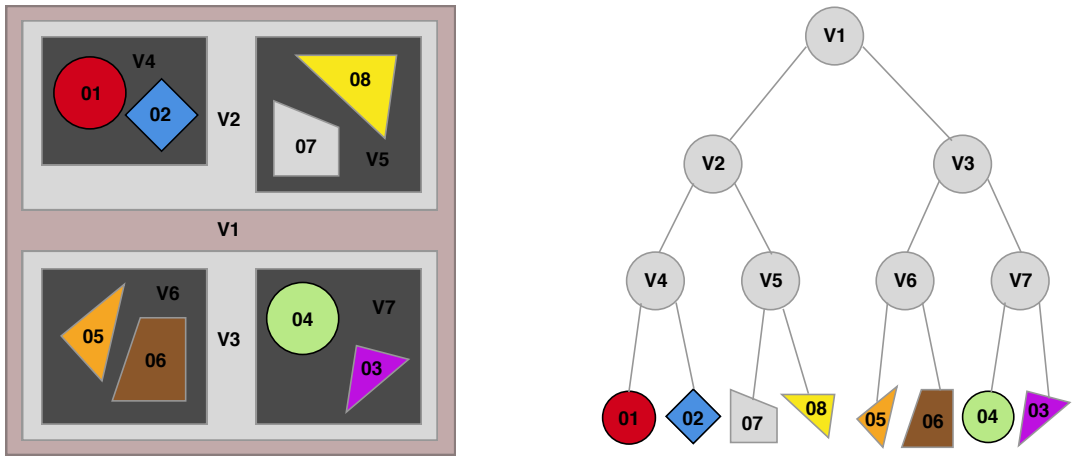


Figura 2.10: Organización de las CEAEs utilizando un árbol binario [14].

1. *btBoxShape*: Cada definida por las medias extensiones de sus lados.

2. *btSphereShape*: Esfera definida por su radio.

3. *btCapsuleShape*: Cápsula alrededor del eje Y.

4. *btCylinderShape*: Cilindro alrededor del eje Y.

5. *btConeShape*: Cono alrededor del eje Y.

6. *btMultiSphereShape*: Cubierta convexa de múltiples esferas [11].

Además de estas primitivas básicas Bullet Physics ofrece la posibilidad de crear figuras de colisión combinando las primitivas que ofrece para crear figuras compuestas (*btCompoundShape*) y también permite representar figuras de colisión más complejas utilizando una malla de triángulos utilizando figuras de cubierta convexa (*btConvexHullShape*).

Para cada par de tipos de figuras de colisión, Bullet Physics eligirá un determinado algoritmo de colisión para cada caso [11]. Los algoritmos elegidos para cada par de figuras de colisión están definidos en la tabla 2.6.

El algoritmo de distancia Gilbert–Johnson–Keerthi (GJK) es un método que determina la distancia euclidiana entre un par de conjuntos convexos [16].

Los métodos más simples que podemos describir para comprender mejor el funcionamiento del detector de colisiones son:

- Colisión esfera contra esfera

Tabla 2.6: Algoritmos para detección de colisión dependiendo el tipo de figura de colisión [11].

	caja	esfera	convexa, cilindro, cono, cápsula	compuesta	malla de triángulos
caja	caja contra caja	esfera contra caja	gjk	compuesta	cóncavo convexo
esfera	esfera contra caja	esfera contra esfera	gjk	compuesta	cóncavo convexo
convexa, cilindro, cono, cápsula	gjk	gjk	gjk	compuesta	cóncavo convexo
compuesta	compuesta	compuesta	compuesta	compuesta	compuesta
malla de triángulos	cóncavo convexo	cóncavo convexo	cóncavo convexo	compuesta	gimpact

- Colisión esfera contra plano
- Colisión esfera contra triángulo
- Colisión esfera contra caja

En el caso de la colisión esfera contra esfera tenemos que los datos necesarios para detectar la colisión son el centro y el radio de cada esfera. Con estos datos se puede determinar la distancia euclidiana entre los centros y comprobar mediante la suma de los radios si las dos esferas están colisionando.

En la figura 2.11 se puede ver que cuando la distancia euclidiana entre los dos centros es menor que la suma de los radios tenemos una colisión. Por lo que la desigualdad (2.23) determina cuando existe una colisión entre dos esferas.

$$\|\mathbf{c}_2 - \mathbf{c}_1\|^2 < (r_1 + r_2)^2, \tag{2.23}$$

donde $\mathbf{c}_1$, r_1, y $\mathbf{c}_2$, r_2 son el centro y el radio de las esferas 1 y 2, respectivamente.

La colisión esfera contra plano se realiza determinando la distancia mínima de un punto al plano. Tomando como punto el centro de la esfera y comparando el radio de la esfera con esta distancia. El plano queda definido por un punto sobre de él y el vector normal a él.

De la figura 2.12 podemos determinar la distancia del centro de la esfera $\mathbf{c}$ al punto arbitrario $\mathbf{p}$ del plano como $\mathbf{v} = \mathbf{c} - \mathbf{p}$. Después, podemos determinar la distancia mínima del plano al centro de la esfera como:

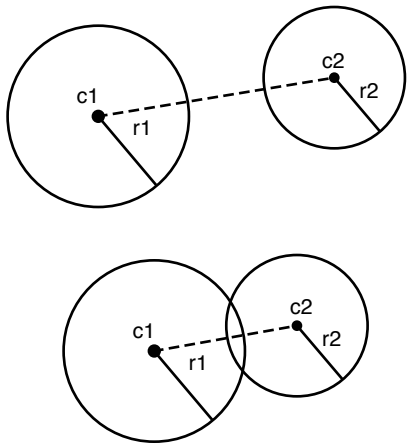


Figura 2.11: Colisión esfera contra esfera.

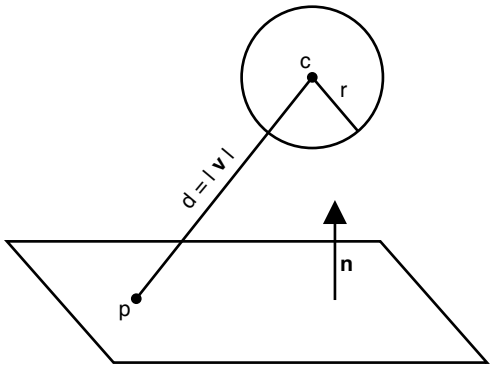


Figura 2.12: Colisión esfera contra plano.

$$d = \mathbf{v} \cdot \mathbf{n}. \tag{2.24}$$

Por medio de distancia de la ecuación (2.24) podemos saber si hay colisión entre el plano y la esfera. Como lo determina la expresión (2.25), si la distancia mínima entre el plano y el centro de la esfera es menor al radio de la esfera, entonces existe una colisión.

$$d = \mathbf{v} \cdot \mathbf{n} < r. \tag{2.25}$$

La colisión de una esfera y una caja consiste en determinar el punto más cercano de la caja al centro de la esfera, cómo se ve en la figura 2.13.

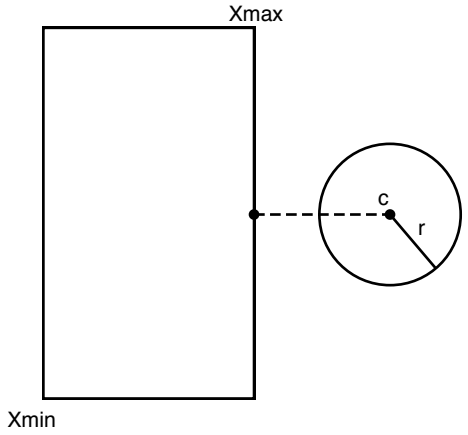


Figura 2.13: Colisión esfera contra caja.

Esto se puede realizar mediante la siguiente función:

```
anclar(Vmin, Vmax, v)
  if v < Vmin
    return Vmin
  else if v > Vmax
    return Vmax
  else
    return v

x = anclar(Xmin, Xmax, cx)
y = anclar(Ymin, Ymax, cy)
```

Para el ejemplo de la figura 2.13 obtendríamos el punto $\mathbf{x} = [\text{Xmax}, c_y]^{\text{T}}$. Una vez teniendo el punto $\mathbf{x}$ más cercano al centro de la esfera podemos determinar si hay colisión mediante el cálculo de la distancia y la comparación con el radio de la esfera, con la expresión 2.26.

$$\|\mathbf{x} - \mathbf{c}\| < r, \qquad \mathbf{x} = [x, y]^T. \tag{2.26}$$

2.4.3. Resumen

En este capítulo se revisaron los conceptos aplicados en el desarrollo del sistema de RA. Se han presentado conceptos como el modelo de la cámara obscura, que es el modelo más simple de una cámara. La calibración de la cámara que es un proceso que se debe realizar para conocer los parámetros intrínsecos de la cámara que se está usando. El cálculo de la homografía como

el proceso para detectar la rotación y translación del modelo del marcador, esto nos permitirá dibujar los objetos virtuales encima del mismo marcador, en el sistema de coordenadas en el que está inmerso el marcador. El tipo de algoritmo utilizado para la detección del marcador. El algoritmo para la detección de colisiones que nos permite conocer cuando el usuario interactua con los objetos. Finalmente, hablamos de los objetos deformables, su modelado y cómo se implementan en nuestro sistema de RA. Estos conceptos fueron revisados con el fin de que el proceso de diseño, construcción e implementación que presentaremos en los siguientes capítulos sean más fáciles de comprender.

Capítulo 3

Diseño del sistema

En este capítulo se presenta el diseño del sistema de RA propuesto. Primero, mostraremos el diagrama que describe el funcionamiento del sistema donde se plantean las tareas principales que se desempeñan en él. Se da una introducción de los elementos de hardware que componen el sistema y se habla de su función dentro del mismo. Finalmente, se presentan las herramientas de software utilizadas para el desarrollo del software del sistema.

3.1. Diagrama del sistema

El funcionamiento del sistema de RA está representado en el diagrama de la figura 3.1. El sistema cuenta con cinco componentes principales: el procesamiento de la imagen, la visión (incluyendo la calibración de la cámara), los objetos virtuales, la detección de colisiones y la simulación basada en física, y por último la interacción con los objetos virtuales

El software procesa cada una de las imágenes capturadas por la cámara para obtener los vértices que componen al marcador. Posteriormente, con los vértices se calcula la homografía y se resuelve el modelo de la cámara obscura para estimar la pose del marcador. Además, en este punto también se realiza la calibración de la cámara para obtener la matriz de calibración. Finalmente, teniendo la pose del marcador y la matriz de calibración de la cámara se pueden presentar los objetos virtuales en la escena y ser visibles para el usuario.

A su vez se procesa la información obtenida del LM sobre el seguimiento del movimiento de las manos. Las coordenadas sobre la posición de las manos que nos brinda el LM son transformadas a coordenadas en el sistema de coordenadas definido por el marcador. Esto nos permite representar las puntas de los dedos índice y pulgar con esferas dentro del sistema, para que dichas esferas colisionen con los objetos virtuales de la escena y así el usuario puede interactuar con ellos.

Por último, con el uso de la biblioteca Bullet Physics proporcionamos a los objetos con un

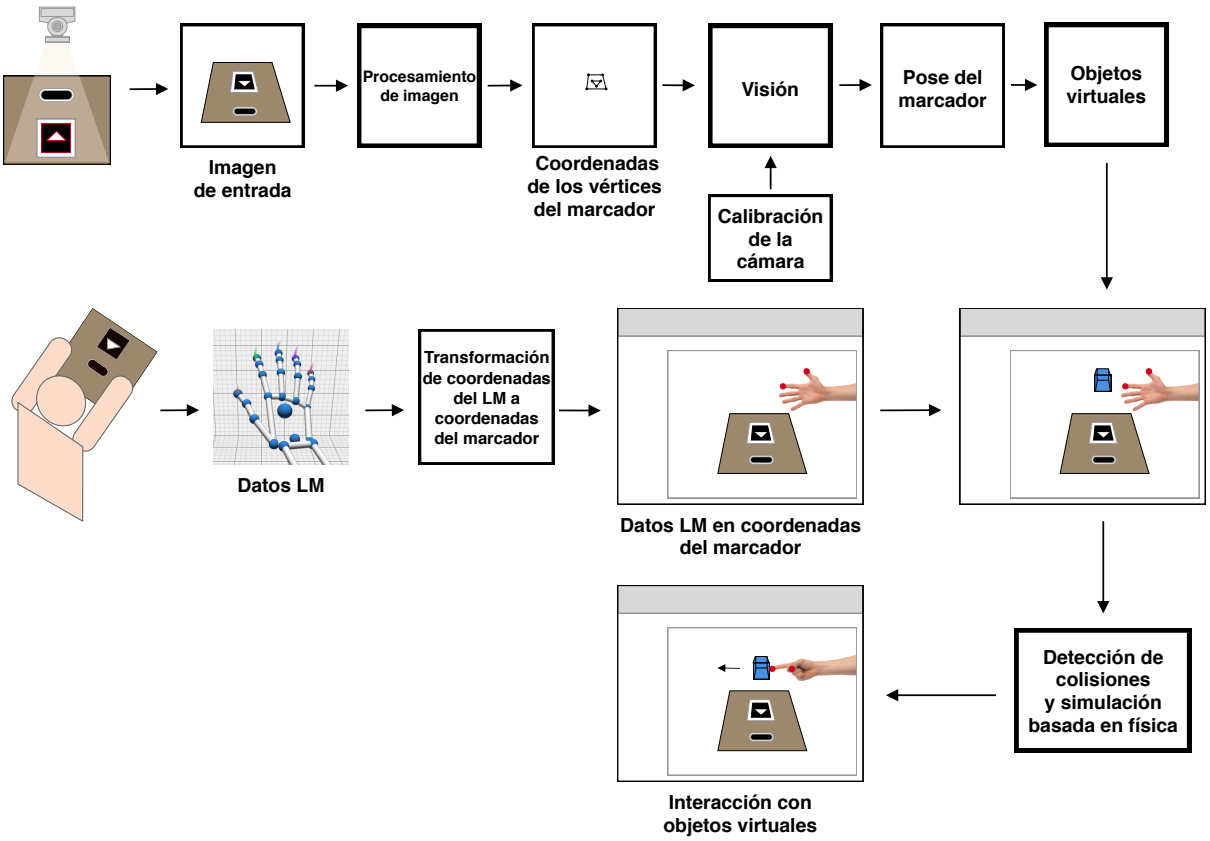


Figura 3.1: Diagrama del sistema de RA.

comportamiento físico realista e implementamos la detección de colisiones para que pueda existir una respuesta por parte de los objetos al momento de interactuar con el usuario.

3.2. Hardware

A continuación se presentan cada uno de los elementos de hardware que componen el sistema y la forma en la que se interconectan todos estos elementos para que el sistema funcione.

3.2.1. Dispositivo Leap Motion

El dispositivo Leap Motion realiza el seguimiento del movimiento de las manos, ofreciéndonos información sobre su posición. LM cuenta con su propio sistema de coordenadas como se muestra en la figura 1.4(b), en la pág. 5. El dispositivo nos ofrece información de la posición de cada uno de los huesos de las manos, proporcionándonos la posición de dónde inicia y termina cada uno

de ellos. En la figura 1.4(a) en la pág. 5, se muestran los huesos que componen la mano humana y el nombre de cada uno de ellos.

El dispositivo Leap Motion tiene la forma de una caja rectangular de dimensiones 80 mm × 30 mm × 11 mm. LM puede conectarse por medio del conector USB a una computadora. El dispositivo está compuesto de 3 LEDs infrarrojos y dos cámaras [3], como se ve en la figura 3.2. El principio del funcionamiento de LM es la visión estereoscópica, donde la luz de los LEDs reflejada se ve desde dos puntos diferentes y se mide las distancias por medio de este reflejo. LM tiene un rango de visión de 150°, el cual tiene la forma de una pirámide invertida [27], tal como se muestra en la figura 1.6.

Figura 3.2: Componentes del dispositivo Leap Motion [3].

3.2.2. Cámara PlayStation Eye

La cámara PlayStation Eye es capaz de capturar video estándar a una velocidad de 60 Hz con una resolución de 640 × 480 píxeles y a 120 Hz con una resolución de 320 × 240 [5]. La cámara puede ser conectada a una computadora por medio de su conector USB.

En la figura 3.3 se observa que la cámara cuenta con un selector de ángulo de la lente. El selector tiene dos posiciones indicada con color azul y rojo. El indicador azul sirve para seleccionar un *ángulo de visión amplio*, mientras que el indicador rojo selecciona un *ángulo de visión normal* [5]. Nosotros usamos el selector puesto en el indicador azul para el sistema de RA.

3.2.3. Características del equipo de cómputo

Para el desarrollo del software se utilizó un equipo de cómputo con las siguientes características:

- **Procesador:** Intel Core i5 2.7 GHz

Figura 3.3: Cámara PlayStation Eye.

- **RAM:** 8GB 1867 MHz DDR3
- **Tarjeta gráfica:** Intel Iris Graphics 6100 1536 MB
- **Sistema Operativo:** Fedora 28

3.2.4. Construcción del sistema

El sistema de RA propuesto está compuesto por el dispositivo LM, la cámara PS Eye, un marcador de RA, un monitor y un equipo de cómputo. El dispositivo LM y la cámara se encuentran encima de una tabla cuadrada de madera. Esto es debido a que ambos deben mantener su distancia fija, ya que esto nos permite calcular la transformación de coordenadas del sistema de coordenadas del LM al sistema de coordenadas del marcador, esto se explicará de forma más clara en la siguiente sección. El sistema se monta sobre alguna superficie plana, como una mesa o escritorio. Sobre esta superficie se coloca la tabla de madera con el LM y el marcador. La cámara se sujeta de la superficie a una altura lo más cercana posible a la vista del usuario para que la cámara tenga una perspectiva lo más cercana a la perspectiva del usuario. La figura 3.4 muestra la configuración del sistema de RA.

El usuario apreciará una aplicación en la pantalla, en donde se mostrará la imagen que captura la cámara y los elementos virtuales que componen la escena.

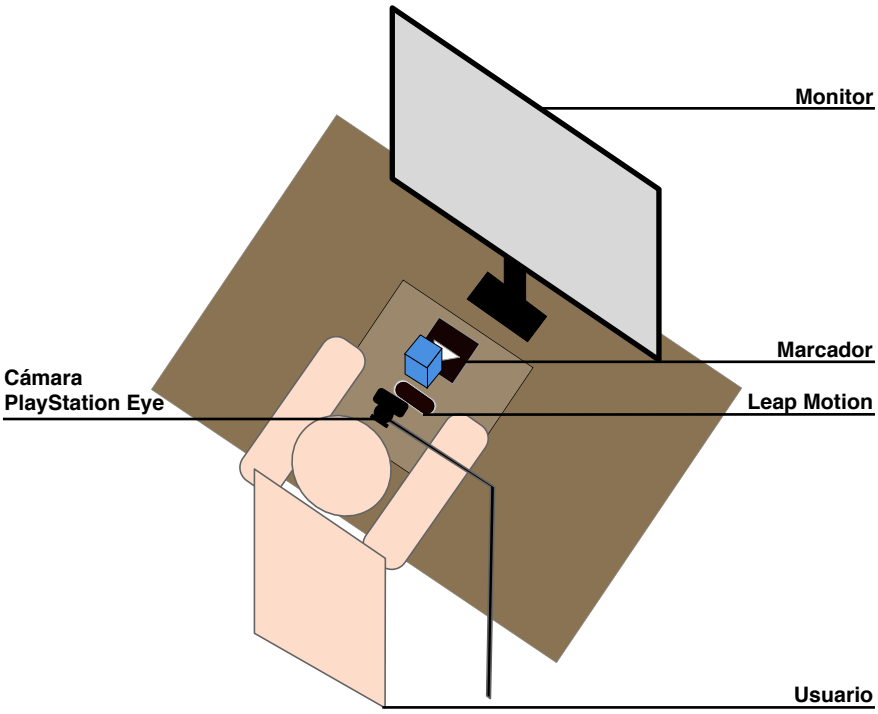


Figura 3.4: Configuración del sistema de RA.

3.3. Herramientas de software

A continuación se presentan las bibliotecas y herramientas empleadas para el desarrollo del software del sistema.

3.3.1. Qt

Qt es un entorno de trabajo orientado a objetos utilizado para la creación de interfaces gráficas. Este entorno se utilizó en el desarrollo de esta tesis precisamente para crear la interfaz del sistema que ve el usuario. La interfaz muestra la imagen que captura la cámara y también cuenta con algunos botones en la parte izquierda, como se muestra en la figura 3.5.

Los botones de la interfaz son los siguientes:

- **Salir:** Cierra la aplicación.
- **Iniciar:** Inicia la simulación de los objetos.

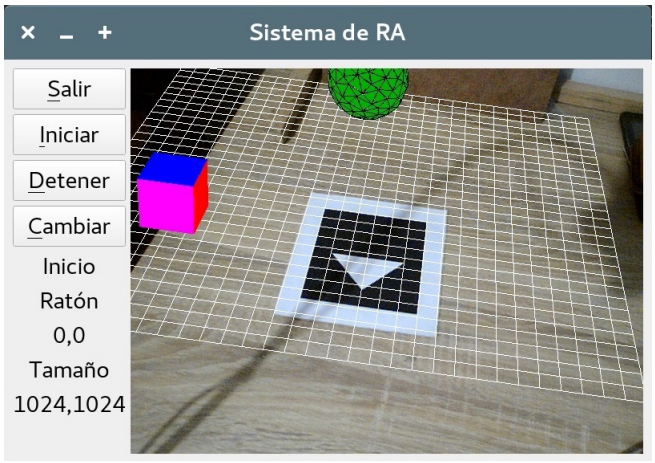


Figura 3.5: interfaz de la aplicación del sistema de RA.

- **Detener:** Detiene la simulación de los objetos.
- **Cambiar:** Cambia entre la imagen a color que ve la cámara o la imagen blanco y negro resultante de la segmentación de la imagen a color.

3.3.2. OpenCV

OpenCV es una biblioteca de código libre que incluye cientos de algoritmos de visión por computadora [1]. OpenCV tiene una estructura modular, es decir, la paquetería incluye varias bibliotecas compartidas o estáticas. Algunos de los módulos más importantes de la biblioteca son listados a continuación.

- **core:** Es el módulo que define las estructuras de datos básicas y las funciones básicas utilizadas por los otros módulos.
- **imgproc:** Es un módulo de procesamiento de imágenes que incluye el filtrado de imágenes lineal y no lineal, transformaciones geométricas de imágenes, histogramas, etc.
- **video:** Es un módulo de análisis de video que incluye algoritmos de estimación de movimiento y seguimiento de objetos.
- **calib3d:** Módulo que incluye algoritmos de geometría, calibración de cámara, estimación de movimiento de objetos, algoritmos de correspondencia estéreo y elementos de reconstrucción 3D.
- **highgui:** una interfaz fácil de usar para la captura de video, lectura y escritura de imágenes y video.

[1] `https://docs.opencv.org/2.4.13.7/modules/core/doc/intro.html`

OpenCV es usado en el sistema de RA para la captura de video de la cámara PS Eye. El video es procesado analizando cada uno de los cuadros que nos entrega OpenCV. El análisis de cada cuadro nor sirve para detectar el marcador y para mostrar la imagen real del sistema a la cual se le superpondrán los objetos virtuales.

3.3.3. OpenGL

La biblioteca OpenGL es una API multilenguaje y multiplataforma utilizada para generar gráficos 2D y 3D [2]. En este proyecto se hizo uso de la versión OpenGL 4.1. OpenGL es utilizado para la visualización de los objetos virtuales en la escena del sistema de RA.

Hay que considerar que OpenGL no proporciona las funciones necesarias para crear una interfaz gráfica, por lo que suele combinarse con alguna otra biblioteca que nos facilite la creación de interfaces, como lo es la biblioteca Qt.

3.3.4. Biblioteca Bullet Physics

Bullet Physics es una biblioteca de código libre para la detección de colisiones y la dinámica de cuerpos rígidos y blandos [11]. Los objetos virtuales dentro del sistema de RA deben tener un comportamiento similar a los objetos reales. Bullet Physics es un motor de simulación basado en física que nos proporciona una solución a la simulación de físicas de objetos. Algo a tomar en cuenta es que Bullet Physics solo proporciona el comportamiento de los objetos, pero no la visualización de los objetos. Para generar los objetos gráficamente se hizo uso de la biblioteca OpenGL descrita en la sección 3.3.3

Esta biblioteca ofrece un conjunto de algoritmos para la detección de colisiones mencionados previamente en la sección 2.4. Además de los algoritmos para detección de colisiones, Bullet Physics proporciona la dinámicas de cuerpos rígidos y blandos. Dentro de los cuerpos rígidos, se pueden definir tres tipos.

- **Cuerpos dinámicos:** Son los cuerpos con movimiento. Tienen una masa positiva y su posición es calculada a cada paso de la simulación para ser actualizada.

- **Cuerpos estáticos:** Son los cuerpos que permanecen en una posición fija durante toda la simulación. Estos cuerpos no presentan movimiento pero sí colisionan con los otros cuerpos. Tiene una masa igual a cero.

- **Cuerpos cinemáticos:** Son cuerpos con masa cero. Estos cuerpos pueden ser animados por el usuario, pero no son afectados por la simulación basada en física [11].

Por otra parte, los cuerpos blandos son los elementos que se deforman al aplicarse una fuerza

[2] `https://www.opengl.org/`

externa sobre ellos. Son los cuerpos utilizados para modelar el objeto deformable en esta tesis. Los cuerpos blandos podrían ser implementados con mallas de sistemas masa, resorte y amortiguador como se describió en la sección 2.3, en la pág. 23, o usando la dinámica basada en la posición, como se describió en la subsección 2.3.1, en la pág. 26. Bullet usa esta última descripción.

3.3.5. API de Leap Motion

El dispositivo Leap Motion nos proporciona una API de desarrollo con la que podemos obtener los datos de LM e incluirlos en nuestra aplicación [3]. La API está disponible para trabajar en lenguajes de programación `C#`, `C++` y Python, entre otros lenguajes.

Al momento que LM detecta las manos y los dedos dentro de su campo de visión, este provee un conjunto de datos en una clase llamada `Frame`. Cada objeto de la clase `Frame` contiene las manos detectadas, detallando sus propiedades en un instante de tiempo determinado. Este objeto es la raíz del modelo de datos de LM [1].

Dentro del objeto `Frame` se tiene el objeto `Hand` el cual detalla información sobre las manos identificadas, como su posición, orientación, el brazo al que está asociado y una lista de los dedos asociado con esa mano. Cabe destacar que LM ofrece seguimiento incluso cuando partes de las manos no son visibles haciendo una predicción de la posición de dichas partes. A pesar de que más de dos manos pueden ser detectadas por el dispositivo, se recomienda tener solo dos manos en el campo de visión de LM para obtener una mejor calidad del seguimiento [1].

El objeto `Arm` proporciona la orientación, longitud, anchura y los puntos finales de un brazo [1]. Cuando el codo no se encuentra a la vista del dispositivo este hace una predicción de su posición con base en los datos previos.

Finalmente, el objeto `Finger` proporciona la información referente a cada dedo de la mano detectada. Si no todos los dedos son visibles, entonces el dispositivo hace una predicción con base en datos anteriores, tal como lo hace para otras partes de la mano. Los dedos se identifican por su nombre en una variable tipo por una etiqueta con su nombre, es decir, se identifican como pulgar, índice, medio, anular y meñique [1]. Cada dedo contiene un objeto de la clase `Bone` que describe la posición y orientación de cada dedo de la mano [1]. Todos los dedos contienen cuatro huesos ordenados de la base a la punta del dedo, como se ve en la figura 1.4(a), en la pág. 5.

Los cuatro huesos de cada dedo son identificados como:

- **Metacarpos:** Son los huesos que se encuentra dentro de la mano que conecta a los dedos con la muñeca (excepto el pulgar).
- **Falanges proximales:** Son los huesos de la base del dedo que conecta con la palma de la mano.

[3] `https://developer-archive.leapmotion.com/documentation/python/api/Leap_Classes.html`

- **Falanges medias:** Son los huesos medio del dedo, entre la base y la punta.
- **Falanges distales:** Es el último hueso de cada dedo, el hueso de la punta.

3.3.6. Resumen

En este capítulo se presentó el diseño del sistema. Se mostró un diagrama general en donde pueden observarse todas las tareas que el sistema realiza para lograr la interacción del usuario con los objetos virtuales. También, se presentaron los componentes de hardware principales como lo son el dispositivo Leap Motion y la cámara PlayStation Eye y su funcionamiento dentro del sistema. Además, se presentaron las herramientas de software empleadas para el desarrollo de la aplicación del sistema. Conocer estos elementos de hardware y software ayudarán a comprender mejor la implementación del sistema, la cual se presenta en el siguiente capítulo.

Capítulo 4

Implementación del sistema

En este capítulo se habla sobre la implementación del sistema, comenzando por el ajuste de los sistema de coordenadas del LM y del marcador. Después, se describe cómo el usuario visualiza los objetos virtuales en la escena del sistema y el uso de la biblioteca Bullet Physics para la implementación del comportamiento físico de los objetos, el detector de colisiones y el objeto deformable. Finalmente, se muestra cómo interactúa el usuario con los objetos virtuales dentro el sistema.

4.1. Ajuste de los sistemas de coordenadas

El marco de referencia para el sistema de RA es el sistema de coordenadas del marcador. Los objetos son diseñados con respecto a este sistema de referencia. Por ello, se deben ajustar los sistemas de coordenadas entre el dispositivo LM y el marcador. En la figura 4.1, se muestran todos los sistemas de coordenadas existentes en el sistema de RA.

Para ajustar el sistema de coordenadas del LM al sistema de coordenadas del marcador se fija la posición de estos dos elementos dentro del sistema de RA. En la figura 4.2 se muestra la posición del LM y la distancia en milímetros a la que se encuentra del marcador.

En la figura 4.2, se observa que el origen del sistema de coordenadas del marcador se encuentra en el centro del marcador. Mientras que en el LM, el origen de su sistema de coordenadas se encuentra en el centro del mismo dispositivo sobre su cara superior. Se define una distancia constante entre los orígenes de los sistemas de coordenadas de ambos sistemas. Esa distancia es $[0, 10, 190]^T$, con respecto al LM. Además, en la figura 4.2 puede apreciarse que entre los sistemas de coordenadas existe una rotación de 90° en x.

Definimos el punto P_L como un punto en el sistema de coordenadas del LM y el punto P_M como un punto en el sistema del marcador. Con la información previamente mencionada se puede

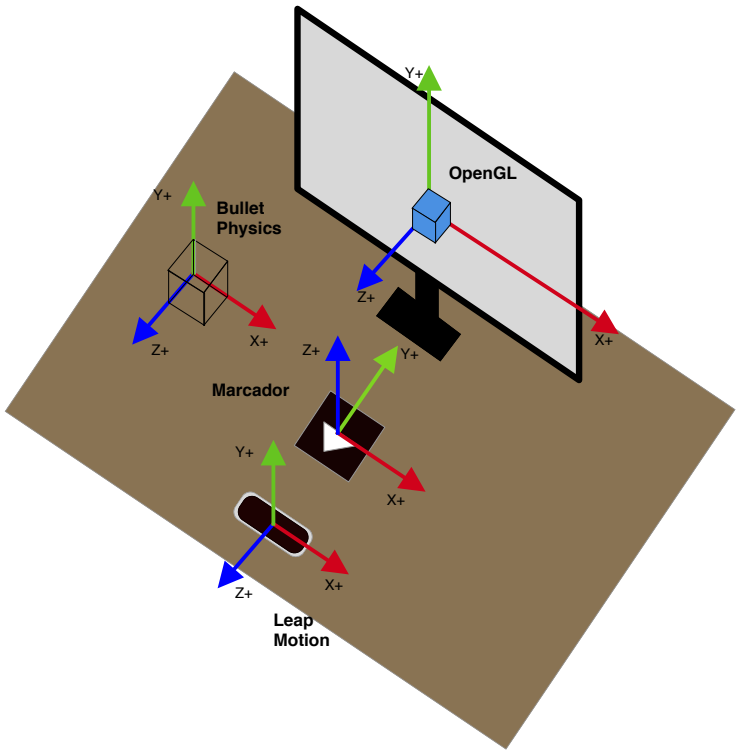


Figura 4.1: Sistemas de coordenadas presentes en el sistema de RA.

derivar la siguiente relación entre los puntos P_L y P_M.

$$P_M = R_x(90°)T(0, 11, 190)P_L$$

La matriz que define la transformación de un punto del sistema de coordenadas del LM al marcador es entonces:

$$M_{M \leftarrow L} = R_x(90°)T(0, 11, 190). \tag{4.1}$$

Esta matriz será usada para transformar las coordenadas que se leen del LM y de esta manera podrán usarse para la interacción con el marcador y los objetos virtuales que se dibujarán sobre él.

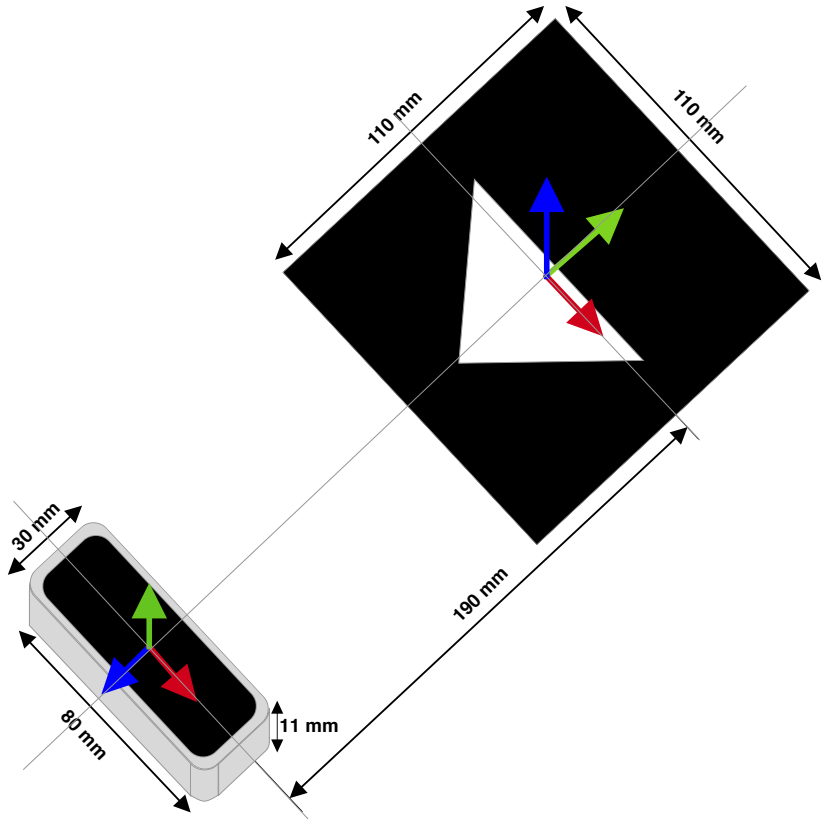


Figura 4.2: Distancias entre los sistemas de coordenadas del LM y el marcador. Las flechas rojas, verdes y azules representan los ejes $x+$, $y+$ y $z+$, respectivamente.

Prueba

En la figura 4.3 se muestra el punto de prueba P_L en coordenadas del sistema de coordenadas del LM y el mismo punto representado en coordenadas del marcador P_M.

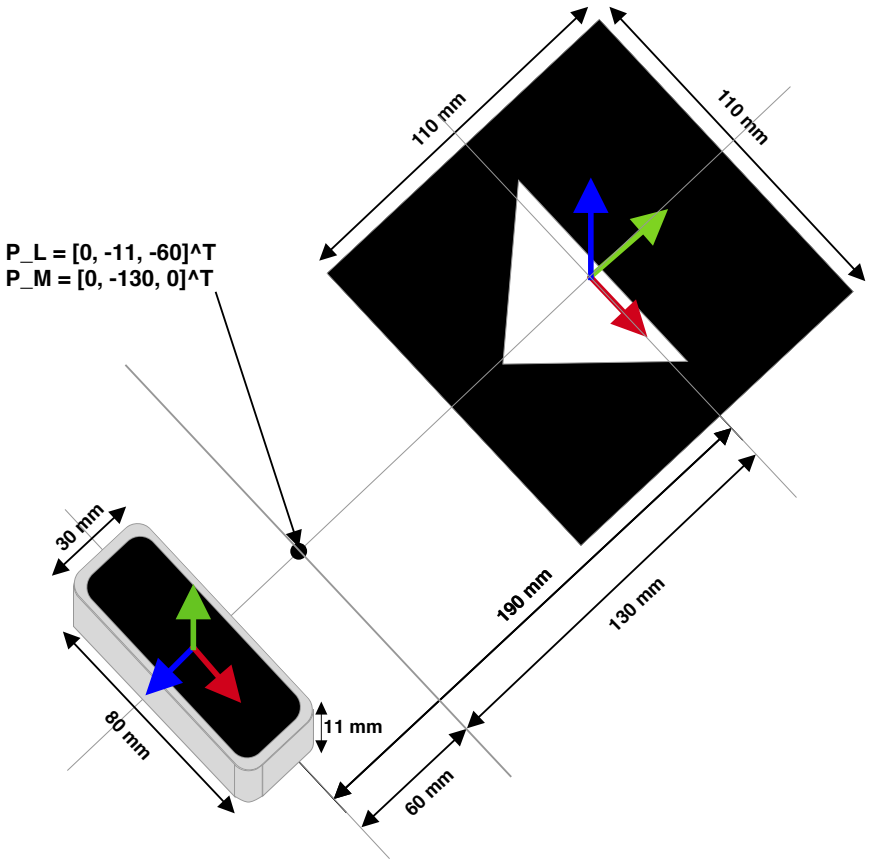


Figura 4.3: Punto de prueba $P_L = [0, -11, -60]^T$ y su representación en coordenadas del marcador $P_M = [0, -130, 0]^T$.

Aplicando la matriz de transformación $M_{M \leftarrow L}$ sobre el punto de prueba P_L tenemos que:

$$
\begin{aligned}
P_M = M_{M\leftarrow L}P_L &= R_x(90^\circ)T(0,11,190)P_L \\
&= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(90^\circ) & -\sin(90^\circ) & 0 \\ 0 & \sin(90^\circ) & \cos(90^\circ) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 11 \\ 0 & 0 & 1 & 190 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ -11 \\ -60 \\ 1 \end{bmatrix} \\
&= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 11 \\ 0 & 0 & 1 & 190 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ -11 \\ -60 \\ 1 \end{bmatrix} \\
&= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & -190 \\ 0 & 1 & 0 & 11 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ -11 \\ -60 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ -130 \\ 0 \\ 1 \end{bmatrix}
\end{aligned}
$$

Obtenemos el punto $P_M = [0, -130, 0]^T$ en el sistema de coordenadas del marcador, el cual coincide con el punto representado en la figura 4.3.

Haciendo la transformación del punto P_M a coordenadas del LM, podemos comprobar que la matriz de transformación es correcta. Para ello necesitamos la matriz inversa de la matriz de transformación $M_{M\leftarrow L}$.

$$
\begin{aligned}
P_L = [M_{M\leftarrow L}]^{-1}P_M &= T(0,-11,-190)R_x(-90^\circ)P_M \\
&= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -11 \\ 0 & 0 & 1 & -190 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(-90^\circ) & -\sin(-90^\circ) & 0 \\ 0 & \sin(-90^\circ) & \cos(-90^\circ) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ -130 \\ 0 \\ 1 \end{bmatrix} \\
&= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -11 \\ 0 & 0 & 1 & -190 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ -130 \\ 0 \\ 1 \end{bmatrix} \\
&= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -11 \\ 0 & -1 & 0 & -190 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ -130 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ -11 \\ -60 \\ 1 \end{bmatrix}
\end{aligned}
$$

De esta forma se obtiene el punto de prueba inicial $P_L = [0, -11, -60]^T$, por lo que se comprueba que la matriz de transformación $M_{M\leftarrow L}$ es correcta.

4.2. Visualización de los objetos

La visualización de los objetos en la escena del sistema de RA se logra a partir de la solución del modelo de la cámara obscura. Las matrices del modelo deben ser definidas en OpenGL dentro de la aplicación para que los objetos que dibujemos se vean sobre el sistema de coordenadas del marcador. Estas son la matriz K del proceso de calibración de la cámara y la matriz de rotación R que se obtiene de la estimación de la pose a partir de la calibración del modelo de la cámara obscura de la sección 2.1, en la pág. 11.

Para dibujar los objetos se utilizaron shaders de OpenGL, por lo que las matrices del modelo de la cámara obscura son definidas en los mismo. Estos están programados en el lenguaje GLSL (Gl Shader Language) que forma parte de OpenGL. Se hace uso de dos shaders: Uno llamado "Vertex Shader" (que se ejecuta por cada píxel) y otro llamado "Fragment shader" (que se ejecuta por cada muestra). A continuación de muestra el código del vertex shader, en donde se definen las matrices K y R.

```
#version 410
uniform mat4 kMat; // Matriz de calibración de la cámara
uniform mat4 RtMat; // Matriz de rotación
uniform mat4 modelMat; // Matriz del modelo
uniform mat4 orthoMat; // Matriz ortho

in vec3 vertexPosition;
in vec3 vertexNormal;
in vec2 vertexTexcoord;
in vec4 vertexColor;
out vec3 normal_eye, position_eye;
out vec2 texture_coords;
out vec4 color;
vec4 p;

void main()
{
  color = vertexColor;
  texture_coords = vertexTexcoord;
  normal_eye = normalize(vec3 (modelMat * vec4 (vertexNormal, 0.0)));

  p = kMat * RtMat * modelMat * vec4 (vertexPosition, 1.0); // Proyectamos el
    punto como en el modelo de la cámara obscura
  p = (1/p.w) * p; // Dividimos entre el factor de escala (lambda).
  position_eye = vec3 ( orthoMat * p ); // multiplicamos por la matriz ortho
  gl_Position = vec4 (position_eye, 1.0); // aquí se define la posición que será
    dibujada.
}
```

Listing 4.1: Vertex Shader

En el código 4.2 la matriz del modelo contiene las transformaciones del objeto (translación y rotación) que nos proporciona la biblioteca Bullet Physics, las cuales representan el comportamiento físico del objeto.

4.3. Uso de la biblioteca Bullet

La biblioteca Bullet Physics está diseñada para ser altamente configurable. Cada tarea principal que realiza esta biblioteca es encapsulado en su propio componente modular, permitiendo que sean reemplazados con facilidad siempre que se heredan de las clases adecuadas [14].

Existen varios componentes que deben ser creados y enlazados para inicializar Bullet. A continuación se describen brevemente los componentes más importantes.

4.3.1. El objeto mundo

Una instancia de la clase `btDynamicsWorld` es el objeto de control principal para una simulación de Bullet [14]. Todos los objetos con físicas serán controlados por las reglas definidas en esta instancia. Por defecto, esta clase define una fuerza de gravedad de -10 en el eje y, aunque este valor puede ser modificado utilizando el método `setGravity`.

4.4. Creación de un objeto con propiedades físicas

Existen tres componentes necesarios para crear un objeto con propiedades físicas en Bullet, los cuales son:

- `Collision Shape`: Define el volumen del objeto y cómo debe responder a las colisiones con otros objetos del mismo tipo.
- `Motion State`: Almacena el seguimiento del movimiento del objeto.
- `Collision Object`: Maneja a los dos componentes anteriores y las propiedades físicas del objeto.

4.4.1. El estado de movimiento

El estado de movimiento almacena la posición y orientación actual del objeto, esto nos permite utilizarla como medio para obtener la matriz de transformación del objeto. Además, se puede pasar esta matriz de transformación a OpenGL para actualizar el objeto gráficamente y que coincida con el objeto que define el comportamiento físico.

Este es el punto más importante de Bullet, ya que ésta no conoce ni se preocupa de cómo los objetos son representados. Se podrían añadir cientos de objetos a nuestra aplicación y Bullet se

encargaría de mover todos ellos, detectar colisiones, etc., pero tenemos que definir en OpenGL un objeto en la misma posición y con las mismas proporciones.

4.4.1.1. Obtener el estado de movimiento

Para obtener el estado de movimiento creamos una clase que herede de `btDefaultMotionState`. La función más importante dentro de esta clase es `GetWorldTransform()`, que retorna un arreglo que representa una matriz de 4×4. Esta matriz representa la rotación y translación del objeto basado en las reglas físicas del mundo [14]. La matriz debe ser consultada a cada paso de la simulación para proporcionar estos datos a OpenGL y que ésta lo dibuje. De esta manera, se verá que los objetos virtuales se mueven con un comportamiento similar a los objetos reales. Además, esta clase también nos proporciona la ubicación y el tamaño del objeto dentro del mundo.

Es relativamente sencillo representar los datos proporcionados por Bullet en OpenGL, ya que ambos usan el mismo sistema de coordenadas, basando en la regla de la mano derecha, que define las direcciones positivas de los ejes x, y y z [14].

4.4.2. El objeto colisión

El objeto colisión es el bloque constructor esencial del objeto, ya que mantiene las propiedades físicas del objeto [14]. Con este objeto podemos alterar la velocidad del objeto, su aceleración, aplicarle una fuerza, y demás propiedades. De este se obtiene la matriz de transformación que se consulta a través del estado del movimiento.

Los objetos más simples y comunes usan el tipo de objeto de colisión `btRigidBody`. Este tipo define un cuerpo rígido que no se deforman como resultado de una colisión, opuesto a los cuerpos blandos que sí lo hacen. Los cuerpos rígidos requieren del objeto `btRigidBodyConstructionInfo`, en el que se definen las propiedades de los cuerpos rígidos, como lo son la masa, la forma de colisión, etc.

4.4.3. Diseño de un objeto

Se pueden crear objetos de distintas formas como cajas, esferas, cilindros, etc. Para crear estos objetos debemos tener una instancia del objeto del estado de movimiento, una instancia de `btRigidBody` y una de `btRigidBodyConstructionInfo`. Esto es algo que hay que hacer siempre para cada objeto creado, por lo que solo tenemos que tomar en cuenta las propiedades únicas de cada objeto, las cuales son:

- la masa

- la forma
- el color
- la posición inicial y la rotación inicial

Por lo tanto, solo debemos crear una clase que reciba como parámetros estos datos y crear el resto de información sobre el objeto para poder acceder posteriormente a ella. Solamente tenemos que tener en consideración un parámetro más del objeto que debemos definir, la inercia local. Este valor normalmente es referido como el **momento de inercia**. Este se refiere a la resistencia del objeto al cambio en la velocidad angular alrededor de cada eje de rotación [14].

Se tiene un caso especial al crear un objeto. Si se define una masa cero, también debe definirse una inercia local a $(0, 0, 0)$. Esto evitará que el objeto rote como resultado de una colisión (crea un objeto estático). Para Bullet una masa cero implica que el objeto debe tener una masa infinita, y por lo tanto no se quiere que el objeto se mueva como resultado de la aplicación de una alguna fuerza, incluida la fuerza de gravedad [14]. Cualquier objeto con una masa mayor a cero debe definir una inercia local utilizando la función `calculateLocalInertia` que recibe como parámetro la masa, así el objeto se moverá como resultado de colisionar con otros objetos dentro de la simulación. Además, se verá afectado por la fuerza de gravedad del mundo.

Finalmente, los objetos deben ser agregados en el objeto mundo para que la biblioteca los tome en cuenta como parte de la simulación. Esto se hace por medio de la función `addRigidBody()` definida dentro de la clase del objeto mundo.

4.5. Definición de los objetos de la escena del sistema

Como ya se ha mencionado, la biblioteca Bullet Physics proporciona los algoritmos para la detección de colisiones y el comportamiento físico de los objetos virtuales. Para el sistema de RA se ha definido una caja que limita el área de trabajo de la escena. Dentro de esta caja vivirán los objetos virtuales, esto para que los objetos no salgan de la escena y por ende de la perspectiva de la cámara ya que si los objetos sobrepasan el campo de visión de la cámara estos se perderán.

En la figura 4.6 se muestra cómo se compone la escena del sistema de RA. El cubo azul es la caja que define los límites de la escena, la cual está compuesta de cuerpos rígidos estáticos ya que no son afectados por la simulación y sólo colisionan con el objeto que se encuentra dentro para impedir que salga de esa área. El cubo que tiene sus catas de distintos colores es el objeto virtual con el que el usuario va a interactuar. Este cubo es un cuerpo rígido dinámico que si es afectado por la fuerza de gravedad definida en el motor Bullet Physics. Finalmente, las dos esferas que se ven dentro del cubo azul son las que representan las puntas de los dedos índice y pulgar del usuario.

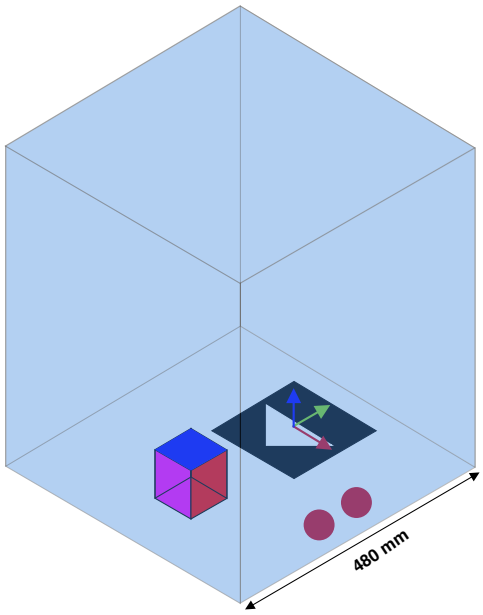


Figura 4.4: Escena del sistema de RA propuesta.

4.6. Uso del sensor Leap Motion

Las esferas de la figura 4.6 se mueven según los datos que proporcionan el dispositivo LM. Éstas son definidas como cuerpos cinemáticos en Bullet, lo que nos permite manipular su posición con base en la información del LM. Para actualizar la posición de las esferas se obtiene una instancia de la clase `Frame` de LM, se busca la posición de la punta de las falanges distales de ambos dedos en la clase `Finger`. Dentro de esta clase existe una variable llamada `btipPosition` que almacena justamente la posición de la punta de cada dedo. Esta posición es usada para especificar la posición del centro de la esfera.

Antes de definir la posición de las esferas con base en las coordenadas del LM, éstas deben ser transformadas a coordenadas del marcador con la matriz de transformación que definimos en el ajuste de los sistemas de coordenadas en la expresión (4.1). Las coordenadas que obtenemos después de la transformación serán las que definan la posición de las esferas.

4.7. Diseño del objeto deformable

El objeto deformable se implementa utilizando los cuerpos blandos de Bullet Physics. Para crearlo utilizamos una clase de Bullet llamada `btSoftBodyHelpers` que contiene algunas funciones para facilitarnos la creación de este objeto. Se utilizó la función `CreateEllipsoid()` con la que se puede crear una esfera compuesta por triángulos, aunque la función crea una elipsoide si se definen todos sus lados del mismo tamaño ésta será una esfera.

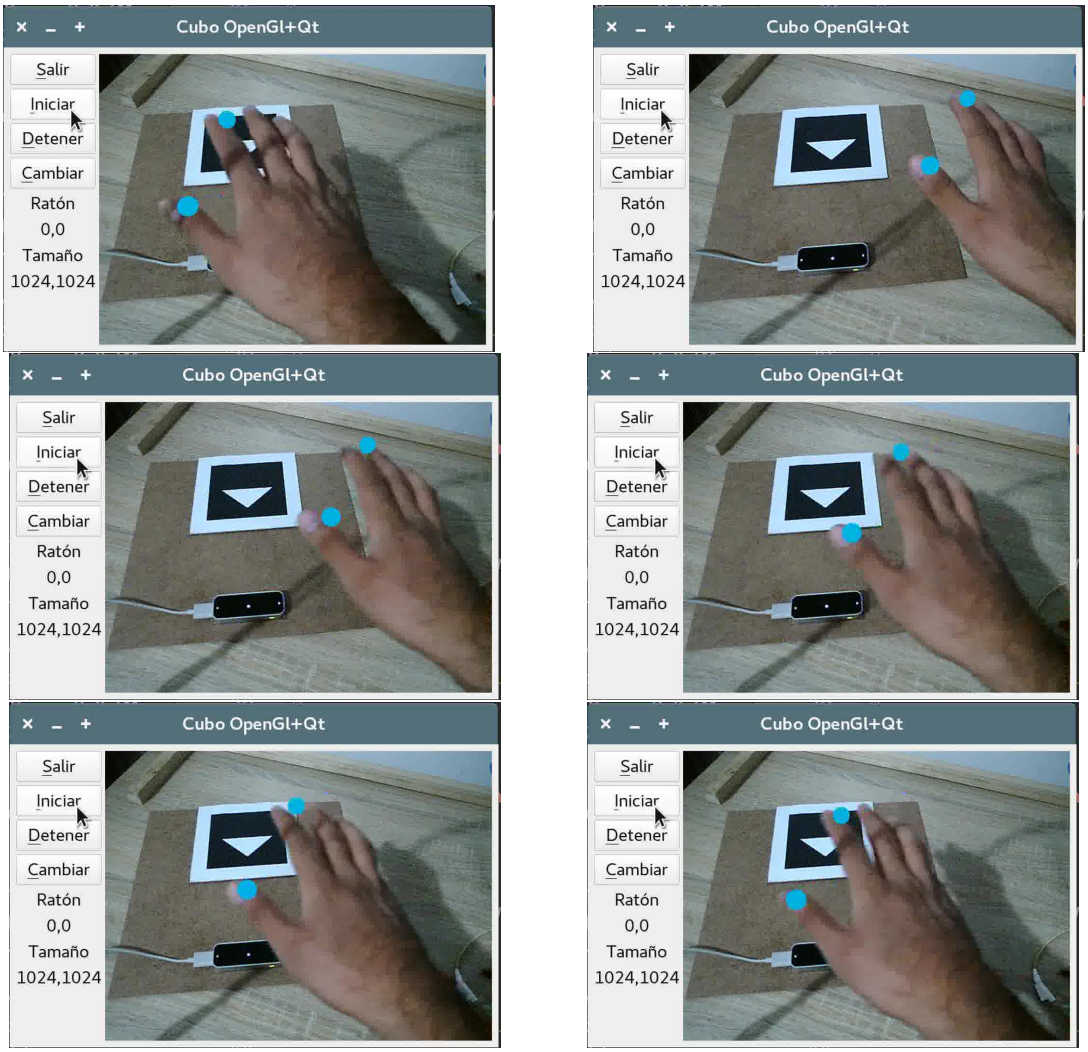


Figura 4.5: Movimiento de las esferas utilizando la información de seguimiento del LM.

```
btSoftBody* pSoftBody =
btSoftBodyHelpers::CreateEllipsoid
(m_softBodyWorldInfo,btVector3(0,0,0),btVector3(3,3,3),128);
m_pSoftBodyWorld->addSoftBody(pSoftBody);
```

Listing 4.2: Vertex Shader

Para definir el comportamiento del objeto deformable hay dos parámetros claves: el coeficiente de conservación de volumen y su rigidez lineal. Cada uno de estos parámetros altera qué tanto el objeto puede mantener su forma cuando se mueve y cuando colisiona con otros objetos [14]. Además, de estos dos parámetros también debemos definir la masa total del objeto con la función `setTotalMass()`. La masa del objeto también tiene un impacto importante en la forma en que el objeto se deforma cuando colisiona.

Finalmente, se debe usar la función `setPose()` para que se generen las restricciones del cuerpo blando y de está forma mantenga la posición y orientación de cada vértice que la compone [14].

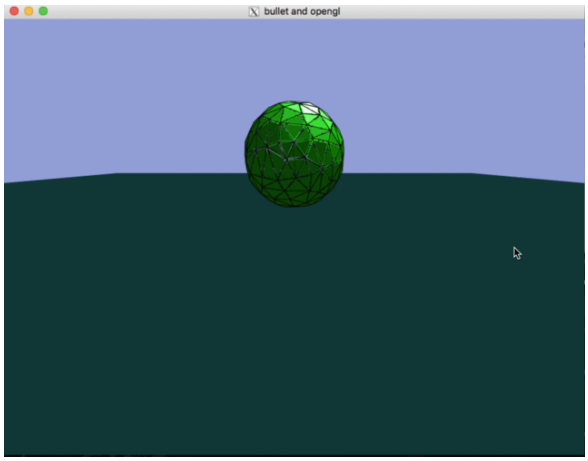


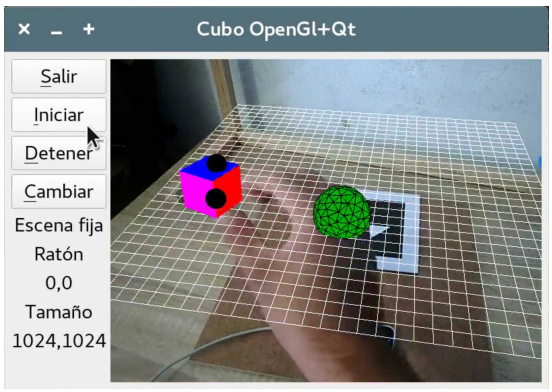
Figura 4.6: Objeto deformable generado con Bullet Physics creando una esfera con la función `CreateEllipsoid()`.

4.8. Interacción del usuario con los objetos

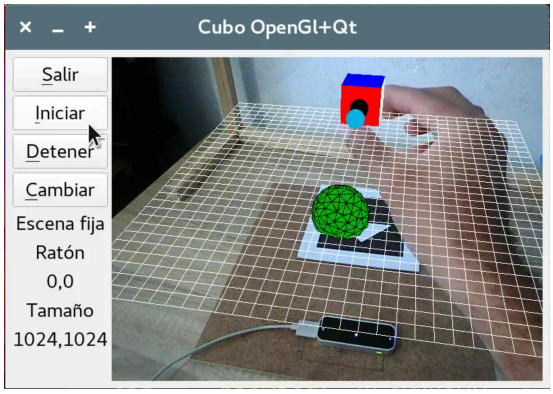
En el sistema, el usuario podrá empujar tanto los objetos rígidos como los objetos deformables. Además, podrá sostener con ambos dedos a los objetos rígidos, con un gesto implementado donde se detecta cuando ambos dedos están en contacto con el objeto como muestra la figura 4.7.

La figura 4.7 describe el proceso a realizar por el usuario para poder sostener un objeto rígido. Primero, el usuario debe hacer contacto con el objeto usando ambos dedos como haciendo un pellizco al objeto. El usuario podrá transportar el objeto a la posición que desee, dentro de los límites del sistema de RA mencionados en la sección 4.5, siempre que mantenga los dedos juntos. Por último, el objeto puede soltarse y caer a la superficie del sistema simplemente separando los dedos.

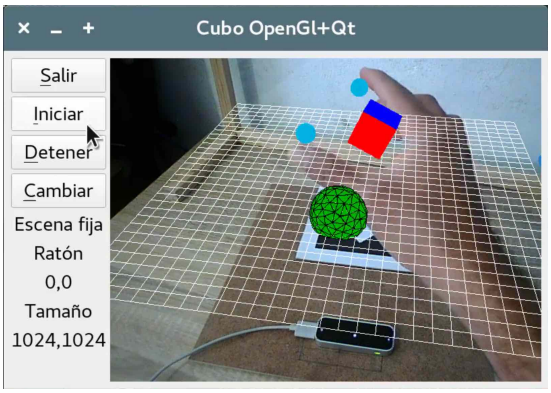
En cuanto al objeto deformable, el usuario puede empujarlo usando cualquiera de los dedos (pulgar y meñique) como se muestra en la figura 4.8. Aquí se puede apreciar una esfera formada por triángulos como nuestro objeto deformable. El usuario puede tocar el objeto deformable con cualquiera de los dos dedos definidos dentro del sistema y se podrá observar como se sume la superficie del objeto justamente donde el dedo está haciendo contacto. La figura 4.8 muestra seis imágenes que muestran dicha interacción, desde el momento previo antes del contacto hasta cuando el contacto dejo de existir.



(a) Se sostiene el cubo haciendo sujetándolo con ambos dedos juntos.



(b) Se puede mover el objeto manteniendo los dedos juntos.



(c) Separando los dedos soltamos el objeto, el cual cae a la superficie definida dentro del sistema.

Figura 4.7: Gesto para sostener un objeto rígido dentro del sistema.

4.8.1. Resumen

En este capítulo se presentó el proceso de implementación del sistema, comenzando con el ajuste de los sistemas de coordenadas del LM y del marcador. Se mostró la visualización de los objetos virtuales dentro de la escena del sistema. También, se describió el uso de la biblioteca Bullet Physics, que utilizamos para proporcionar a los objetos de un comportamiento físico realista, para implementar el detector de colisiones y para crear el objeto deformable. Por último, se mostró como se da la interacción del usuario con los objetos virtuales.

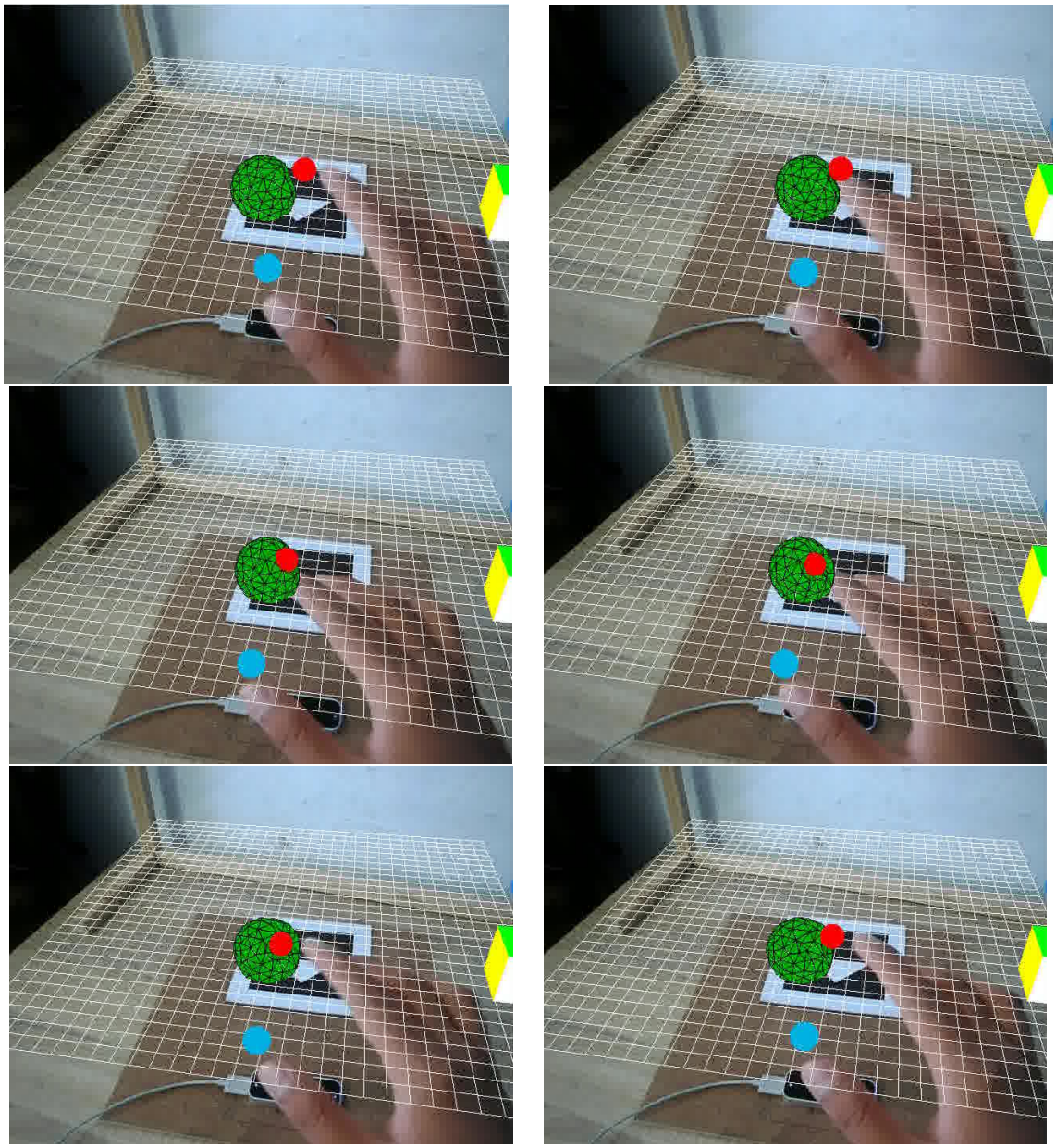


Figura 4.8: Interacción del usuario con el objeto deformable.

Capítulo 5

Metodología de desarrollo y validación del sistema

En este capítulo se presenta la metodología de desarrollo y validación del sistema. Se ha dividido el sistema en cuatro módulos que comprenden las etapas principales del sistema y se han marcado sobre el diagrama general del sistema. En cada sección del capítulo se presenta un modulo y las validaciones que se llevaron a cabo en esa etapa para asegurar el correcto funcionamiento del sistema.

El sistema fue construido de forma incremental, buscando siempre contar con una interfaz gráfica para probar que el funcionamiento parcial del sistema era correcto. De esta forma fue sencillo ir revisando incrementalmente el sistema, se tenía un sistema con una interfaz gráfica donde la validación consistió en revisar interactivamente su correcto funcionamiento.

Estos sistemas parciales realizaban:

1. Interacción con un objeto virtual.

2. El punto (1) y con la detección de colisiones.

3. El punto (2) y con la simulación basada en física con objetos cinemáticos de la biblioteca Bullet.

4. El punto (3) y la incorporación de la interacción usando las coordenadas que provee el LM, y finalmente:

5. El punto (4) más un objeto deformable de la biblioteca Bullet.

Cada subsistema tuvo que revisarse varias veces, en etapas de corrección y pruebas –usando su interfaz gráfica– hasta lograr su correcto funcionamiento.

El funcionamiento de cada uno de estos subsistemas se describirá brevemente y también la teoría que se tuvo que estudiar antes de su realización.

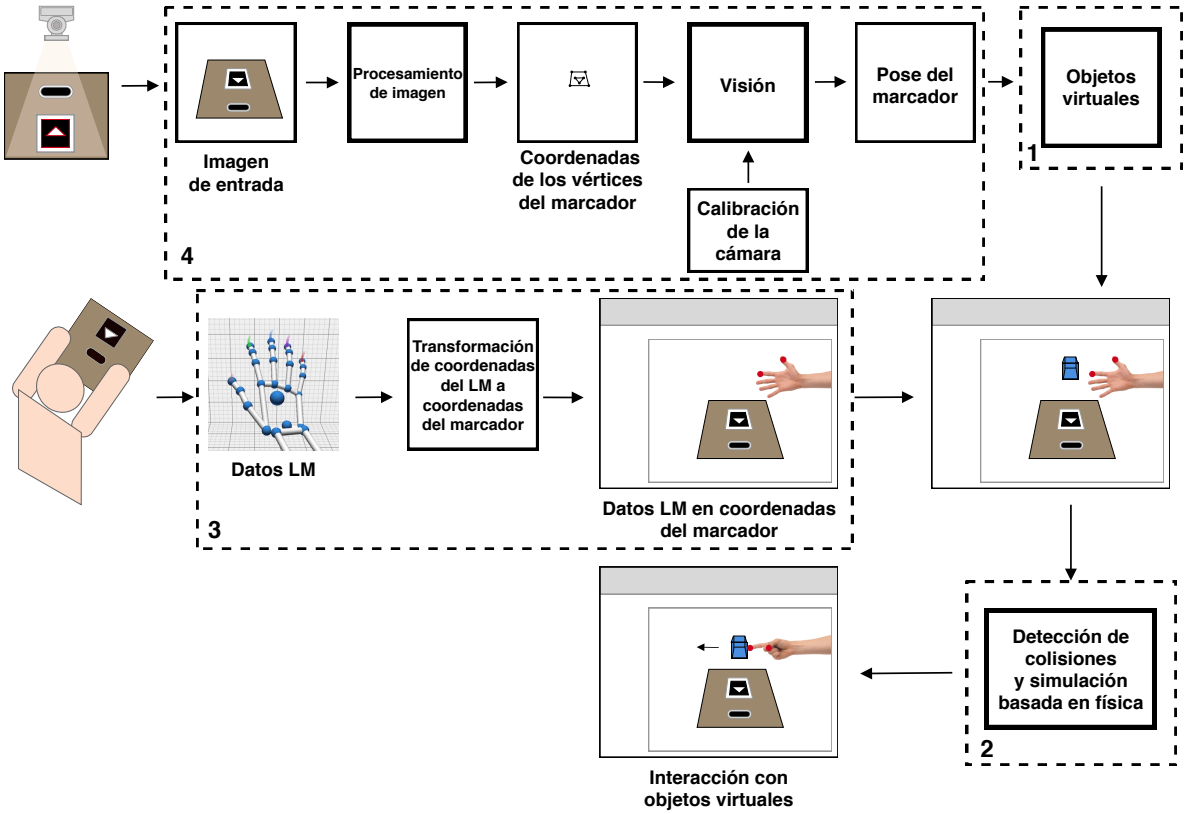


Figura 5.1: Diagrama del sistema de RA, en donde los cuadrados con líneas punteadas destacan los módulos que comprenden las etapas principales del sistema.

5.1. Interacción con un objeto virtual

Al inicio del trabajo de tesis se diseñó un cubo virtual (el diseño se muestra en el apéndice C). El cubo está compuesto por 8 vértices y 12 triángulos (cada una de las 6 caras del cubo se compone de dos triángulos), y cada cara tiene un color distinto. Para ahorrar memoria se utilizaron índices a la lista de vértices para especificar las caras. Se ahorra memoria porque cada vértice se usa en más de una cara y sólo se almacena una vez.

El cubo se mostró en la interfaz gráfica construida con Qt y OpenGL, ya descritos en las secciones 3.3.1 y 3.3.3, en las páginas 39 y 41 de esta tesis, respectivamente.

Se usó la versión 4.1 de OpenGL, esto obliga a usar los constructores denominados *shaders*.

Es muy importante el diseño de este cubo mostrado en el apéndice C, en la página 93, la

interfaz gráfica debe mostrar exactamente el mismo cubo diseñado. De lo contrario, se tiene algún error en la transcripción de los valores de los vértices, de los índices de las caras o en la especificación del color de alguna cara. El diseño permite validar su correcta implementación, ya que ambos deben ser iguales.

La interfaz gráfica en esta primera etapa, mostrada en la figura 5.2, tenía la siguiente funcionalidad:

- Botón salir: al presionarlo se cierra la aplicación.
- Botón iniciar: al presionarlo imprime en consola el nombre del botón para verificar que fue presionado.
- Botón detener: al presionarlo imprime en consola el nombre del botón para verificar que fue presionado.
- Si se presionaba el ratón y al mismo tiempo se movía dentro del widget de OpenGL se podía rotar interactivamente el cubo: al mover sobre el eje x de la ventana se rota +/- 180 grados sobre el eje y del mundo tridimensional. Al mover sobre el eje y de la ventana se rota +/- 180 grados sobre el eje x del mundo tridimensional.
- Se tiene en la aplicación un widget donde se imprimen las coordenadas del cursor del ratón mientras mantenemos presionado su botón izquierdo.
- Por último, la ventana de la aplicación puede cambiar de tamaño estirándola al mantenerla presionado con el botón derecho del ratón, como con lo hacemos con cualquier otra ventana, dentro de la interfaz se muestra una etiqueta (un widget de Qt) que indica el tamaño de la misma.

Todas estas funcionalidades se revisaron exhaustivamente.

5.2. Detección de colisiones y simulación basada en física

La detección de colisiones y simulación basada en física comprende el modulo marcado con un cuadrado con líneas punteadas etiquetado con un '2', mostrado en la figura 5.1, en la pág. 60.

Se utilizó por primera vez la biblioteca Bullet Physics, con la cual se implementó un cubo que caía simplemente por efecto de la fuerza de gravedad. Esto nos permitió aprender a utilizar la matriz de transformación que nos proporciona Bullet para dotar a un objeto gráfico en OpenGL con un comportamiento físico. Después, se agregó una superficie fija, utilizando los cuerpos estáticos de Bullet, para que los demás objetos cayeran sobre ella y no se perdieran de vista como mostramos en la figura 5.3.

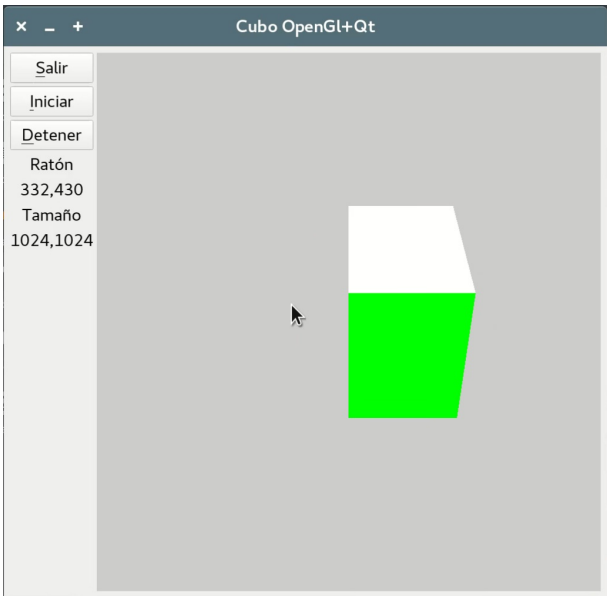


Figura 5.2: Interfaz gráfica de la aplicación de RA.

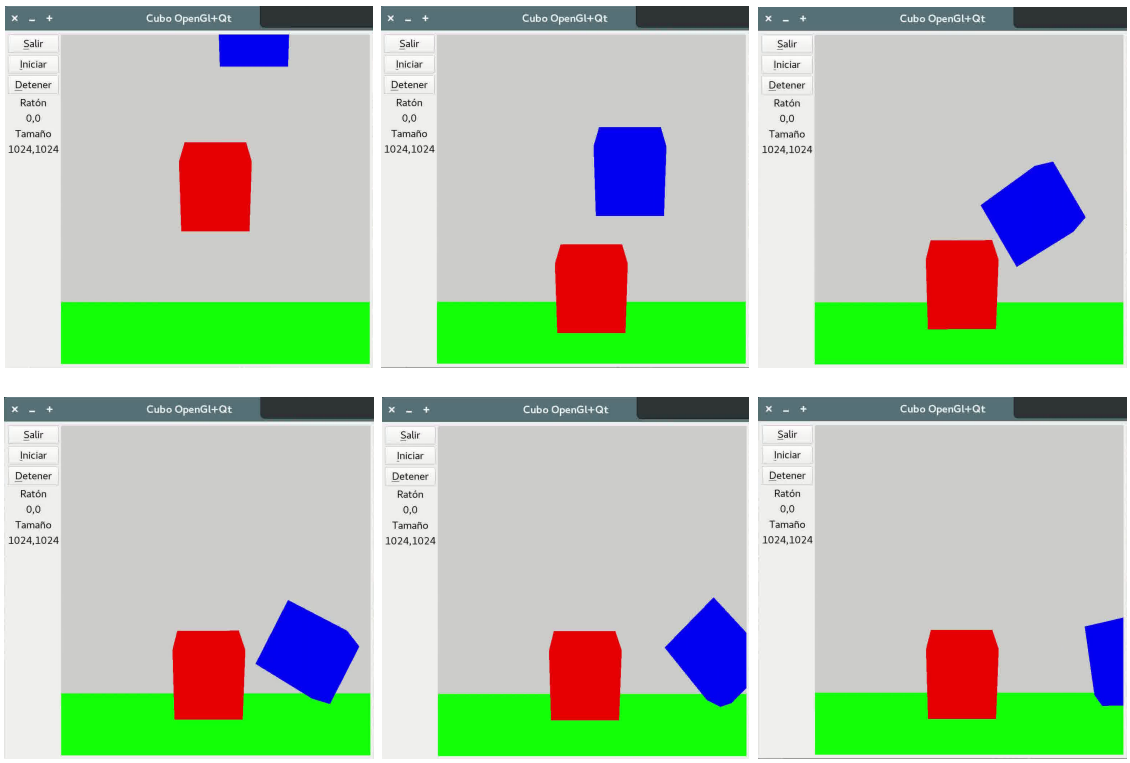


Figura 5.3: Simulación utilizando Bullet Physics donde dos cubos caen por el efecto de la fuerza de gravedad sobre una superficie fija.

Se cambió la posición de la cámara en la prueba anterior con lo que notamos que la escena no se veía correctamente, dónde se observaba que los cubos no colisionaban como lo harían dos cubos

en el mundo real. Para tener más información de lo que ocurría se implementó un depurador para dibujar las formas de colisión de los objetos de Bullet, el cual dibujaba sólo el contorno de las cajas usadas en Bullet. Con el depurador pudimos apreciar que los objetos dibujados no coincidían con las formas de colisión definidas en Bullet, como se observa en la figura 5.4.

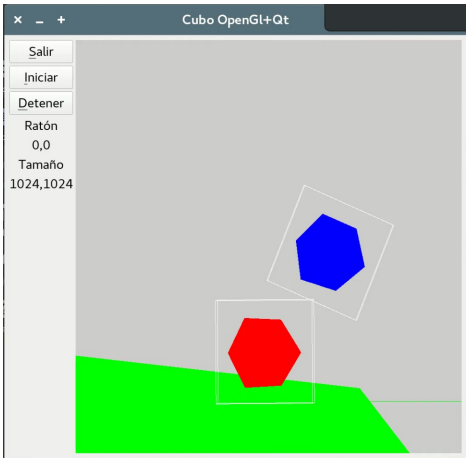


Figura 5.4: Los objetos dibujados no coinciden en la posición de las formas de colisión definidas por Bullet.

El depurador dibujaba correctamente los objetos definidos en Bullet, pero estaba programado en una versión de OpenGL que no utilizaba shaders, mientras que el cubo que dibujamos sí estaba utilizando shaders. Por lo tanto, configuramos el widget de OpenGL de la interfaz para que sólo utilizará la versión 4.1 con la que fueron programados el fragment y vertex shader. Una vez que el widget fue configurado para la versión 4.1 y todo se dibujaba utilizando shaders, incluida la información que se dibujaba en el depurador, los objetos dibujados y las formas de colisión coincidieron viéndose de forma correcta. En la figura 5.5 se muestra una prueba de esto, en donde además agregamos una esfera para visualizar su comportamiento al colisionar con un cubo.

El desplazamiento de los objetos del sistema fue delimitado dentro de un cubo que envuelve toda la escena, ya que si se mueven más allá de la superficie fija que los sostiene éstos caerán infinitamente y dejarán de ser visibles para el usuario. Las paredes de este cubo envolvente se definieron de la misma forma que la superficie fija que los sostiene, que también es una cara del cubo envolvente, pero éstas no son visibles para el usuario, sólo están ahí para colisionar con los objeto e impedir su desplazamiento más allá de la superficie fija.

Luego se agregó una esfera a la escena que utilizamos para definir la punta de los dedos de la mano usando los datos del LM. La esfera tiene un comportamiento distinto al del cubo de colores, su desplazamiento depende del movimiento de las manos del usuario, es decir, la esfera se moverá según los datos obtenidos del LM. Revisando el manual de Bullet nos dimos cuenta que la esfera debería ser del tipo cinemático, que permite que el objeto sea manipulado por el usuario e ignora el efecto que produce la fuerza de gravedad. Así que, para probar la colisión de la esfera con el cubo definimos en el código su movimiento, desplazándolo en dirección al cubo actualizando su posición a cada paso de la simulación hasta llegar al límite de la escena. Para detectar el momento en el que ocurre una colisión la esfera cambia de color. Al inicio, la esfera

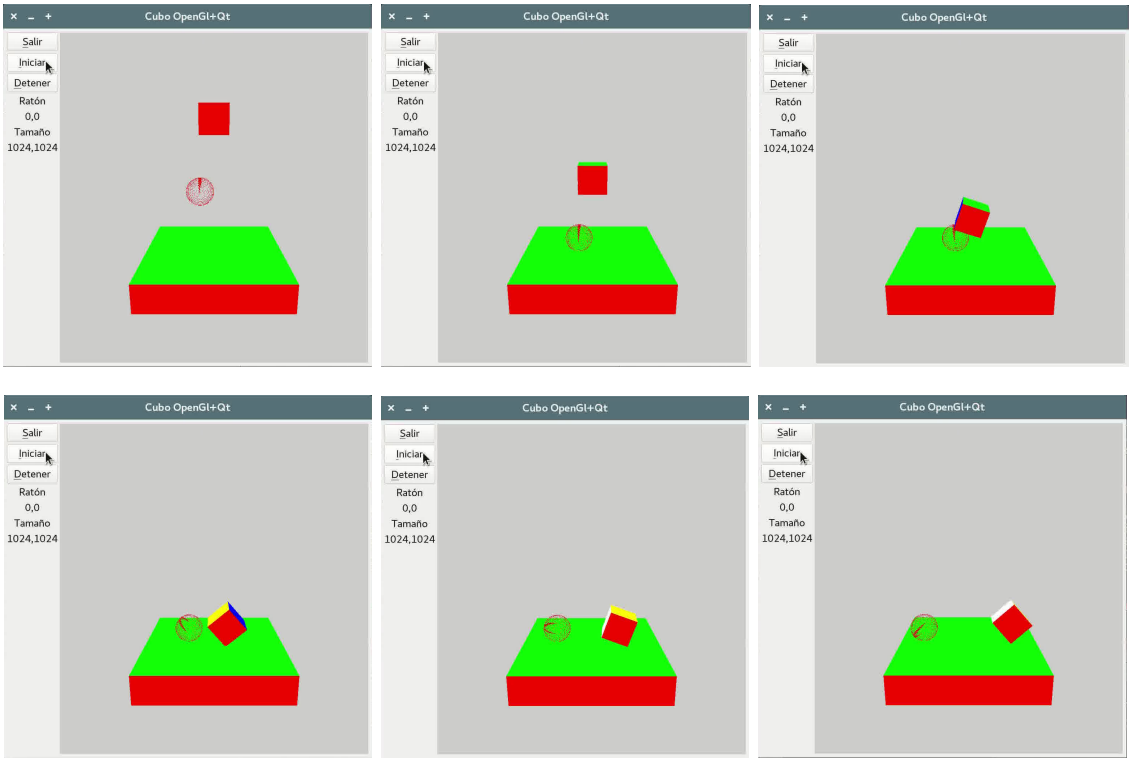


Figura 5.5: Simulación donde un cubo y una esfera caen por el efecto de la fuerza de gravedad y colisionan entre sí al llegar a la superficie fija.

es de color azul. Cuando existe una colisión la esfera es de color negro y se mantiene así siempre que la colisión siga existiendo. Una vez que ya no hay contacto entre los objetos, el cubo vuelve a ser de color azul. Además, para distinguir cuando el cubo alcanzo los límites de la escena la esfera pasa a ser de color rojo. Todo esto se ilustra en la figura 5.6.

Por último, se colocó el cubo de colores junto con un dibujo del marcador y uno del LM para mostrar cómo se vería la escena del sistema, sin usar el marcador aún, como se muestra en la figura 5.7. Esto para tener una idea de la perspectiva que tendría el usuario y saber cómo se debe ver la escena cuando agreguemos el marcador.

5.3. Datos del LM en coordenadas del marcador

El uso del dispositivo LM comprende las etapas dentro del cuadrado con líneas punteadas etiquetado con un '3', de la figura 5.1.

Comenzamos a trabajar con el dispositivo LM, agregamos el archivo de cabecera del dispositivo `Leap.h`. Este archivo nos permite definir un objeto de tipo `Controller` que sirve como conexión al servicio del LM y con él podremos tener acceso a un objeto `Frame` del LM, de dónde podemos

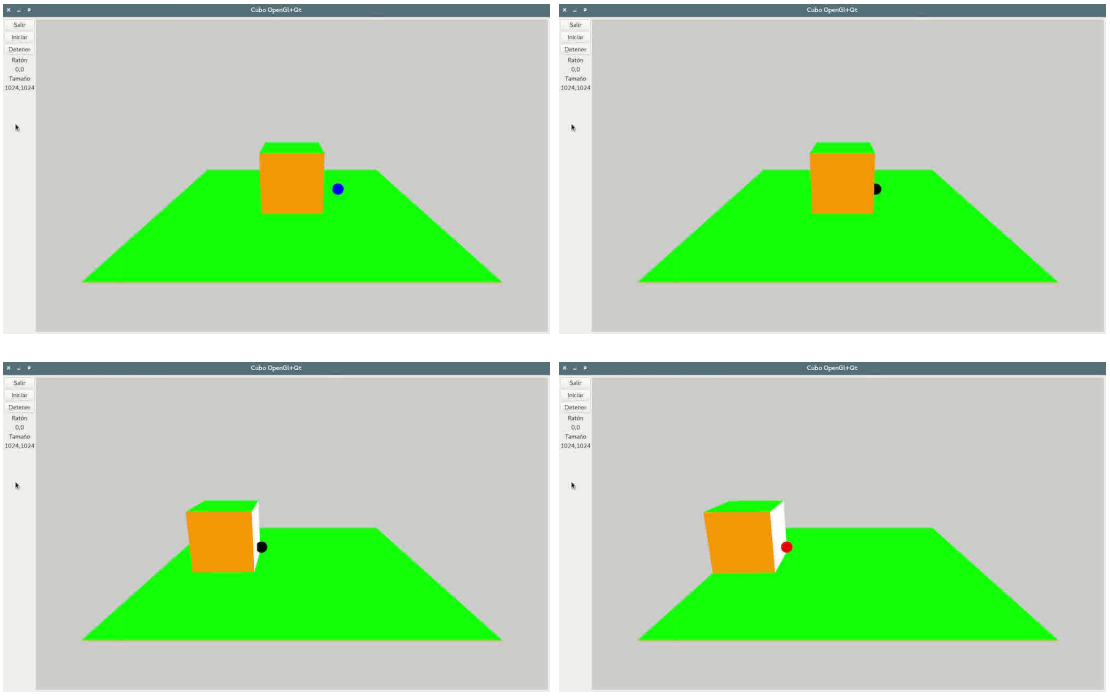


Figura 5.6: Prueba de la colisión de la esfera con un cubo

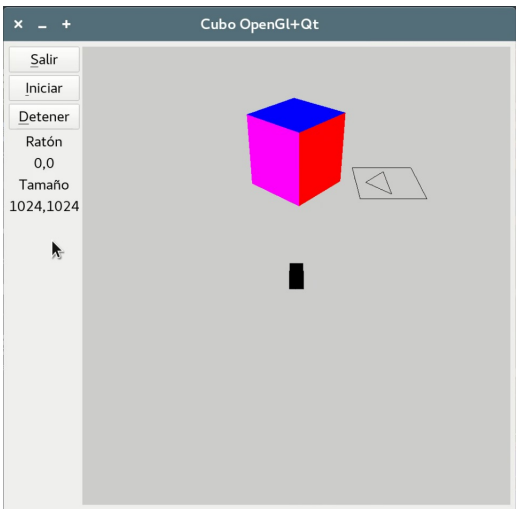


Figura 5.7: Escena virtual que muestra el cubo de colores girando colocado frente al dibujo del marcador, debajo del cubo se muestra un dibujo del LM.

obtener la información que ofrece el dispositivo tal como se describió en la sección 3.3.5 en la pág. 42. Primero probamos mover la esfera que definimos en la escena con las coordenadas que nos entregaba el dispositivo, sustituyendo el desplazamiento que habíamos programado para la esfera por la posición de la punta del dedo índice de la mano derecha del usuario. Esto nos permitió probar la colisión de la esfera con el cubo en distintas direcciones y verificar que todas las paredes que definimos en los límites efectivamente impedían que el cubo de colores saliera de

la escena. La figura 5.8 muestra algunas imágenes del movimiento de la esfera con los datos del LM y la colisión con el cubo.

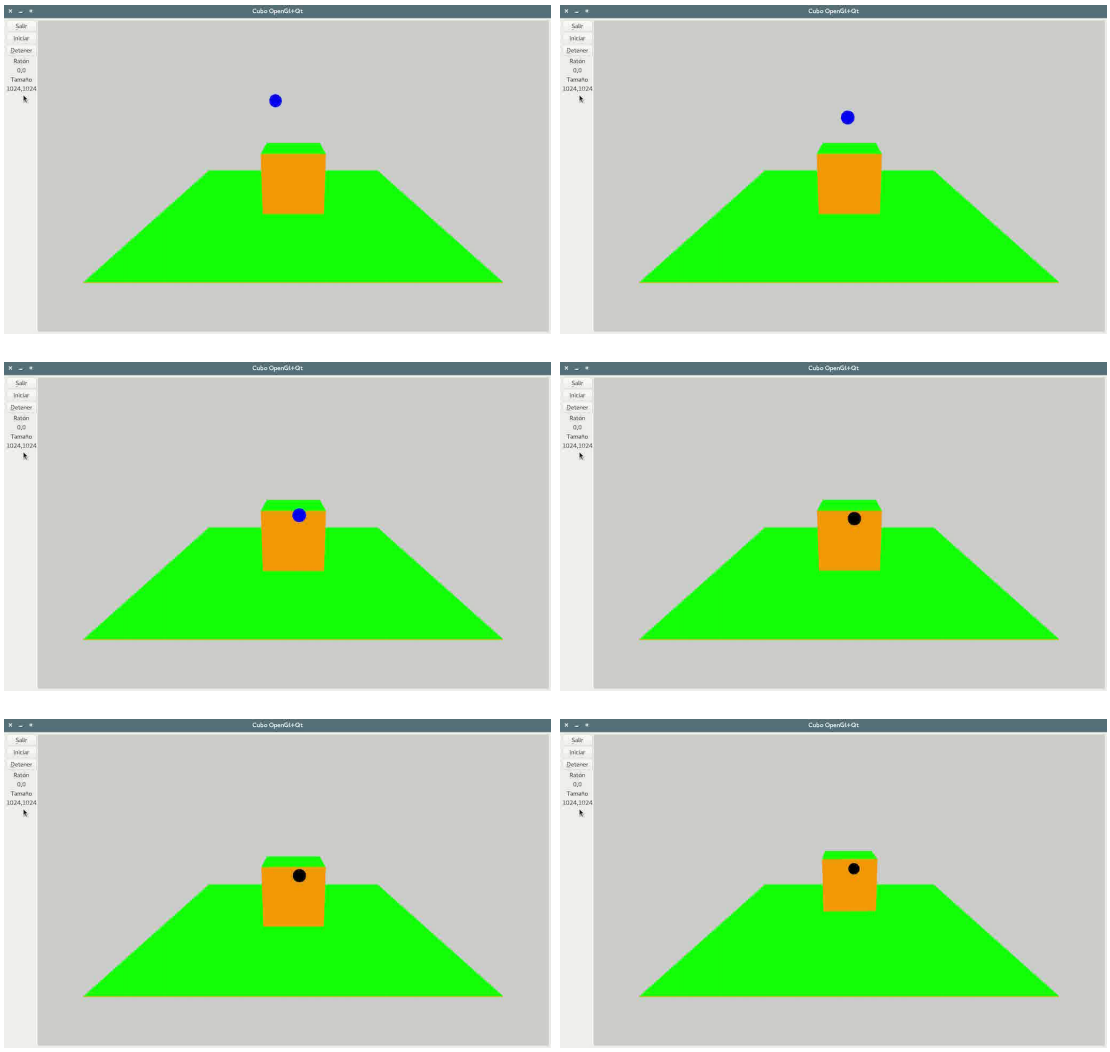


Figura 5.8: Se observa como la esfera ahora puede tomar otras posiciones en la escena, ya que se mueven según la posición de la mano del usuario. Ahora puede probarse la colisión con la cara del cubo que vemos de frente, lo cual se aprecia cuando la esfera se vuelve de color negro y también se observa como el cubo es desplazado como resultado de la colisión.

Hicimos pruebas de las formas en que el usuario podía interactuar con el cubo. Primero, intentamos levantar el cubo utilizando dos dedos (índice y pulgar), por lo que definimos una segunda esfera en la escena. La superficie sobre la que están los objetos se cambió a color gris para distinguir mejor el cubo y las esferas. En esta prueba nos dimos cuenta que era complicado poder levantar el cubo, dado que las esferas en lugar de sostener el cubo lo empujaban como resultado de la colisión de ambas esferas con él, como se muestra en la figura 5.9.

Decidimos agregar un tercer dedo (el dedo medio) para intentar levantar el cubo. Con esto logramos levantar el cubo colocando uno de los tres dedos debajo del cubo y usando los otros

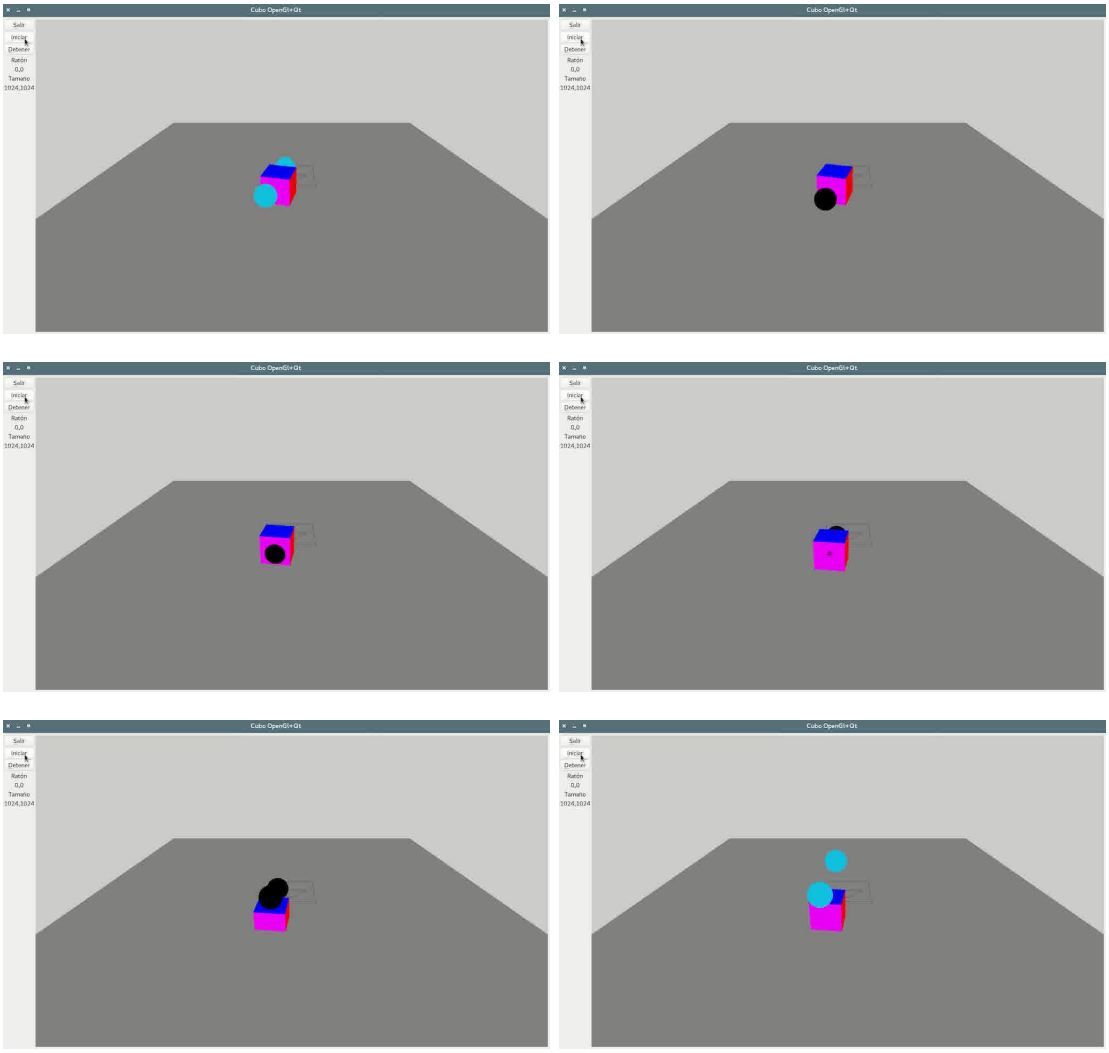


Figura 5.9: Se muestra como se hizo la prueba para levantar el cubo utilizando ambos dedos definidos en el sistema, en donde el cubo no puede sostenerse y solamente se desplaza un poco al momento de la colisión con ambas esferas. Al levantar la mano, se observa como se elevan las esferas pero el cubo se mantiene en la superficie fija.

dos para evitar que se cayera por los costados, aunque seguía siendo complicado de hacer. En la figura 5.10 se muestra una prueba para levantar el cubo con tres dedos.

Nosotros queríamos que para el usuario no fuera complicado levantar el cubo, por lo que decidimos implementar un gesto que nos permitiera usar solo el dedo índice y pulgar para levantar y desplazar el cubo con facilidad. Logramos implementar el gesto utilizando `constraint` de Bullet Physics, tomando la idea que describen en el libro de Chris Dickinson [14] para sostener y mover objetos en una aplicación apuntándolos con el cursor del ratón. Con este elemento, como su nombre lo indica, podemos crear restricciones que limita el rango de movimiento de un objeto con respecto a otro o con respecto a un punto en el espacio [14]. Un uso común de este elemento es la definición de objetos que se componen de distintas partes, por ejemplo una silla que se

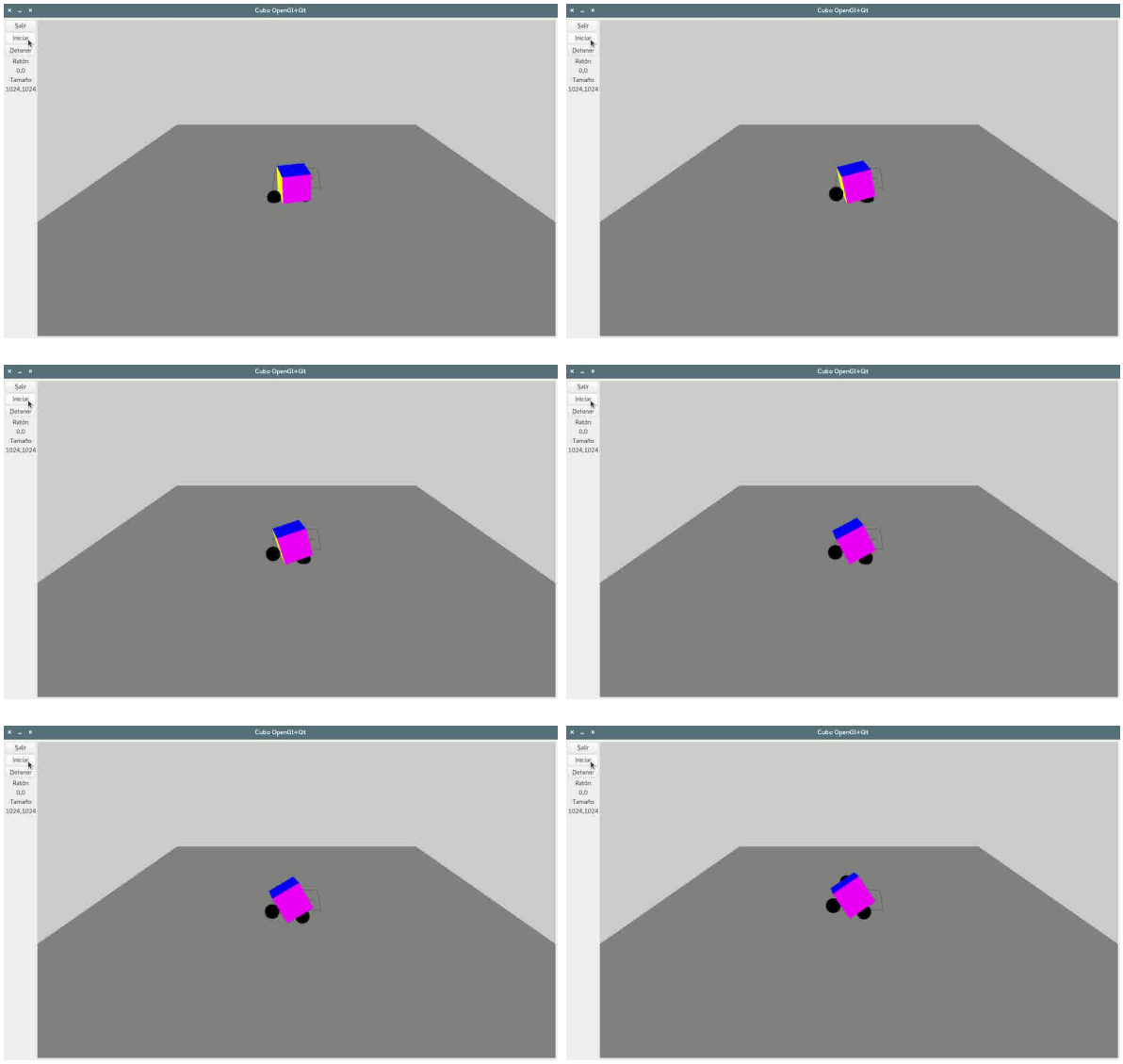


Figura 5.10: Se muestra como se logra levantar el cubo utilizando el dedo índice, pulgar y medio para ello, colocando el dedo medio por debajo del cubo y tratando que el cubo no caiga sosteniéndolo por los costados con el dedo índice y pulgar.

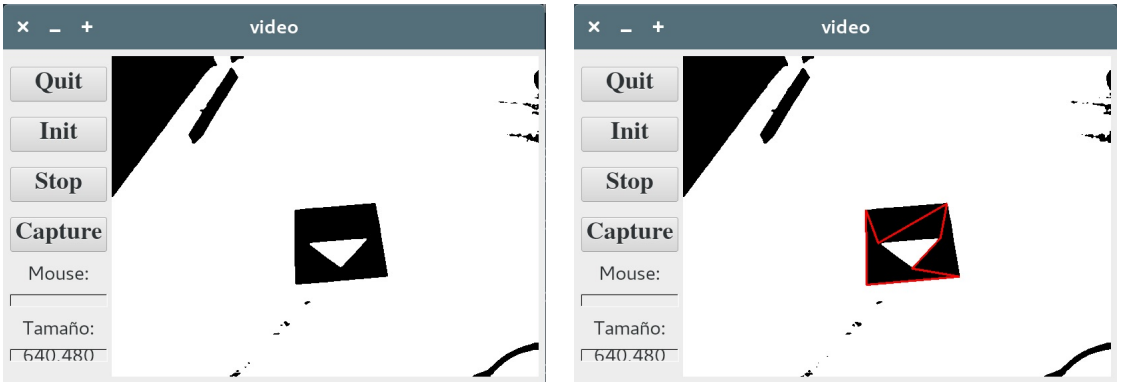
componen del asiento, el respaldo, las patas, etc., ya que si alguna de sus partes se mueve todas las otras partes también deberían moverse con ésta. De esta manera, lo que el gesto hace es definir una restricción temporal que une las esferas con el cubo al estar en contacto, logrando que se muevan juntos, siempre y cuando ambas esferas se mantengan juntas. Una vez que las esferas se separan, lo cual detectamos midiendo la distancia entre ambas esferas y comprobando si es mayor a un umbral, la restricción se elimina para que el cubo vuelva caer por el efecto de la fuerza de gravedad, tal como se mostró en la figura 4.7 en la pág. 57.

5.4. Detección del marcador

Con la escena y el uso del dispositivo implementados en nuestra aplicación, incluimos el marcador de RA en nuestro sistema para que la escena se visualice sobre el mundo real. Los pasos para realizar la etapa para detectar el marcador se visualizan en la figura 5.1, con el cuadrado con líneas punteadas etiquetado con un '4'.

Es este punto comenzamos a utilizar la cámara PS Eye para obtener las imágenes que procesaríamos en la detección del marcador. La imagen de la cámara se colocó como fondo en el widget de OpenGL en lugar del fondo gris claro que se tenía, sobre la cual proyectaremos la escena que trabajamos en las secciones anteriores.

Primero, agregamos el marcador al proyecto junto con el algoritmo para detectar el marcador. Tuvimos un problema con el tamaño de la imagen de la cámara, que era de 1280×960, ya que no se detectaban los vértices del marcador. Esto ocurría debido a que el algoritmo requería de una imagen de tamaño 640×480. El problema fue corregido simplemente cambiando el tamaño de la imagen. Para verificar que los vértices obtenidos correspondían a los vértices del marcador se dibujó una línea que une los 7 vértices detectados encima de la imagen, como mostramos en la figura 5.11.



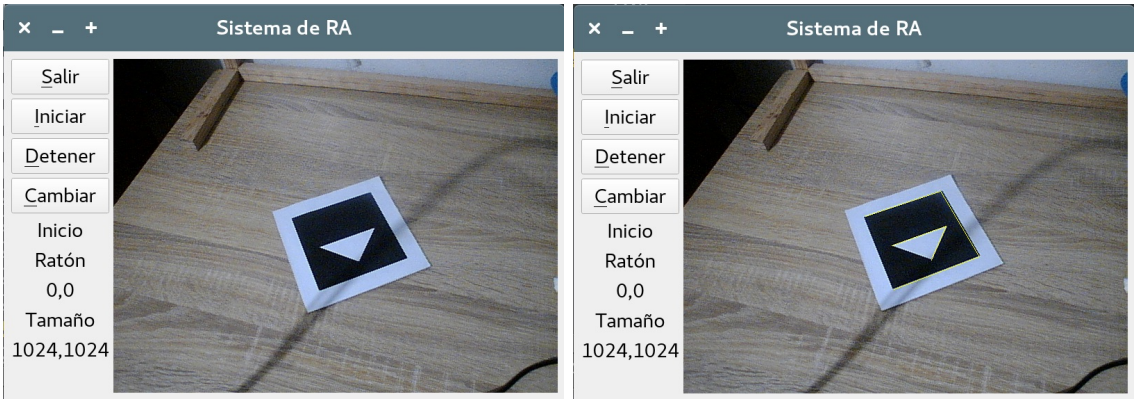
(a) Imagen del marcador tomada por la cámara.

(b) Vértices detectados marcados con líneas rojas sobre la imagen del marcador.

Figura 5.11: Imagen del marcador y vértices encontrados en el proceso de detección.

Teniendo los vértices del marcador se puede calcular la rotación y translación para dibujar el marcador a partir del modelo del mismo. Para realizar esto es necesario tener por anticipado la cámara calibrada. La matriz de calibración de la cámara y la matriz de rotación y translación se definen dentro del shader, como se indicó en la sección 4.2 en la página 50. Como prueba proyectamos los vértices del modelo del marcador, los cuales deben verse sobre los vértices del marcador en la imagen como se ve en la figura 5.12.

Al aplicar el modelo de la cámara obscura sobre los vértices del cubo de colores podremos visualizar dicho objeto sobre el sistema de coordenadas que define el marcador como se observa en la figura 5.13. En esta figura se observan cuatro imágenes en donde se rotó el marcador para



(a) Imagen del marcador

(b) Imagen del marcador con los vértices del modelo del marcador proyectados sobre ella.

Figura 5.12: Imagen del marcador e imagen con los vértices del modelo del marcador proyectados.

poder apreciar todas las caras del cubo, con excepción de la cara verde que es la cara inferior del cubo que no puede verse.

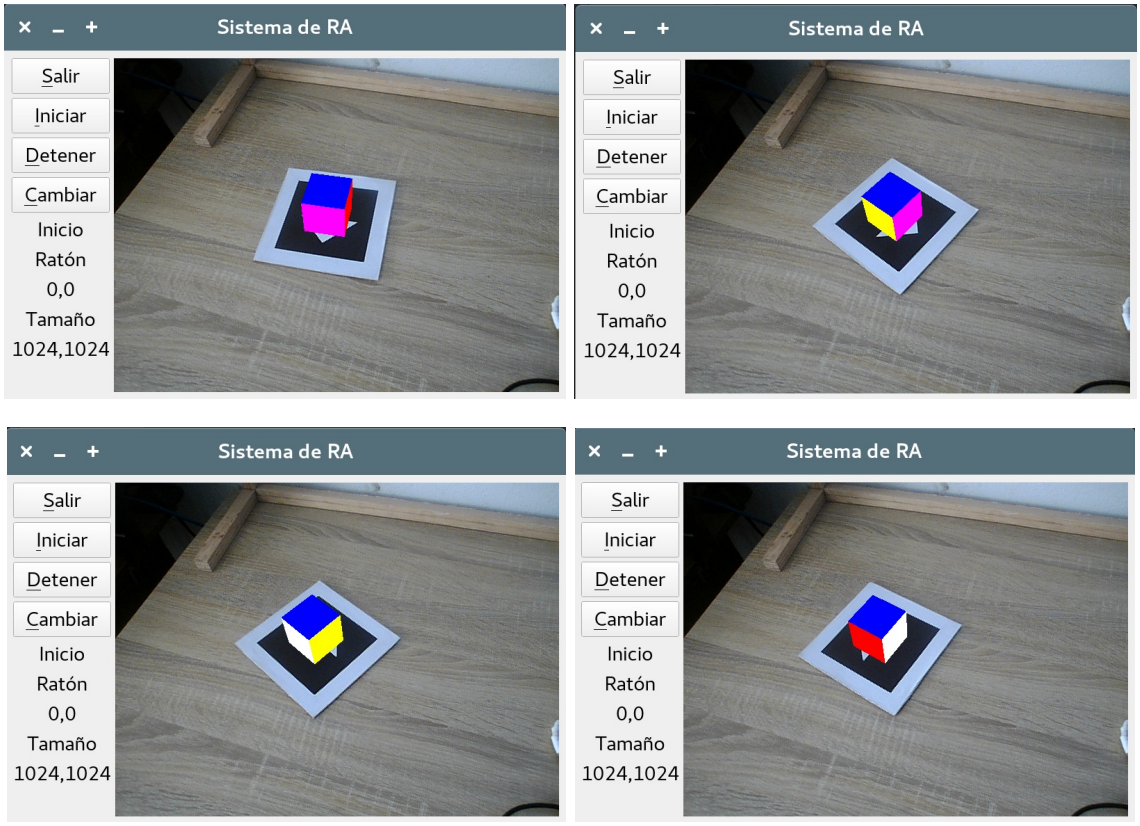


Figura 5.13: Visualización del cubo sobre el marcador. El cubo es dibujado en el sistema de coordenadas del marcador. Las cuatro imágenes ayudan a visualizar todos los colores de las caras, excepto el verde que está en su base.

Después, proyectamos las esferas que movemos con los datos de LM sobre el marcador. Para este es necesario aplicar la matriz de transformación de coordenadas del LM a coordenadas del marcador, de lo contrario las esferas no se verían sobre el marcador. Las esferas deben verse justo en la punta de los dedos índice y pulgar, pero esto no ocurría en nuestra primera prueba, como mostramos en la figura 5.14.

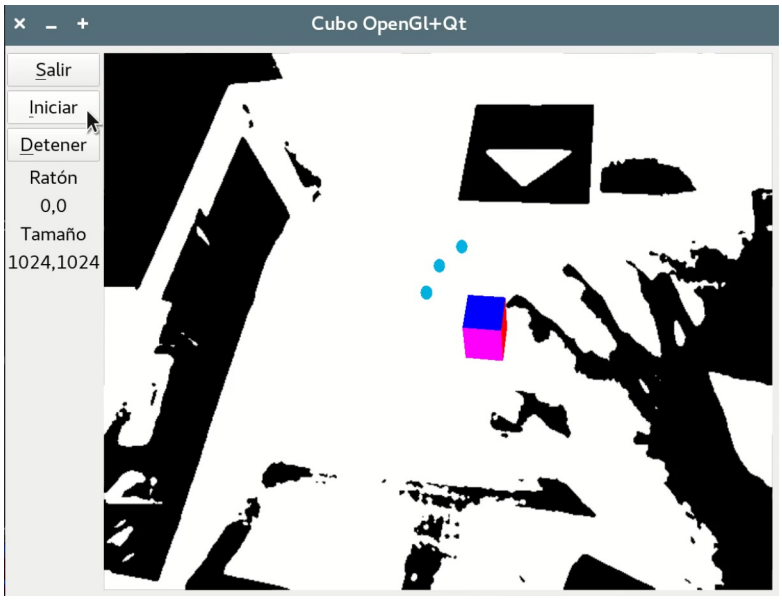


Figura 5.14: Problema en el que la posición de las esferas no coincidían con la posición de la punta de los dedos.

Este problema era debido a que el modelo del marcador con el que estábamos trabajando no estaba definido en milímetros y tampoco estaba definido el origen del sistema de coordenadas del marcador en el centro del mismo. Tomando en consideración estos dos puntos se planteó el modelo del marcador que ya definimos en la figura 2.5 en la pág. 22, con el que se corrigió este problema como se ve en la figura 5.15.

En este punto notamos que el dispositivo LM no proporcionaba los datos de movimiento tan rápido como para seguir el movimiento de las manos, por lo que se nota un ligero retardo del movimiento de las esferas con respecto al movimiento de la mano, como observamos en la figura 5.16.

Además del uso del marcador, se agregó un botón `cambiar` que sirve para cambiar entre la imagen a color y la imagen en blanco y negro de la cámara, ya que hasta el momento sólo mostrábamos la imagen blanco y negro que procesa la aplicación, como se muestra en la figura 5.17.

También, se agregó un comando para que los objetos de la escena volvieran a aparecer en su posición inicial. El comando se activa haciendo un desplazamiento horizontal hacia la izquierda con la mano, como se observa en la figura 5.18.

Por último, hay que mencionar que durante la implementación del sistema se probó la posibi-

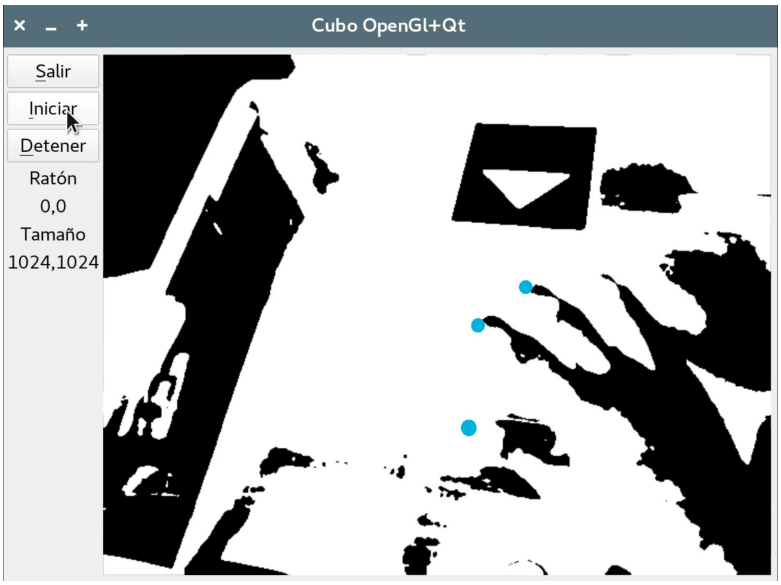


Figura 5.15: Posición de las esferas después de cambiar el modelo del marcador a mm y con el origen del sistema de coordenadas en el centro.

lidad de usar una lámina de acrílico como superficie en donde los objetos virtuales iban a caer y sobre la cual el usuario podría interactuar con ellos, colocando el dispositivo Leap Motion debajo del acrílico. Esta opción fue desechada ya que el acrílico comenzó a presentar rayaduras, y también al ensuciase con la grasa que tienen los dedos de la manos, lo que afectaba ostensiblemente la respuesta del Leap Motion. Debido a esto el LM nos proporcionaba información imprecisa sobre la pose de las manos del usuario. Por lo tanto, se decidió quitar la lámina de acrílico y se colocó el LM sobre una base de madera junto al marcador, como se muestra en la figura 5.19. Esto tiene la ventaja de poder mover la base de madera con los dos elementos sin tener afectación en la transformación de coordenadas de LM a coordenadas del marcador, además de que ya no hay elementos que puedan perjudicar el desempeño del LM.

5.5. Interacción con objetos deformables

Para la creación del objeto deformable hicimos uso de los cuerpos blandos de Bullet. Estos cuerpos nos permiten crear una esfera de triángulos que se deforma definiendo el número de triángulos que la componen y el radio de la esfera. Lo que no es sencillo es establecer los parámetros del objeto que definen su comportamiento. Nosotros definimos los parámetros principales para crear una esfera con un comportamiento similar al de una pelota, probando con distintos valores. La elección de los valores de cada parámetro es algo complicado, debido a que Bullet carece de una buena documentación sobre la definición de estos parámetros, así que nos limitamos a modificar los parámetros básicos de los cuerpos blandos que ya mencionamos en la sección 4.7 en la pág. 54. La elección de un mal valor en algún parámetro muchas veces hacía que el objeto deformable perdiera completamente su forma, como se muestra en la figura 5.20, por lo que probamos distintos valores hasta encontrar unos que nos entregaban un comportamiento lo

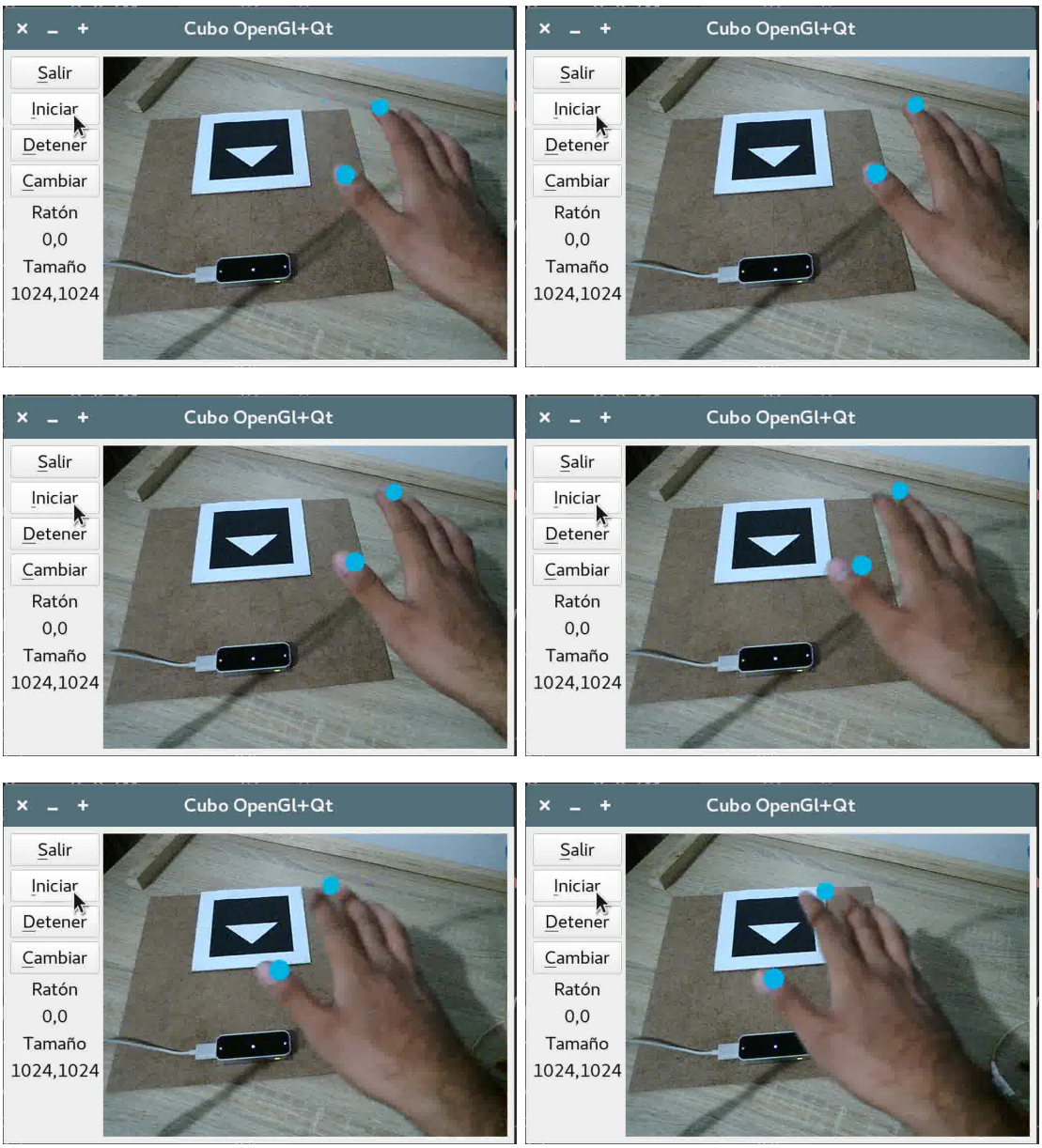


Figura 5.16: Retardo en el movimiento de las esferas con respecto al movimiento de la mano. Esto se aprecia en las imágenes en donde inicialmente las esferas coinciden con la posición de los dedos, al mover la mano las esferas siguen ese movimiento pero no se encuentran exactamente sobre las puntas de los dedos debido al retardo.

más cercano posible al de una pelota y lograban que el objeto mantuviera su forma durante la simulación.

Una vez seleccionados los parámetros del objeto deformable que nos daban el comportamiento más similar al de una pelota, se probó con dos esfera: una formada por 128 triángulos y otra formada por 512 triángulos. La interacción del usuario al tocar el objeto deformable es más evidente en la esfera formada por un mayor número de triángulos, ya que la colisión entre las

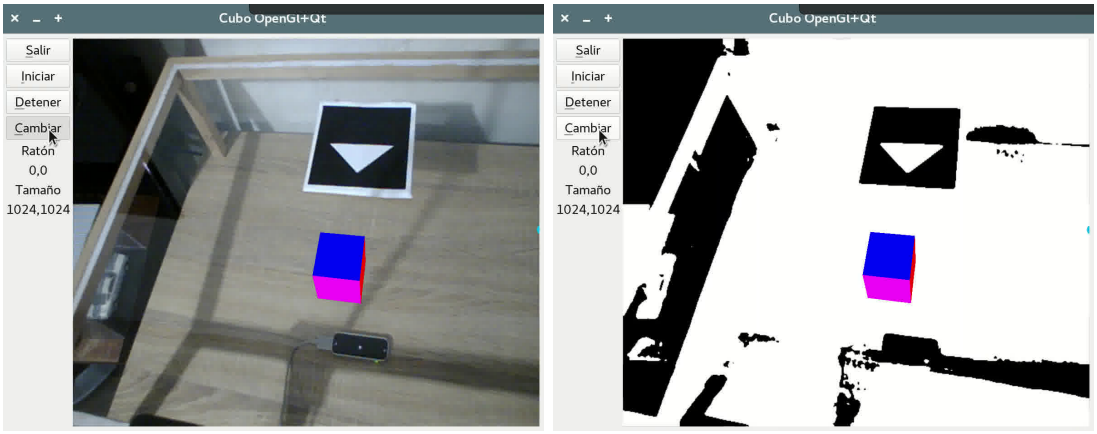


Figura 5.17: Botón que cambia entre la imagen en blanco y negro y la imagen a color de la cámara.

esferas de los dedos y el objeto deformable se da en los vértices. Un objeto deformable con mayor número de vértices tendrá una mejor respuesta al contacto con el usuario, ya que hay un mayor número de puntos en donde puede existir una colisión. En la figura 5.21 y 5.22 se observan las esferas de 128 triángulos y 512 triángulos, respectivamente.

El objeto deformable no puede ser sujetado por el usuario, como los objetos rígidos. Esto debido a que la colisión de una esfera de cualquiera de los dedos con el objeto deformable no genera ningún evento de colisión (esto solo ocurre en colisiones entre objetos rígidos). Esto hace que no se tenga una referencia del momento en que hay colisión y por ende no se puede crear la restricción con la que el gesto para sujetar los objetos fue implementado. Así que, los objetos deformables solo pueden ser empujados por el usuario dentro del sistema, a través del movimiento de las esferas cinemáticas en las puntas de los dedos de la mano.

5.6. Desventajas y limitaciones del sistema de RA

Ventajas:

- El dispositivo LM permite que el usuario pueda interactuar con los objetos virtuales de forma intuitiva, por lo que no requiere de mucho tiempo de entrenamiento para poder utilizar el sistema.
- El sistema es fácil de montar y puede ejecutarse en cualquier equipo de computo con Linux.
- Es sencillo agregar nuevos objetos al sistema, aunque este trabajo se limitó a probar con un objeto deformable.
- La base de madera donde se colocan el dispositivo LM y el marcador permite mover la escena sin tener afectación en la transformación de coordenadas del LM a coordenadas del

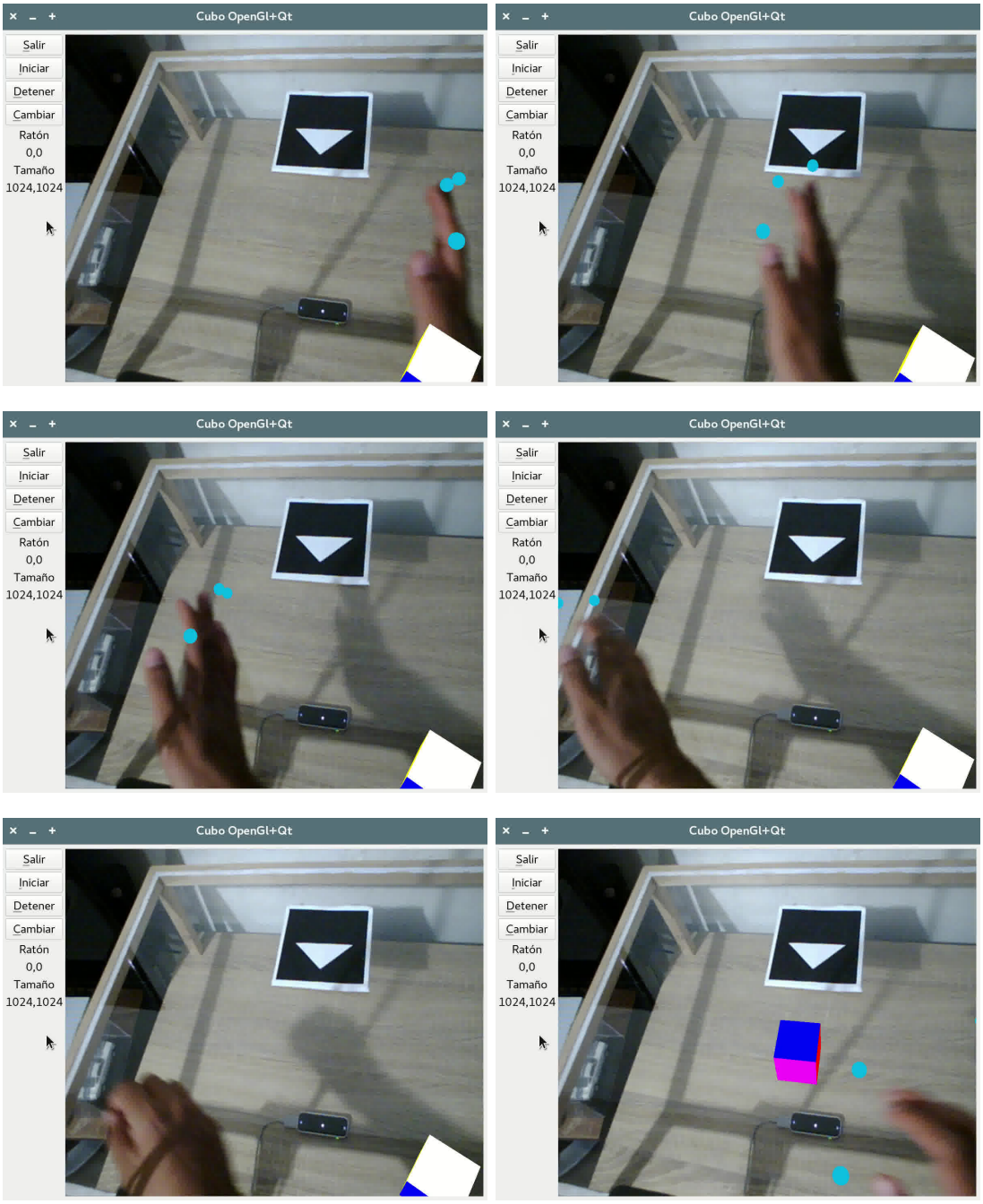
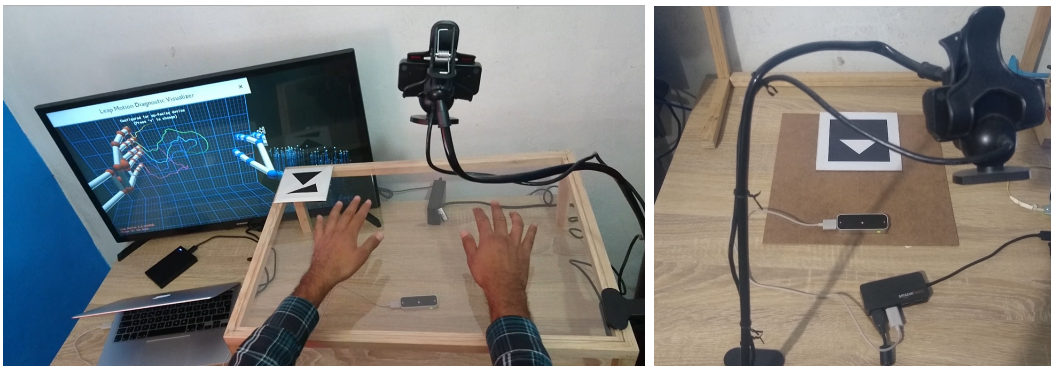


Figura 5.18: Comando para reiniciar la posición de los objetos virtuales del sistema a su posición inicial.

marcador. Por lo tanto, la escena puede moverse a la posición que el usuario quiera siempre y cuando el marcador sea visible para la cámara.

- El sistema utiliza dispositivos comerciales con un costo razonable.



(a) Configuración del sistema utilizando la superficie de acrílico

(b) Configuración del sistema colocando el marcador y el dispositivo LM sobre una base de madera

Figura 5.19: Configuración del sistema con acrílico empleada al inicio y la configuración del sistema por la que se sustituyo el acrílico.

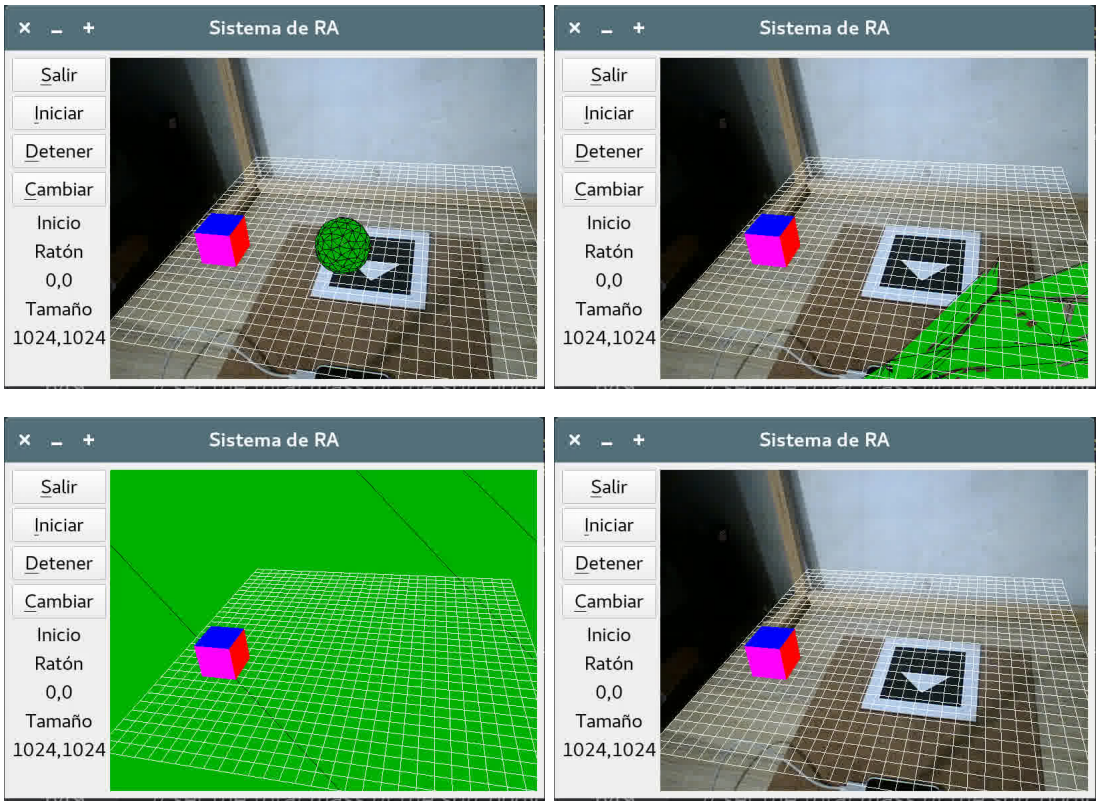


Figura 5.20: Objeto deformable que pierde su forma al caer a la superficie debido a un valor erróneo definido en el coeficiente de conservación de volumen.

Desventajas:

- La posición de la cámara da al usuario una perspectiva fija que limita la visualización de

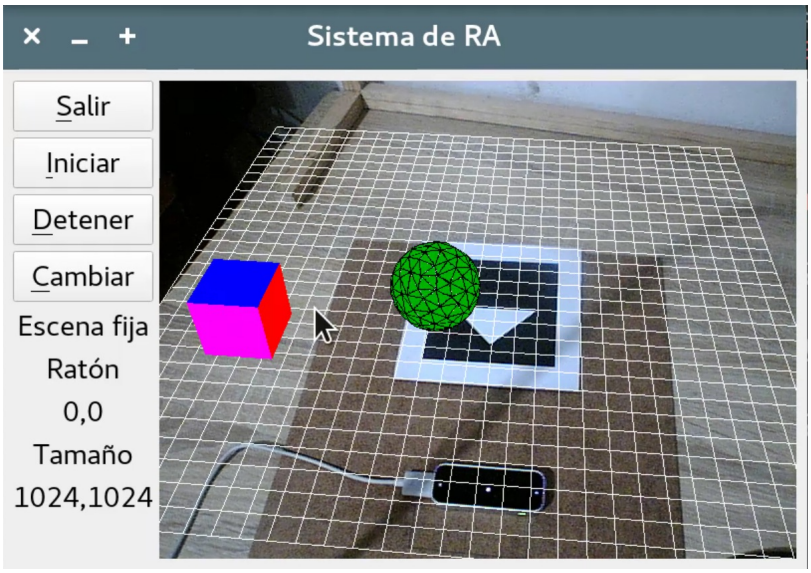


Figura 5.21: Esfera formada por 128 triángulos.

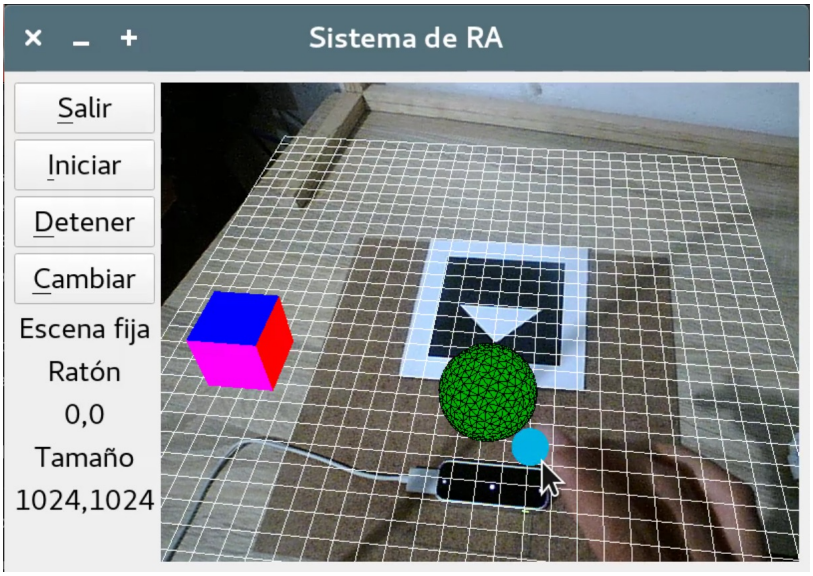


Figura 5.22: Esfera formada por 512 triángulos

la escena.

- El hecho de no contar con unas gafas de realidad virtual hace que el usuario tenga que estar viendo la pantalla del equipo de computo para visualizar los objetos virtuales mientras interactúa con ellos.

- El marcador y el LM deben mantener la misma distancia con la que fue calculada la matriz de transformación durante la ejecución del sistema, de lo contrario los datos del LM no coincidirán con la posición de las manos del usuario.

- La aplicación no puede utilizarse en MacOS debido a que no existen controladores para utilizar la cámara en dicho sistema.

Limitaciones:

- El tamaño de la escena del sistema está limitada por el rango de visión de Leap Motion y por la perspectiva de la cámara.
- A pesar de que el LM es capaz de detectar varias manos, el sistema está diseñado para que lo utilice solo una persona.
- El usuario puede levantar los objetos rígidos sujetándolos con ambos dedos (índice y pulgar) pero no los objetos deformables.
- El sistema sólo reconoce el uso de los dedos índice y pulgar de la mano derechoa, a pesar de que el LM nos proporciona la posición de un modelo completo de ambas manos.
- Al mover la mano, se aprecia que el movimiento de las esferas no es tan rápido como el de la mano, debido a que el LM no es tan rápido.

5.6.1. Resumen

En este capítulo se presentó la metodología de desarrollo y validación del sistema. Se presentaron las pruebas efectuadas en cada uno de los cuatro módulos del sistema, con lo que se muestra que todo el sistema funciona de manera adecuada.

Capítulo 6

Conclusiones

En esta tesis se desarrolló un sistema de RA que permite al usuario interactuar tanto con objetos rígidos como con objetos deformables en tiempo real. La interacción con los objetos ocurre de una forma intuitiva y natural, ya que el usuario no tiene que llevar puesto ningún dispositivo extra para poder realizar la interacción, gracias al uso del dispositivo Leap Motion. Este dispositivo nos proporciona la posición con una buena precisión logrando que estos datos sean aprovechados dentro del sistema para detectar cuando existe contacto entre las manos y los objetos virtuales. Representamos la punta de los dedos índice y pulgar de la mano derecha del usuario con esferas para que se pudiera detectar la interacción de éstas con los objetos. Encontramos que el LM no nos entregaba la posición de las manos lo suficientemente rápido como el movimiento de las mismas, por lo que se aprecia un pequeño retardo entre el movimiento de las esferas y el movimiento de la mano.

Uno de los problemas a resolver dentro del sistema era relacionar los datos de la posición de la mano que nos proporciona el dispositivo LM con la posición de los objetos virtuales en la escena. Resolviendo este problema podríamos conocer cuando la mano del usuario entra en contacto (colisiona) con los objetos virtuales. Para detectar la colisión entre la mano del usuario y los objetos tuvimos colocar las coordenadas del LM como del marcador en un mismo marco de referencia. Para lograr esto, se propuso colocar ambos sobre una base de madera en la cual definimos una distancia fija entre ambos dispositivos. Conocer esta distancia nos permitió calcular una matriz que transforme las coordenadas del LM a coordenadas del marcador. La base de madera permite al usuario desplazar ambos elementos dentro del sistema sin afectar la transformación de coordenadas.

La biblioteca Bullet Physics es una gran herramienta que facilitó la detección de colisiones entre los objetos y la definición del comportamiento basado en física de los mismos. La biblioteca nos permitió probar nuestro sistema tanto con objetos rígidos como con objetos deformables, logrando moverlos y levantarlos en el caso de los objetos rígidos y deformarlos en el caso del objeto deformable utilizando solamente nuestra mano tal como se haría con objetos reales. A pesar de la facilidad para crear los objetos, los parámetros que describen el comportamiento de

los objetos deformables eran complicados de definir, ya que en la documentación de la biblioteca no es muy clara tanto en definir el rango de valores de cada parámetro como el impacto que este provoca en el comportamiento. Por lo tanto, se probaron distintas combinaciones de parámetros y a prueba y error determinamos los parámetros que más se acercaban al comportamiento que deseamos.

Inicialmente la escena virtual vibraba bastante, por lo que se decidió agregar un filtro Kalman a cada coordenada de los vértices detectados. Esto mejoró la estabilidad del sistema, disminuyendo la vibración de los objetos que dibujamos en la escena. Además, para evitar el problema de oclusión del marcador con las manos del usuario lo cual impide su detección, se incluyo un gesto que permite fijar la escena virtual que se visualiza. De esta forma, el usuario tiene mayor de libertad de movimiento en la escena sin afectar la detección del marcador. También, se agregó un gesto para reiniciar el sistema que permite al usuario volver a iniciar la simulación de los objetos virtuales y reactivar la detección del marcador.

Una gran limitante del sistema es el hecho de que la cámara tenga una posición fija. Esto hace que el usuario no tenga una perspectiva completamente realista, ya que el usuario puede moverse y tener una perspectiva distinta a la qué ofrece la cámara. El sistema podría ser mejorado incluyendo unas gafas de realidad virtual al sistema, con lo que el usuario tendría una mejor perspectiva de la escena. Además, otra limitante importante es que el usuario ve los objetos virtuales a través de una aplicación en el monitor del equipo de computo, lo que provoca que el usuario tenga que dirigir la mirada hacia la aplicación en lugar de ver hacia la escena real lo cual sería más natural.

6.1. Trabajo Futuro

A continuación se enlistan las mejoras que se pueden hacer sobre el sistema de RA propuesto:

- Definir un modelo completo de ambas manos en el sistema, ya que sólo se puede usar la punta de los dedos índice y pulgar de la mano derecha. De esta forma, el usuario podrá usar cualquiera de sus manos y esto permitiría la definición de más gestos para mejorar la interacción con los objetos virtuales.

- Definición de un gesto para levantar los objetos deformables dado que nuestro sistema sólo permite deformarlo empujándolo con la punta de los dedos.

- Probar con más objetos virtuales y determinar cuántos objetos puede manejar el sistema simultáneamente.

- Incorporar unas gafas de realidad virtual en el sistema para que el usuario tenga una experiencia más realista, ya que el sistema tiene una perspectiva fija por la posición de la cámara.

- Tomar la lectura de los datos del LM a una mayor frecuencia, ya que ahora el sistema lo hace a 60 Hz, dado que el manual indica que el LM puede trabajar hasta una frecuencia de 200 Hz. Con esto se espera que las esferas colocadas en la punta de los dedos se muevan con mayor velocidad y se reduzca el retardo que existe al seguir el movimiento de las manos.

Apéndices

Apéndice A

Cálculo de una homografía

El siguiente script permite calcular la homografía de dos conjuntos de puntos en dos dimensiones.

```
# encoding: utf8
import math
import numpy as np
import numpy.linalg as la

def mediaVarianza( M ) :

  rows = len( M )
  cols = len( M[0] )

  media = [0.0, 0.0]
  varianza = [0.0, 0.0]
  i = 0
  while i < cols :
    # Calcumos medias y sigmas de los puntos p1
    med = 0.0
    sig = 0.0
    j = 0
    while j<rows :
      if  cols == 1 :
        v = M[j]
      else :
        v = M[j][i]

      med += v
      sig += v*v
      j += 1

    media[i] = med/rows
    varianza[i] = sig/rows - media[i]*media[i]
    i += 1

  return ( media, varianza )

```

```
35 # Calcula la homografía de dos conjuntos de puntos en 2D
36 #
37 def homografia( p1, p2 ) :
38   # Calcumos medias y sigmas de los puntos p1 y p2
39   med1, sig1 = mediaVarianza( p1 )
40   med2, sig2 = mediaVarianza( p2 )
41
42   sig1[0] = math.sqrt(sig1[0])
43   sig1[1] = math.sqrt(sig1[1])
44   sig2[0] = math.sqrt(sig2[0])
45   sig2[1] = math.sqrt(sig2[1])
46
47   n = len( p1 )
48   # print n
49   A = np.zeros( (2*n, 9) )
50   i = 0
51   while i < n :
52     u1 = ( p1[i][0] - med1[0] ) / sig1[0]
53     v1 = ( p1[i][1] - med1[1] ) / sig1[1]
54
55     u2 = ( p2[i][0] - med2[0] ) / sig2[0]
56     v2 = ( p2[i][1] - med2[1] ) / sig2[1]
57
58     k = 2*i
59     A[ k ][ 0 ] = u1
60     A[ k ][ 1 ] = v1
61     A[ k ][ 2 ] = 1.0
62     A[ k ][ 3 ] = 0.0
63     A[ k ][ 4 ] = 0.0
64     A[ k ][ 5 ] = 0.0
65     A[ k ][ 6 ] = - u1 * u2
66     A[ k ][ 7 ] = - v1 * u2
67     A[ k ][ 8 ] = u2
68
69     k = 2*i + 1
70     A[ k ][ 0 ] = 0.0
71     A[ k ][ 1 ] = 0.0
72     A[ k ][ 2 ] = 0.0
73     A[ k ][ 3 ] = u1
74     A[ k ][ 4 ] = v1
75     A[ k ][ 5 ] = 1.0
76     A[ k ][ 6 ] = - u1 * v2
77     A[ k ][ 7 ] = - v1 * v2
78     A[ k ][ 8 ] = v2
79
80     i += 1
81
82   # Ax = b,
83   # QR x = b,
84   # (QR)^{-1} QR x = (QR)^{-1} b
85   # R^{-1} Q^T Q R x = R^{-1} Q^T b,
86   # R x = Q^T b.
87   # Se realiza la descomposición QR de la matriz aumentada
88   # [A, b] = [QR, b] = Q[ R, Q^T b ].
89   R = la.qr( A, 'r' )
90   # print R
```

```
91
92      # Retrosustitución
93    vh = np.zeros( 8 )
94
95    vh[7] = R[7][8]/R[7][7]
96
97    i = 6
98    while i>=0 :
99      suma = 0.0;
100     j = i + 1
101     while j<8 :
102       suma += R[i][j] * vh[j]
103       j += 1
104
105     vh[i] = ( R[i][8] - suma )/R[i][i]
106     i -= 1
107
108   Hp = [ [ vh[0], vh[1], vh[2] ],
109          [ vh[3], vh[4], vh[5] ],
110          [ vh[6], vh[7],  1.0 ] ]
111
112   T1 = [ [ 1.0/sig1[0],      0.0     , -med1[0]/sig1[0] ],
113          [     0.0     , 1.0/sig1[1], -med1[1]/sig1[1] ],
114          [     0.0     ,      0.0     ,         1.0       ] ]
115
116   T2i = [ [ sig2[0],      0     ,  med2[0]    ],
117           [    0    ,   sig2[1],  med2[1]    ],
118           [    0    ,      0    ,     1       ] ]
119
120   # l p2 = H p1
121   # l T2 p2 = H T1 p1
122
123   H = np.dot( np.dot( T2i, Hp ), T1 )
124
125   return H
126
127 def lee_archivo_puntos( nombre, M ) :
128   try :
129     arch = open( nombre, "r" )
130   except:
131     print "No se puede abrir el archivo", nombre
132     exit(2)
133
134   i = 0
135   while 1 :
136     linea = arch.readline( )
137     if linea == "" :
138       break
139
140     if linea[0] == "#" :
141       continue
142
143     if i == 4 :
144       break
145
146     d = linea.split( )
```

```
147     n = len( d )
148     if n == 2 :
149       M[i][0] = float( d[0] )
150       M[i][1] = float( d[1] )
151 
152     i += 1
```

Apéndice B

Proyección del marcador utilizando la matriz de calibración

El siguiente script hace la proyección de puntos según la matriz de calibración y posición de la cámara definidos dentro del mismo.

```
# encoding: utf8
# Se proyecta el marcador.
# se rota en z con el ángulo especificado y
# se translada según dx y dy
#
# La cámara está fija en el punto
# c = np.array( [  [50], [-200], [200] ] )
#
# 19.06.2019
# Fraga
import numpy as np
import numpy.linalg as la
import math
import sys

n = len( sys.argv )
if n != 5 :
  print "Args: theta2 theta1 dx dy"
  sys.exit(1)

theta2 = float( sys.argv[1] )
theta1 = float( sys.argv[2] )
dx = float( sys.argv[3] )
dy = float( sys.argv[4] )

K = [ [ -548.22,    0.0, 333.26], #  640
      [    0.0, -548.22, 257.91], #  480
      [    0.0,    0.0,   1.0] ]

P = [ [-55.0, -55.0,  0.0, 1.0 ],
      [ 55.0, -55.0,  0.0, 1.0 ],
```

```
      [  0.0, -37.5,  0.0, 1.0 ],
      [ 37.5,   0.0,  0.0, 1.0 ],
      [ 55.0,  55.0,  0.0, 1.0 ],
      [-37.5,   0.0,  0.0, 1.0 ],
      [-55.0,  55.0, 0.0, 1.0  ] ]

# Lo rotamos alrededor de z
ang = math.radians( theta1 )
R1 = np.eye( 3 )
R1[0,0] =  math.cos(ang)
R1[0,1] = -math.sin(ang)
R1[1,0] =  math.sin(ang)
R1[1,1] =  math.cos(ang)

# Lo rotamos alrededor de y
ang = math.radians( theta2 )
R2 = np.eye( 3 )
R2[0,0] =  math.cos(ang)
R2[0,2] =  math.sin(ang)
R2[2,0] = -math.sin(ang)
R2[2,2] =  math.cos(ang)

R = np.dot( R2, R1 )

# Rotamos el cuerpo
p1 = np.array( [ [0.0], [0.0], [0.0] ] )
n = len( P )
i = 0
while i < n :
  p1[0][0] = P[i][0]
  p1[1][0] = P[i][1]
  p1[2][0] = P[i][2]
  p1 = np.dot( R, p1 )
  P[i][0] = p1[0][0]
  P[i][1] = p1[1][0]
  P[i][2] = p1[2][0]
  i += 1

# Transladamos el cuerpo
i = 0
while i < n :
  P[i][0] += dx
  P[i][1] += dy
  i += 1

# Modelo de la cámara obscura
# lp = K [ R|t ] P, o
# lp = K R [ I| -c ] P
# viendo al punto O y el vector arriba es a = [0 0 1]'
a = np.array( [ [0], [0], [1] ] )

c = np.array( [  [50], [-200], [200] ] )
print "# c^T :", np.transpose( c )

z = c/la.norm(c)

```

```
88 # x = a x z
89 xp = np.cross( a, z, axis=0 )
90 x = xp/la.norm(xp)
91
92 y = np.cross( z, x, axis=0 );
93
94 R = []
95 R = x.transpose()
96 R = np.append( R, y.transpose(), axis=0 )
97 R = np.append( R, z.transpose(), axis=0 )
98
99 I = np.eye(3,4)
100 I[:,3] = -c[:,0]
101 # M = [ I, -c]
102 # M = K * R * [ I, -c];
103 M = np.dot( R, I )
104 M = np.dot( K, M )
105
106 t = np.dot( R, -c )
107 print "# t^T :", np.transpose( t )
108
109 i = 0
110 while i < 7 :
111   p = P[i]
112   p1 = np.dot( M, np.transpose(p) )
113   u = p1[0]/p1[2]
114   v = p1[1]/p1[2]
115   print( "{0:3.6f} {1:3.6f}".format(u, v) )
116
117   i += 1
```

Apéndice C

Diseño de un cubo virtual

Los arreglos del código mostrado en C.1 definen los vértices, los colores y los índices de cada cara del cubo y la figura C.1 muestra cómo debe verse el cubo.

```
// vértices del cubo
 static float Mvertices[8][3] = {
                {  55.0,  55.0,   0.0},
                {  55.0, -55.0,   0.0},
                { -55.0,  55.0,   0.0},
                { -55.0, -55.0,   0.0},
                {  55.0,  55.0, 110.0},
                {  55.0, -55.0, 110.0},
                { -55.0,  55.0, 110.0},
                { -55.0, -55.0, 110.0} };

// colores de cada cara
static float Mcolores[6][3] = {
                {1.0,0.0,0.0},
                {0.0,0.0,1.0},
                {0.0,1.0,0.0},
                {1.0,0.0,1.0},
                {1.0,1.0,0.0},
                {1.0,1.0,1.0} };

// índices de cada cara del cubo
static int Mcaras[6][4] = {
                { 0, 1, 2, 3},
                { 4, 5, 6, 7},
                { 0, 1, 4, 5},
                { 1, 5, 3, 7},
                { 3, 7, 2, 6},
                { 2, 6, 0, 4} };
```

Listing C.1: Diseño de un cubo virtual

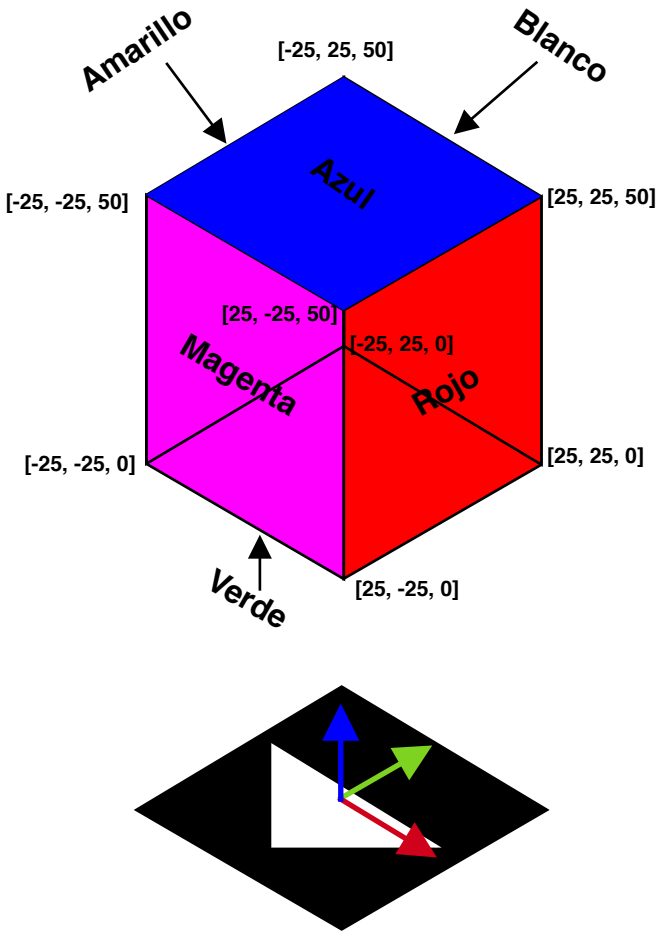


Figura C.1: Diseño del cubo.

Apéndice D

Archivo de parámetros de la cámara

El archivo D.1 muestra el archivo que contiene los parámetros de la cámara obtenidos del proceso de calibración de la cámara.

```
<?xml version="1.0"?>
<opencv_storage>
<calibrationDate>"jue 20 dic 2018 22:24:18 CST"</calibrationDate>
<framesCount>25</framesCount>
<cameraResolution>640 480</cameraResolution>
<cameraMatrix type_id="opencv-matrix">
  <rows>3</rows>
  <cols>3</cols>
  <dt>d</dt>
  <data>5.4821753862873572e+02 0. 3.3326289320495243e+02 0.
    5.4821753862873572e+02 2.5791273657626022e+02 0. 0. 1.</data></cameraMatrix>
<cameraMatrix_std_dev type_id="opencv-matrix">
  <rows>4</rows>
  <cols>1</cols>
  <dt>d</dt>
  <data>0. 1.8562883166269470e+00 6.6223700510272365e-01
    8.0389625317748836e-01</data></cameraMatrix_std_dev>
<dist_coeffs type_id="opencv-matrix">
  <rows>1</rows>
  <cols>5</cols>
  <dt>d</dt>
  <data>-1.2471214068773409e-01 2.4422634818101269e-01 0. 0.
    -6.8198894180386202e-02</data></dist_coeffs>
<dist_coeffs_std_dev type_id="opencv-matrix">
  <rows>5</rows>
  <cols>1</cols>
  <dt>d</dt>
  <data>1.0135055976960525e-02 6.1851084146791337e-02 0. 0.
    1.1584063413202061e-01</data></dist_coeffs_std_dev>
<avg_reprojection_error>3.1216986659320090e-01</avg_reprojection_error>
</opencv_storage>
```

Listing D.1: Parámetros de la cámara

Bibliografía

[1] *API Overwiew*, https://developer-archive.leapmotion.com/documentation/cpp/devguide/Leap_Overview.html?proglang=cpp, 2019.

[2] 2019, *Camera calibration With OpenCV*, https://docs.opencv.org/2.4/doc/tutorials/calib3d/camera_calibration/camera_calibration.html, 2019.

[3] 2019, *How Does the Leap Motion Controller Work?*, http://blog.leapmotion.com/hardware-to-software-how-does-the-leap-motion-controller-work/, 2019.

[4] 2019, *Interactive camera calibration application*, https://docs.opencv.org/master/d7/d21/tutorial_interactive_calibration.html, 2019.

[5] 2019, *Playstation Eye*, https://www.playstation.com/es-es/content/dam/support/manuals/scee/web-manuals/peripherals/ps3/ps3-eye/SCEH-00448_PS%20Eye%20Web_ES.pdf/, 2019.

[6] Sergio Herman Peralta Benhumea, *Interfaz de lenguaje natural usando Kinect*, Tesis de maestría, Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional, Departamento de Computación, 2012.

[7] Volkert Buchmann, Stephen Violich, Mark Billinghurst, and Andy Cockburn, *Fingartips - gesture based direct manipulation in augmented reality*, Association for Computing Machinery (ACM) (2004), 212–221.

[8] Adrian Clark and Thammathip Piumsomboon, *A realistic augmented reality racing game using a depth-sensing camera*, Proceedings of VRCAI 2011: ACM SIGGRAPH Conference on Virtual-Reality Continuum and its Applications to Industry (2011).

[9] Sam Corbett-Davies, Andreas Dünser, Richard Green, and Adrian Clark, *An advanced interaction framework for augmented reality based exposure treatment*, (2013), 19–22.

[10] Erwin Coumans, https://github.com/bulletphysics/bullet3, 2019.

[11] Erwin Coumans, *Bullet 2.80 physics sdk manual*, 2012.

[12] Heriberto Cruz-Hernández and Luis Gerardo de la Fraga, *A fiducial tag invariant to rotation, translation, and perspective transformations*, Pattern Recognition **81** (2018), 213 – 223.

[13] Heriberto Cruz-Hernández and Luis Gerardo de la Fraga, *Order type dataset analysis for fiducial markers*, Data in Brief **20** (2018), 1068 – 1072.

[14] Chris Dickinson, *Learning game physics with bullet physics and opengl*, Packt Publishing Ltd., Livery Place 35, Livery Street, Birmingham B3 2PB, UK., 2013.

[15] David Forsyth and Jean Ponce, *Computer vision: The modern approach*, Pearson, 2011.

[16] E. G. Gilbert, D. W. Johnson, and S. S. Keerthi, *A fast procedure for computing the distance between complex objects in three-dimensional space*, IEEE Journal on Robotics and Automation **4** (1988), no. 2, 193–203.

[17] Richard Hartley and Andrew Zisserman, *Multiple view geometry in computer vision*, Cambridge University Press, New York, 2004.

[18] Reiji Katahira and Masato Soga, *Development and evaluation of a system for ar enabling realistic display of gripping motions using leap motion controller*, 19th International Conference on Knowledge Based and Intelligent Information and Engineering Systems (2015).

[19] Micky Kelager, Sarah Niebe, and Kenny Erleben, *A triangle bending constraint model for position-based dynamics.*, VRIPHYS **10** (2010), 31–37.

[20] Leap Motion Controller, `https://www.leapmotion.com/`, 2019.

[21] Microsoft Kinect, `https://developer.microsoft.com/en-us/windows/kinect`, 2019.

[22] Paul Milgram and Fumio Kishino, *A taxonomy of mixed reality visual displays*, IEICE Transactions on Information Systems (1994).

[23] Eray Molla and Vincent Lepetit, *Augmented reality for board games*, IEEE International Symposium on Mixed and Augmented Reality 2010 (2010).

[24] Matthias Müller, Bruno Heidelberger, Marcus Hennix, and John Ratcliff, *Position based dynamics*, Journal of Visual Communication and Image Representation **18** (2007), no. 2, 109 – 118.

[25] Matthias Müller-fischer, *Hierarchical position-based dynamics*, Virtual Reality Interaction and Physical Simulation VRIPHYS (2008), no. 8996337.

[26] Daybelis Jaramillo Olivares, *Prototipo de un sistema de ojo de halcón*, Tesis de maestría,

Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional, Departamento de Computación, 2018.

[27] Axel Salazar Ordoñez, *Análisis de los dispositivos space navigator, myo y leap motion y su evaluación en aplicaciones de software*, Tesis de maestría, Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional, Departamento de Computación, 2016.

[28] Julio Guadalupe Moctezuma Ramírez, *Manipulación Tridimensional de Objetos Deformables Virtuales*, Tesis de maestría, Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional, Departamento de Computación, 2006.

[29] Phattanapon Rhienmora, Kugamoorthy Gajananan, Peter Haddawy, Matthew N. Dailey, and Siriwan Suebnukarn, *Augmented reality haptics system for dental surgical skills training*, the 17th ACM Symposium on Virtual Reality Software and Technology (2010).

[30] Claudia Magdalena Ramírez Trejo, *Animación de Modelos Deformables*, Tesis de maestría, Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional, Departamento de Computación, 2005.

[31] Ryosuke Umeda, Mohamed Atef Seif, Hiroki Higa, and Yukio Kuniyoshi, *A medical training system using augmented reality*, International Conference on Intelligent Informatics and BioMedical Sciences (ICIIBMS) (2017).

[32] Paolo Valentini, *Natural interface for interactive virtual assembly in augmented reality using leap motion controller*, International Journal on Interactive Design and Manufacturing (IJIDeM) (2018).