E.T.S. de Ingeniería Industrial, Informática
y de Telecomunicación

Aplicación VR de Diseño gráfico en 3D

Máster Universitario
en Ingeniería Informática

Trabajo Fin de Máster

Alumno: Rubén Baztán Larrea

Director: Oscar Ardaiz Villanueva

Pamplona, 25 de Septiembre de 2020

upna
Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

Agradecimientos

No puedo empezar una memoria de un trabajo de fin de estudios sin antes dar las gracias, las gracias a todas las personas sin las cuales todo este proceso no habría sido posible.

En primer lugar, agradezco la oportunidad que brinda la Universidad Pública de Navarra respecto a la posibilidad de adquirir formación en posgrado, de manera asequible para la mayoría de las personas. Agradezco también la labor de los profesores, implicados en su trabajo. En especial la de Oscar Ardaiz, sin el cual no habría sido posible la elaboración de este proyecto.

También quiero agradecer el hecho de haber tenido los compañeros que me han tocado. Siempre hemos tenido un buen clima, favorecedor del trabajo, pero también del disfrute y apoyándonos unos a otros cuando lo hemos necesitado.

Por último, pero no menos importante, quiero agradecer a mi familia y amigos el apoyo que me han dado durante el transcurso de estos estudios, especialmente en los momentos en los que he estado algo atascado.

Índice

1. Introducción 3

2. Objetivos y motivación del proyecto 3

3. Herramientas utilizadas 4

4. Funcionalidad de la aplicación 5

5. Desarrollo 11

6. Resultados 34

7. Conclusiones 37

8. Bibliografía y referencias 38

1. Introducción

En esta memoria se expone el trabajo realizado como proyecto final del Máster de Ingeniería Informática en la UPNA. El proyecto se ha llevado a cabo en el departamento de Estadística, Informática y Matemáticas. Las personas implicadas en el mismo han sido el profesor Oscar Ardaiz Villanueva y el alumno de máster Rubén Baztán Larrea. En la memoria se explica la finalidad del proyecto, ¿de dónde ha surgido esta idea? Pondremos sobre la mesa las herramientas con las que hemos contado para llevarlo a cabo. Posteriormente, ahondaremos en la funcionalidad de la aplicación, así como en el desarrollo de la misma, en cómo se ha programado y en aquellas ideas que no se han podido materializar debido a las circunstancias especiales que estamos viviendo.

2. Objetivos y motivación del proyecto

Este proyecto consiste, en la elaboración de una aplicación de realidad virtual. La funcionalidad de la misma es la de dibujar y pintar sobre diferentes superficies virtuales. La idea surgió a partir de otras aplicaciones de realidad virtual, hemos probado aplicaciones como Gravity Sketch u Oculus Medium. Nosotros queríamos darle una vuelta a lo que ofrecen esas aplicaciones, con el material del que disponíamos.

En primer lugar, puede utilizarse como aplicación de diseño, ya sea de interiores, de vehículos, de prendas de ropa… Puede utilizarse tanto para diseños artísticos como para diseños más técnicos, aunque para estos últimos la aplicación no es tan completa y precisa como otras aplicaciones que ya existen. La cuestión es que el usuario pueda colocar en la escena un modelo 3D en blanco y comenzar a pintarlo a su antojo. De esta manera se podría realizar el diseño en 3D de forma relativamente rápida, antes de invertir en los materiales necesarios para hacerlo real.

También puede tener una utilidad educativa. Para desarrollar la creatividad de los alumnos, así como generar retos que les ayuden a desarrollar su inteligencia, creatividad y habilidades motrices. El hecho de que la aplicación sea de realidad virtual podría resultar más atractivo y motivante para los niños y niñas. Por ejemplo, que tengan que repasar una superficie virtual, con la máxima precisión posible o que tengan que dibujar algo en concreto sobre una superficie curva, en lugar de plana. No obstante, esta utilidad no ha podido ser comprobada del todo de momento, debido a que el coronavirus no nos ha permitido realizar las pruebas necesarias.

La aplicación está pensada para utilizarse en un espacio relativamente amplio, lo suficiente como para realizar ciertos movimientos libremente. También se recomienda disponer de una silla con ruedas que pueda girar para garantizar una experiencia óptima.

3. Herramientas utilizadas

Para efectuar este proyecto se ha precisado de diferentes herramientas de hardware y software.

En el apartado de hardware, hemos contado con:

- **Unas gafas de VR Oculus Rift S**, herramienta de visualización del entorno virtual. Además, cuenta con dos mandos por control remoto que nos han permitido darle una mayor riqueza de interacción a la aplicación. Para el funcionamiento de esta herramienta, ha sido necesario que el equipo utilizado precisara de una tarjeta gráfica lo suficientemente potente, concretamente una NVIDIA GeForce GTX 1060.

- **Una tableta gráfica Wacom Intuos S**. Junto a las Oculus, es el principal elemento de interacción con la aplicación, pues lo que dibujemos sobre ella se trasladará al mundo virtual. Esto se explicará detalladamente en un apartado posterior. Cuenta con un puntero táctil especial y puede detectar tanto la posición de contacto como la presión que se ejerce con el mismo.

En cuanto a las herramientas de software:

- **Unity** ha sido la principal. Se trata de un motor de videojuegos que permite desarrollar aplicaciones en realidad virtual. Su instalación incluye el editor Visual Estudio.

4. Funcionalidad de la aplicación

Para sumergirse en la aplicación, el usuario debe ponerse las Oculus en un espacio lo suficientemente amplio como para poder desplazarse por el entorno virtual. Una vez que se sumerja en la aplicación, el usuario podrá interactuar con dicho entorno.

4.1 Elementos del entorno

Para poder interactuar, antes debemos conocer los elementos principales que encontramos en el entorno virtual.

1) **Cámara principal:** Se trata de un elemento especial, realmente no es una cámara como tal, sino un objeto del tipo *OVRCameraRig*, que contiene varias cámaras y otros elementos que permiten la realidad virtual. Básicamente es el elemento que representa al jugador y que le permite ver e interactuar con el entorno virtual.

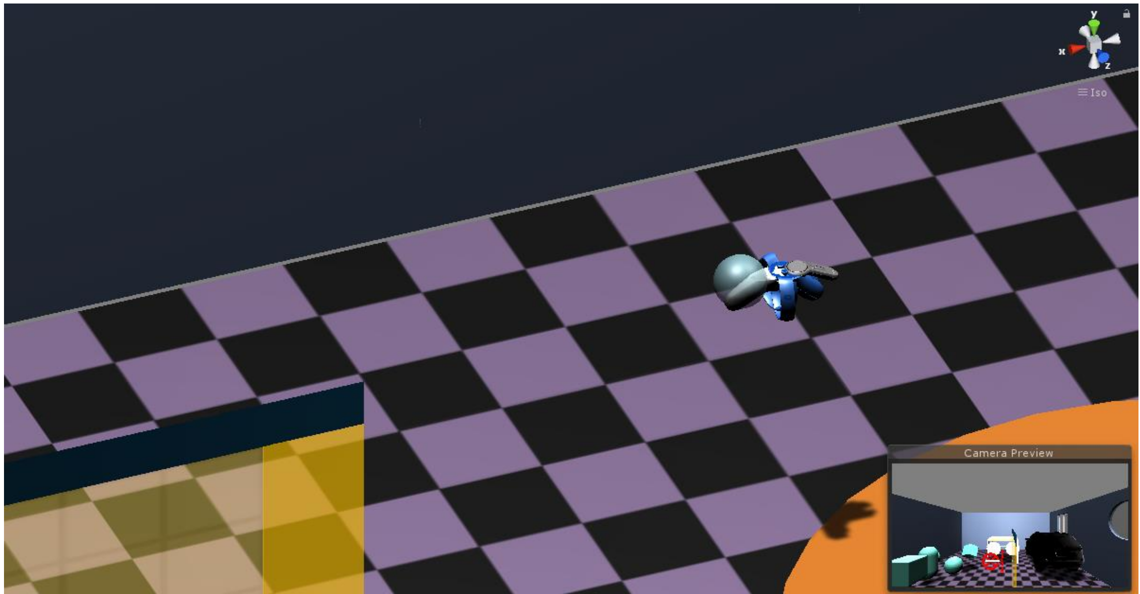

Figura 1. Imagen del entorno virtual. En la esquina inferior derecha vemos lo que estaría siendo captado por la cámara que representa al jugador.

2) **Lienzo virtual:** Este lienzo es la representación virtual de la tableta sobre la que dibujamos en la realidad (si es que utilizamos la tableta). En el apartado 4.2 explicaremos su funcionamiento.

Figura 2. Imagen del lienzo virtual, con el puntero/lápiz virtual sobre él.

3) **Modelos:** Se trata de los elementos, predefinidos por el usuario, sobre los cuales este quiere realizar algún tipo de diseño. Puede tratarse de una casa, un maniquí vestido, un coche… Modelos en 3D prefabricados previamente.

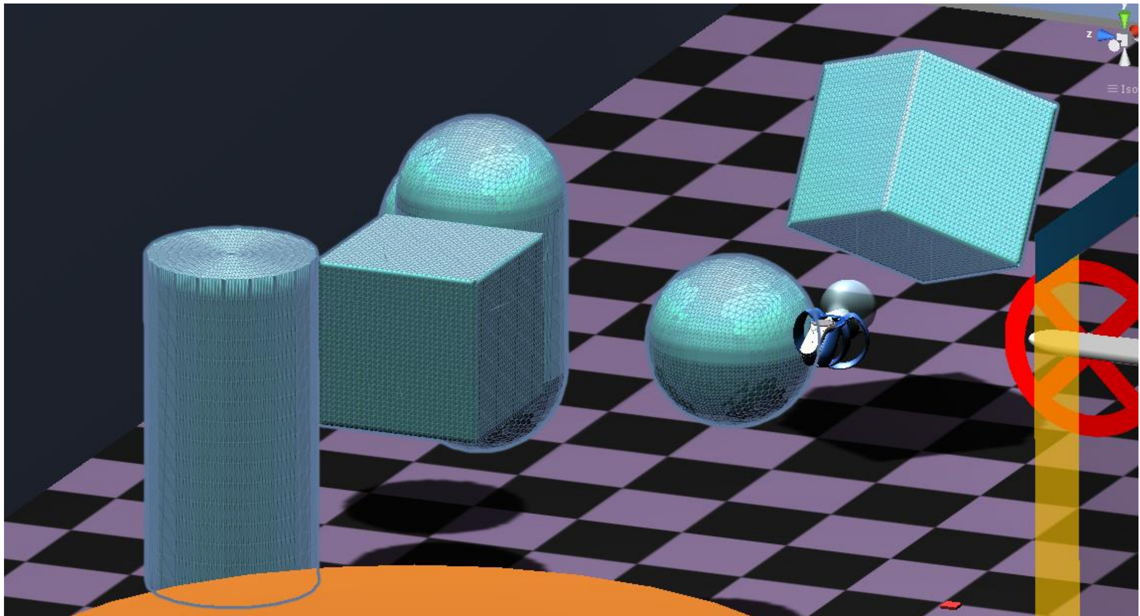

Figura 3. Imagen de varios modelos dentro del entorno virtual.

4) **Habitación:** Es el elemento que permite limitar el espacio y que el usuario se desplace por el entorno virtual, sin necesidad de hacerlo en el entorno real.

4.2 Formas básicas de interacción con el entorno

Habiendo identificado los elementos principales, el usuario puede interactuar con ellos de diferentes formas:

4.2.1 Mecanismos de pintura

La pintura es la forma de interacción principal. La aplicación cuenta con tres mecanismos principales para pintar de forma diferente. Cuando iniciamos la aplicación elegimos el mecanismo que queremos utilizar. Los describo aquí.

1) **Pintura con tableta:** En este modo, se emplea la tableta Wacom como herramienta para dibujar. En el entorno encontraremos un "lienzo virtual" (antes mencionado). Este lienzo es la representación virtual de la tableta sobre la que dibujamos en la realidad. Al apoyar el puntero táctil sobre un punto determinado de la tableta real y pulsar el botón del puntero, la pintura aparecerá en el punto mapeado correspondiente del lienzo virtual (también aparece un lápiz virtual, allí donde tocamos). Después, si desplazamos el lienzo, la pintura se mantendrá donde la hemos colocado. También tendremos el apoyo de uno de los mandos de Oculus para realizar otras acciones (mando de apoyo).

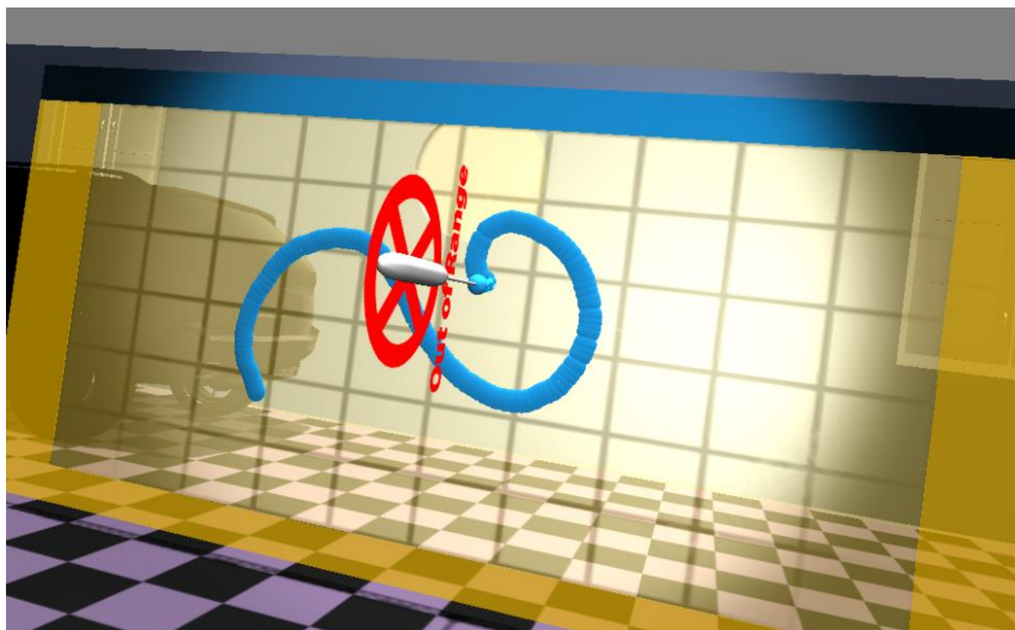

Figura 4. Mecanismo de pintura utilizando la tableta. Vemos el lienzo virtual, que es la representación real de la tableta, junto al lápiz.

2) **Pintura con mandos y rayo:** En este modo, no emplearemos la tableta Wacom. Simplemente utilizaremos los mandos de Oculus. Así como en el mecanismo anterior utilizábamos solo uno, como apoyo, en este utilizaremos los dos, uno para dibujar y otro como apoyo. Desde el mando de dibujo parte un rayo; al mantener pulsado uno de los botones, se formará un trazo sobre aquella superficie con la que intersecte el rayo.

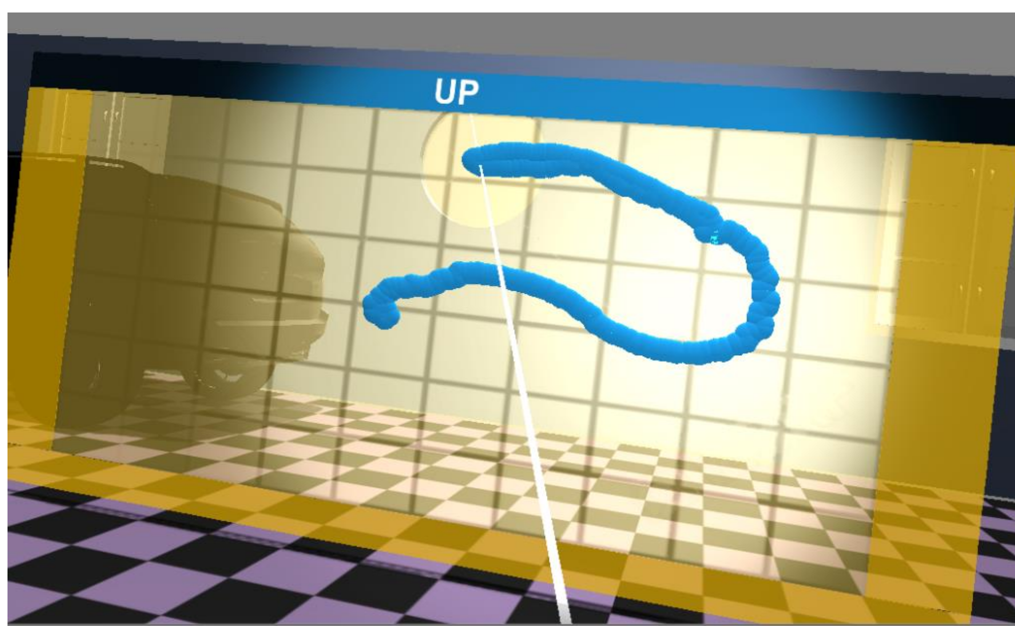

Figura 5. Mecanismo de pintura utilizando el mando de Oculus, con un rayo que parte del mismo.

3) **Pintura con mandos en el aire:** Es similar al mecanismo anterior, pero el trazo aparecerá allí donde se encuentre nuestra mano. Es decir, nos permite pintar en el aire, en lugar de hacerlo sobre una superficie.

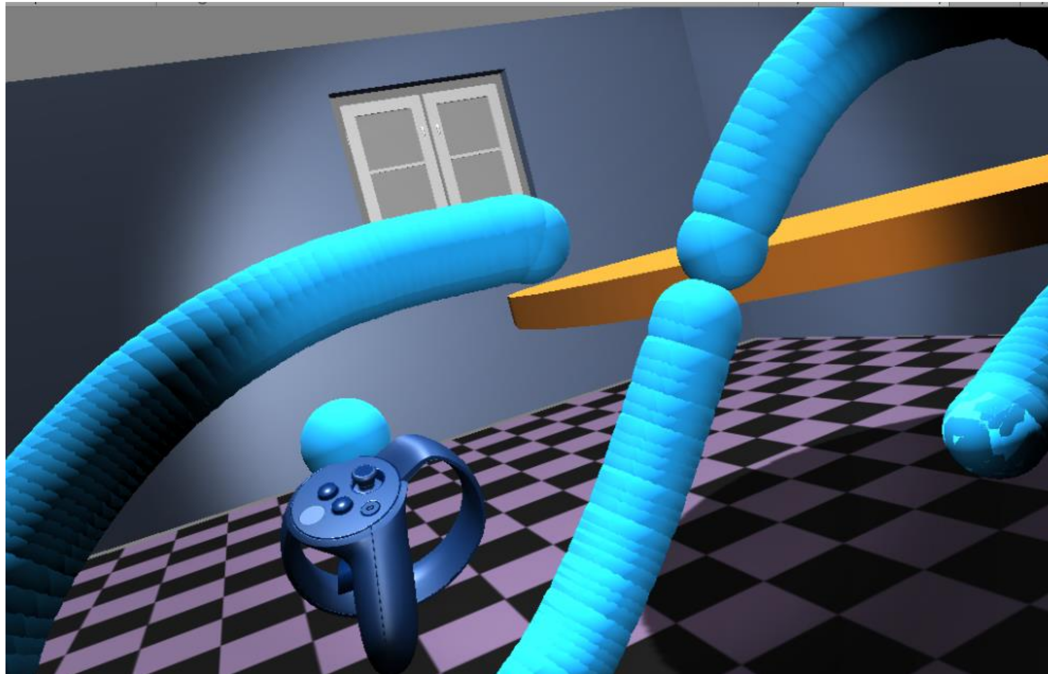

Figura 6. Mecanismo de pintura utilizando el mando de Oculus, generando el trazo en la propia mano.

En los mecanismos 1) y 2), si la pintura se genera sobre la superficie de un modelo, tenemos la opción de que esta queda adherida a él. Es decir, si despueś desplazamos el modelo, la pintura se desplazará con él.

4.2.2 Otras formas básicas de interacción

- **Desplazamiento propio:** Puede desplazarse físicamente, o bien puede utilizar el mando de apoyo. Utilizando el joystick, puede desplazarse fácilmente al punto del entorno que desee.
- **Selección de objetos:** Se puede realizar una serie de acciones con los objetos virtuales (incluyendo tanto los modelos como el lienzo virtual), por lo que tenemos una función para seleccionarlos. Una vez seleccionamos un objeto, podemos:
    - Desplazarlo
    - Rotarlo
    - Modificar su escala
    - Clonarlo
    - Cortarlo

- **Deformación de modelos:** En determinado modo de uso, podemos generar un agujero en la superficie del objeto, con el fin de, además de pintarlo, poder moldearlo.

Figura 7. Interior del modelo de un coche, siendo pintado gracias al mecanismo de la tableta.

En la imagen se puede observar que nos encontramos en el interior del modelo de un coche y estamos pintando el volante. Si nos fijamos, el volante está en contacto con un plano, que se trata del lienzo virtual. También se puede ver el lápiz virtual, desde donde emana la pintura azul.

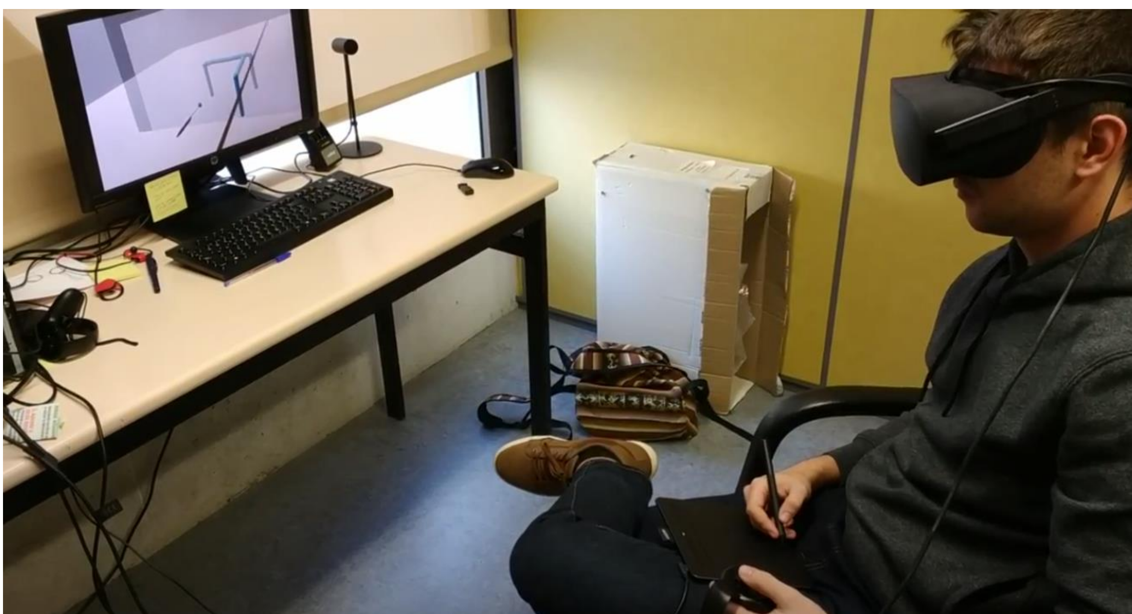

Figura 8. Usuario utilizando la tableta para pintar en la aplicación. Podemos ver en el monitor lo que estaría viendo el usuario.

En esta otra imagen, podemos ver al usuario utilizando la aplicación (vista externa). Si nos fijamos, tiene la tableta Wacom sobre su regazo, el lápiz táctil en su mano derecha y el mando de Oculus en la izquierda.

5. Desarrollo

El desarrollo de la aplicación consiste, básicamente, en idear e implementar el mecanismo que nos permitirá pintar sobre los diferentes modelos del espacio virtual. A partir de ello, podemos ir añadiendo funcionalidades para una mejor experiencia.

Para ello, mediante Unity, creamos una escena en la que inicialmente solo se encontraba el lienzo virtual (un simple rectángulo), junto a un lápiz acoplado a él.

5.1 Adaptación de la tableta Wacom

Antes de nada, es importante tomar en consideración cómo funciona la tableta Wacom. Realmente, está tableta está diseñada para realizar la función de ratón. Una vez la conectamos a nuestro ordenador y realizamos las configuraciones necesarias, el hecho de tocar la tableta con el puntero táctil, es equivalente a hacer clic con el ratón. También podemos desplazar el cursor del ratón, acercando el puntero a la tableta sin tocar. Por último, podemos configurar el uso de los botones con los que cuenta el puntero, para realizar otras acciones básicas (clic derecho, scroll…)

Sabiendo de esta equivalencia, conseguimos, gracias a una librería básica, detectar la posición del puntero de la tableta sobre ella. De tal forma que, cuando el puntero estaba en contacto o cerca de la tableta, nuestro lápiz se colocaba en esa misma posición respecto al lienzo virtual. Si el puntero se encuentra más lejos (en ese caso no podemos detectarlo), entonces nuestro lápiz virtual desaparece.

5.2 Los trazos

A partir de este punto, lo único que necesitábamos para poder dibujar sobre nuestro lienzo, era diseñar la forma en la que se generaban los trazos. Definimos el trazo como el conjunto de gotas de pintura que se genera desde que apoyamos el puntero táctil con el botón pulsado, hasta que, o bien levantamos el puntero, o bien dejamos de pulsar el botón. En el caso de utilizar únicamente los mandos, el trazo se generaría desde que pulsamos el botón de pintar hasta que lo soltamos.

5.2.1 Propiedades del trazo

El trazo cuenta con una serie de propiedades, variables que podemos configurar. Estas son las siguientes:

- **Amplitud:** Llamamos amplitud a la cantidad de superficie que cubre la gota de pintura cuando la colocamos sobre el modelo.
- **Grosor:** El grosor determina el relieve de la gota de pintura, es decir, cuanto sobresale respeto a la superficie sobre la que estamos pintando.
- **Material:** Básicamente designa el color y textura que tendrá el trazo.

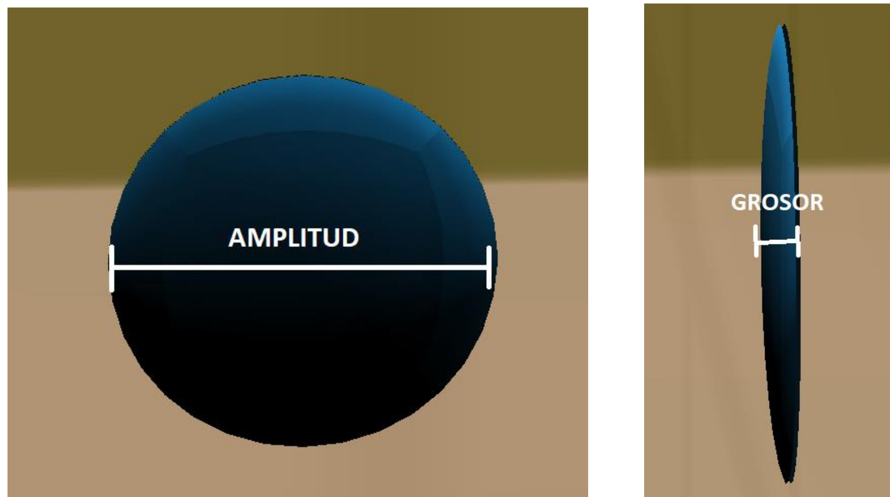

Figura 9. Diferenciación entre la amplitud y el grosor de la gota de pintura.

5.2.2 Esquema de creación de un trazo

La creación de un trazo consta de varias fases:

- **Creación o inicialización del trazo:** Se ejecuta en cuanto cumplimos las condiciones para que se genere un trazo. Consiste simplemente en crear el objeto virtual que representará al trazo. Si tenemos activada la opción de adherir pintura, debemos tener en cuenta sobre qué superficie estamos pintando, para colocar correctamente el trazo en la jerarquía. Los trazos que vamos pintando se van numerando, además se introducen en una pila que nos permitirá deshacerlos y rehacerlos de forma sencilla.

```
protected GameObject CreaNuevoTrazo(Transform trazoPadre)
{
    // Cuando apoyamos el lápiz sobre la tableta, se crea un nuevo trazo (objeto)
    trazoActual = new GameObject();
    // Le asignamos padre y le damos nombre
    if (configurador.adherirPintura && trazoPadre.tag.Equals("Pintable"))
    {
        trazoActual.transform.SetParent(trazoPadre.Find("Trazos"));
        //Debug.Log(trazoActual.transform.parent.name);
        trazoActual.name = trazoActual.transform.parent.parent.name + " - Trazo " + nTrazos;
    }
    else
    {
        trazoActual.transform.SetParent(GameObject.Find("TrazosSueltos").transform);
        trazoActual.name = "Trazo " + nTrazos;
    }
    trazoActual.transform.localPosition = Vector3.zero;


    // Además lo añadimos a la pila de objetos trazados y vaciamos la pila de deshechos
    pilaTrazos.Push(trazoActual);
    pilaTrazosDeshechos.Clear();
    // Aumentamos el número de trazos realizados
    nTrazos++;
    return trazoActual;
}
```

- **Construcción del trazo:** Se trata del proceso en el cual se van generando gotas de pintura y estas son acopladas al trazo. Se ejecuta en cada frame de la ejecución, mientras sigamos pulsando el botón de pintar y siempre que la última gota generada no esté demasiado cerca del punto en el que nos encontramos. Por cada gota generada, esta se acopla a la malla que forma el trazo. Un trazo puede estar generado por varios subtrazos, debido a que, en este proceso de acoplamiento, puede excederse el número máximo de vértices que permiten el correcto funcionamiento de Unity. En esta fase entra en juego el tipo de gotas de pintura que formarán el trazo. Después se explicarán los tipos de trazo en base a ello.

- **Cierre del trazo:** Ocurre en cuanto dejamos de pulsar el botón de pintar, o bien levantamos el puntero táctil (si estamos usando la tableta). Básicamente ordenamos todos los subtrazos generados y le hacemos saber al programa que la próxima vez que pintemos, tendremos que generar un nuevo trazo.

```
public override void TerminarTrazo() {
    // Cuando levantamos el lápiz, terminamos (cerramos) el trazo que estábamos realizando
    // En primer lugar, acoplamos la última gota creada
    AcoplarGota(ultimaGota);
    // Centramos el objeto correspondiente al subtrazo,
    PosicionaSubtrazo();
    foreach (GameObject subtrazo in subtrazos) {
        // Por cada subtrazo del trazo actual, añadimos un MeshCollider con su malla
        if (!configurador.adherirPintura) {
            subtrazo.AddComponent<MeshCollider>();
            subtrazo.GetComponent<MeshCollider>().convex = true;
            subtrazo.GetComponent<MeshCollider>().isTrigger = true;
        }
        subtrazo.transform.SetParent(trazoActual.transform);
    }
    // Destruímos la última gota del trazo
    Destroy(ultimaGota);
    trazoActual = null;
    subtrazoActual = null;
    subtrazos.Clear();
    listaPosGotas.Clear();
    pintando = false;
}
```

5.2.3 Tipos de trazos

Como se ha mencionado anteriormente, el trazo no es más que un conjunto de “gotas de pintura” que se agrupan en un solo objeto. Los subtipos de trazo que hemos diseñado, dependen precisamente de cómo se generan dichas gotas de pintura. Comenzamos explicando el primer tipo de trazo diseñado. Los sucesivos tratan de mejorar a los anteriores.

a) **Trazo formado por esferas sencillas:** En primer lugar, planteamos la posibilidad de que el trazo que pintamos estuviera compuesto de esferas básicas, que se fueran colocando en el espacio allí por donde pasáramos el lápiz (puntero). Este método es muy sencillo, pues en Unity las esferas se generan fácilmente con una función predefinida.

```
protected virtual void NuevaGota(Vector3 posicion) {
    GameObject gota;
    if (configurador.pintaEsferas) {
        gota = GameObject.CreatePrimitive(PrimitiveType.Sphere);

        // A cada esfera le asignamos el tamaño, posición, material y el padre
        gota.transform.localScale = new Vector3(
            configurador.diametroGota,
            configurador.diametroGota,
            configurador.diametroGota);

        gota.transform.position = posicion;
        gota.GetComponent<MeshRenderer>().material = configurador.materialesPintura[colorActual];
        gota.transform.SetParent(subtrazoActual.transform);
    }
```

El principal problema surgía cuando realizábamos un trazo de forma excesivamente rápida. Lo que ocurría era que no se generaba en suficiente número de esferas y quedaban huecos en el trazo.

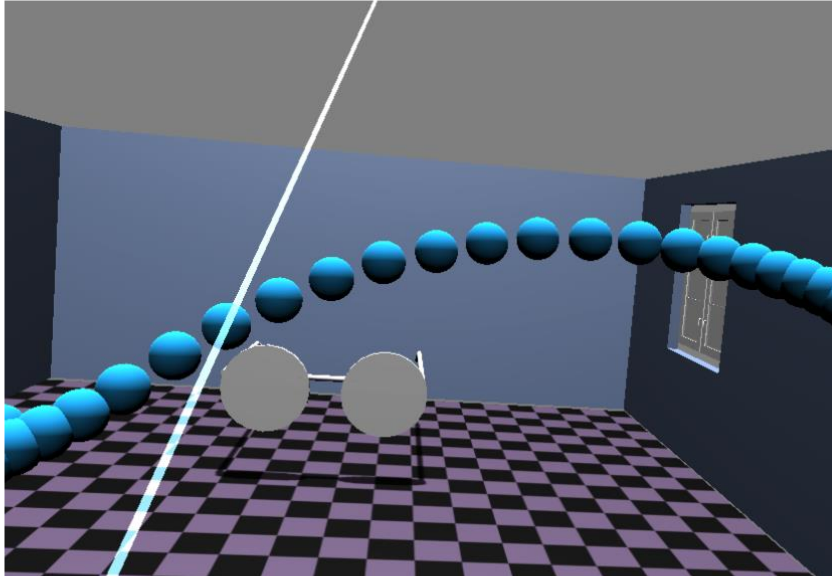

Figura 10. Trazo formado por esferas, que se han separado debido a la velocidad.

Una posible solución fue colocar esferas intermedias entre las esferas consecutivas del trazo, en mayor cantidad cuanto mayor fuera la distancia entre ellas.

```
if (configurador.completaTrazo) {
    Vector3 aux = ultimaPosicion;
    int nGotas = (int)(distancia / (configurador.diametroGota / 5));
    for (int i = 1; i <= nGotas; i++) {
        NuevaGota(aux + i * separacion / nGotas);
    }
}
```

Es tan sencillo como realizar un bucle de llamadas a la función NuevaGota, que recibe como parámetro la posición en la que colocaremos cada esfera.

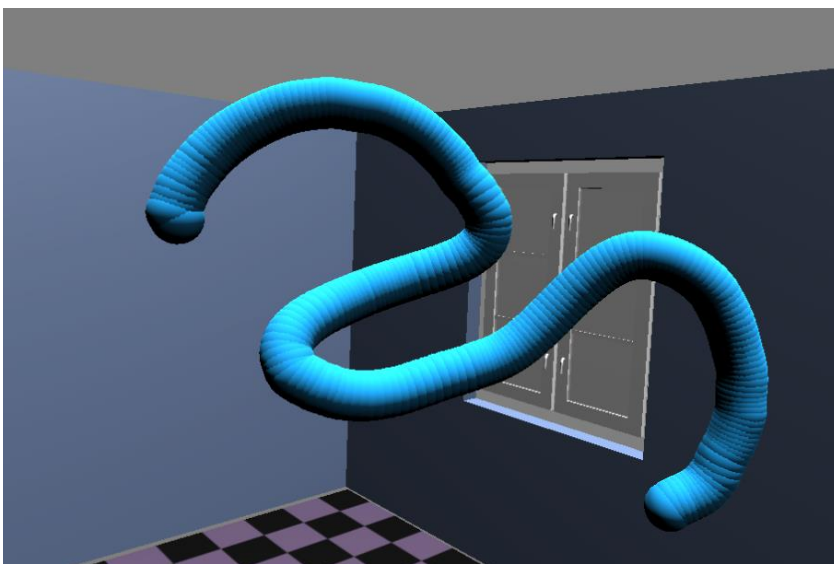

Figura 11. Trazo formado por esferas, con esferas intermedias.

b) **Trazo formado por esferas y cilindros:** Este segundo método trata de solucionar el problema mencionado en el anterior a cerca de la separación entre las esferas que componen el trazo. Se trata de colocar cilindros a modo de tubos que unen dos gotas (esferas) consecutivas. De esta manera, se rellenan los huecos que quedaban entre esferas consecutivas. El aspecto que conseguimos con este tipo de trazo es más uniforme, pero es cierto que ralentiza ligeramente la ejecución del programa.

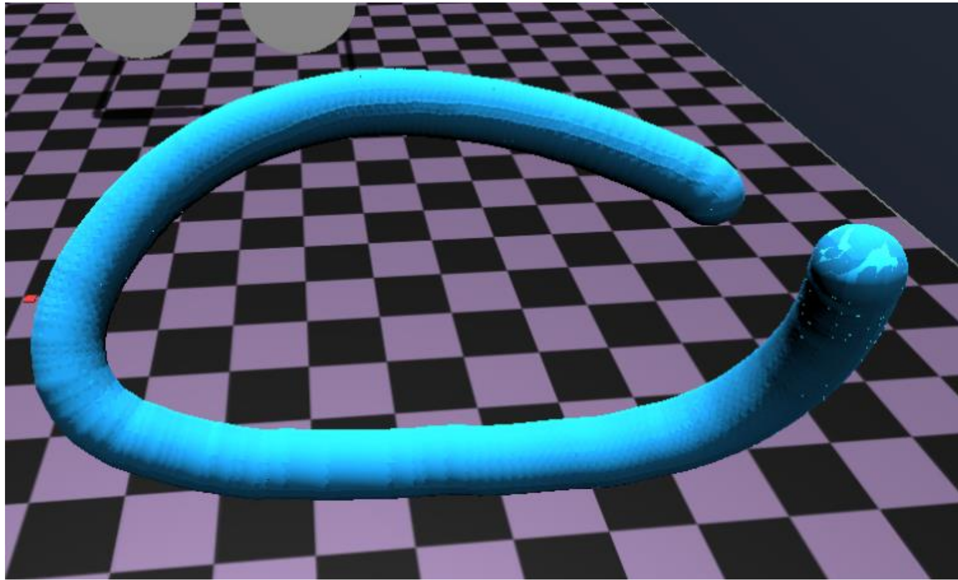

Figura 12. Trazo formado por esferas, con cilindros que conectan esferas.

c) **Trazo formado por gotas:** El objetivo de este tipo de trazo es encontrar una apariencia más similar a la pintura cuando se coloca sobre una superficie en la vida real. Para ello, lo que hacemos es "aplastar" una esfera para que tenga forma de gota de pintura.

```
GameObject proyeccion = InicializaProyeccion(posicion);
GameObject saliente = InicializaProyeccion(posicion);

// Definimos cada una de las mallas
Mesh mallaProyeccion = new Mesh();                         // Malla de la proyecc
Mesh mallaSaliente = new Mesh();                           // Malla saliente de l

// Variables para construir las mallas
List<Vector3> verticesProyeccion = new List<Vector3>();     // Vertices correspond
List<Vector3> verticesSaliente = new List<Vector3>();       // Vértices correspond
List<int> mapeoVertices = new List<int>();                  // Aquí mapearemos los
List<int> triangulosProyeccion = new List<int>();           // Lista de triángulos

// Proyectamos los vértices del foco proyector sobre la superficie
ProyectaVertices(verticesProyeccion, verticesSaliente, mapeoVertices, posicion);
// Ahora creamos los triángulos
AsignaTriangulosGota(triangulosProyeccion, mapeoVertices);

// A cada malla le damos sus vértices y triángulos
mallaProyeccion.SetVertices(verticesProyeccion);
mallaProyeccion.SetTriangles(triangulosProyeccion, 0);
mallaSaliente.SetVertices(verticesSaliente);
mallaSaliente.SetTriangles(triangulosProyeccion, 0);

// A cada parte de la gota le damos su malla
proyeccion.GetComponent<MeshFilter>().mesh = mallaProyeccion;
saliente.GetComponent<MeshFilter>().mesh = mallaSaliente;

// Unimos ambas partes
gota = FusionaPartesGota(proyeccion, saliente);
```

Proyectamos los vértices de la esfera sobre la superficie en la que estamos pintando. Debemos tener en cuenta que podemos estar pintando sobre una superficie y que el trazo cambie después a otra. Por ejemplo, si estamos pintando sobre el lienzo, pero en medio tenemos un modelo que podemos pintar, al pasar el puntero por encima del modelo, cambiamos de superficie. Cada gota se colocará siempre sobre la superficie cercana más cercana al puntero. Después le damos cierta profundidad a los vértices, mayor cuando más nos acercamos al centro de la forma y menor en la zona cercana al perímetro. Se forma así una nueva malla. Con esto conseguimos un aspecto bastante parecido a una gota de pintura. Además, tratamos de que la textura del material que forma la gota sea similar al de la pintura real.

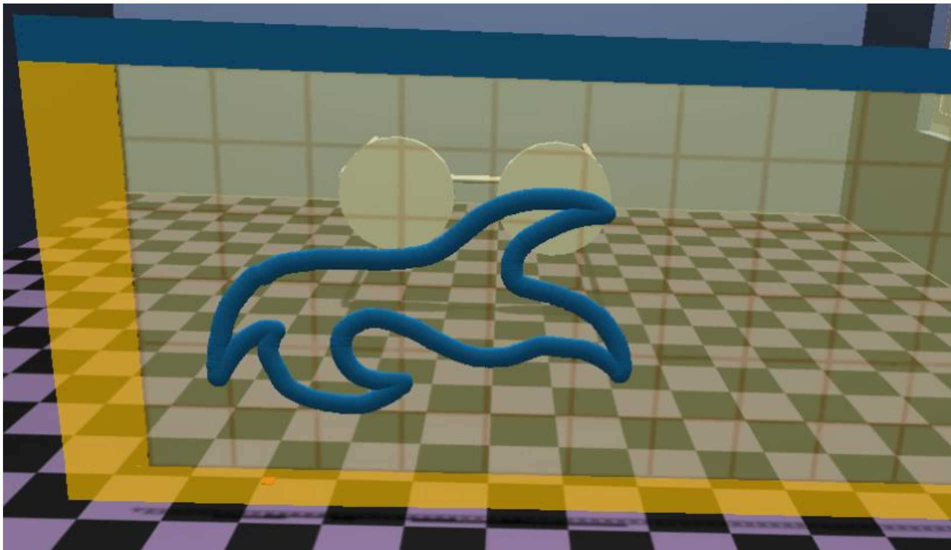

Figura 13. Trazo formado por gotas generadas mediante proyección.

d) **Trazado sobre modelos:** Llegó el momento en el que tuvimos que encontrar la forma de aplicar el sistema que habíamos diseñado para pintar sobre los modelos. Al fin y al cabo, el diseño sobre los mismos era la finalidad principal de la aplicación. Para ello, utilizamos un collider para saber cuándo el lápiz virtual entraba en contacto con el objeto. En ese caso se lanzaba un rayo desde el extremo contrario a la punta del lápiz, de manera que se detectaba la superficie del objeto y la pintura se colocaba sobre ella. En la jerarquía, esos trazos de pintura se generan como hijos del objeto en cuestión, de manera que quedan adheridos a este.

Figura 14. Modelo de una cabeza siendo pintada. Se trata de diseñar un casco.

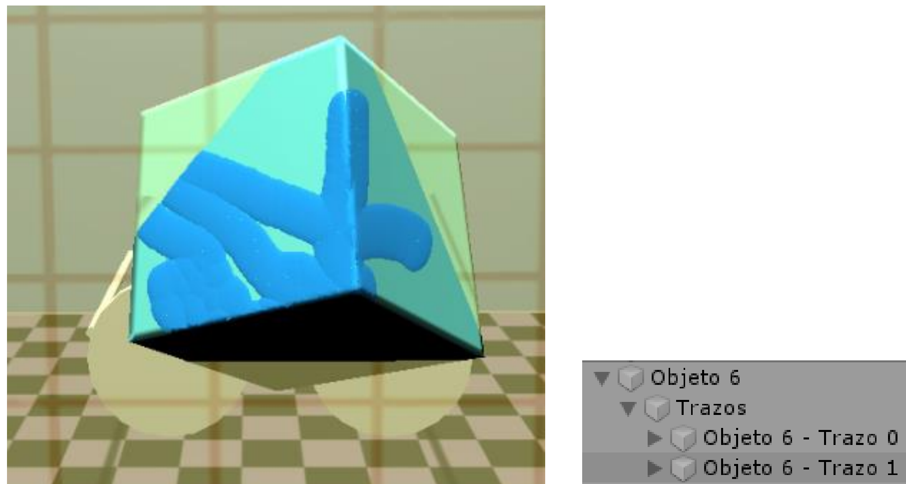

Figura 15. Modelo de un cubo siendo pintado. A la derecha aparece la jerarquía con el modelo y los trazos que contiene.

5.2.3 Acoplamiento de gotas en los trazos

Como hemos mencionado anteriormente, las gotas que forman el trazo se van acoplando en una misma malla, para poder darle al trazo sus componentes físicas (meshcolider). Por ello, vamos a explicar la función que utilizamos para efectuar esta tarea.

```
protected void AcoplarGota(GameObject gota) {
    GameObject capsula = GameObject.CreatePrimitive(PrimitiveType.Cylinder);
    GameObject[] objetos;
    if (configurador.pintaCilindros)
    {
        capsula.transform.localScale = new Vector3(
            configurador.diametroGota,
            configurador.diametroGota,
            configurador.diametroGota);
        capsula.GetComponent<MeshRenderer>().material = configurador.materialesPintura[colorActual];
        capsula.transform.SetParent(subtrazoActual.transform);

        Vector3 start = ultimaGota.transform.position;
        Vector3 end = gota.transform.position;
        capsula.transform.position = (end - start) / 2 + start;

        Vector3 v3T = capsula.transform.localScale;
        v3T.y = (end - start).magnitude / 2;
        capsula.transform.localScale = v3T;

        capsula.transform.rotation = Quaternion.FromToRotation(Vector3.up, end - start);
        objetos = new GameObject[] { subtrazoActual, gota, capsula };
    }
    // Combinamos la malla del subtrazo actual con la de la gota
    else objetos = new GameObject[] { subtrazoActual, gota };
    subtrazoActual.GetComponent<MeshFilter>().mesh.CombineMeshes(CombinaMallas(objetos, true));

    // Destruimos la gota, ya que sus vértices y triángulos ya están en la malla


    // Si en la malla del subtrazo hay más vértices de lo permitido, terminamos el subtrazo y creamos
    int nVertices = subtrazoActual.GetComponent<MeshFilter>().mesh.vertexCount;
    if (nVertices > 65000) {
        PosicionaSubtrazo();
        CreaNuevoSubTrazo(subtrazoActual.transform.parent);
    }

    Destroy(ultimaGota);
    Destroy(capsula);
}
```

En todo momento, tenemos guardado el tramo de trazo que hemos dibujado hasta ahora. La función CombineMeshes nos permite combinar mallas de diferentes objetos, por lo que la utilizamos para que la malla de la nueva gota se combine con la malla del trazo actual.

La función se vuelve más compleja si tenemos activada la opción de completar los trazos mediante cilindros entre las gotas. En ese caso, creamos el cilindro (más bien es una cápsula), teniendo en cuenta las dimensiones, posición y orientación que lo hacen encajar perfectamente entre la gota actual y la anterior. Después, el cilindro también se adhiere a la malla del trazo, junto con la nueva gota de pintura.

Debemos tener en cuenta que, en el lenguaje en el que estamos trabajando, las mallas tienen un límite en la cantidad de vértices que pueden asumir. Esto es debido a que, los índices de estos vértices son del tipo *int*, que tiene un rango de algo más de 65.000 posibles valores. Si excediéramos esta cantidad de vértices, experimentaríamos un problema de *overflow*. Por ello, lo que hacemos es dividir los trazos en varios subtrazos. Cuando el primer subtrazo alcanza los mencionados 65.000 vértices, entonces lo cerramos y creamos un segundo subtrazo, con una

nueva malla. Cada subtrazo se coloca en la jerarquía como hijo del objeto que representa al trazo padre.

Por último, destruimos la gota generada, así como la cápsula en caso de haber sido creada. Ya hemos acoplado su malla al trazo, ya no necesitamos esa información.

5.2.2 Trazo como modificación del *shader* del modelo

Ideamos también otro tipo de trazo, el cual consiste en modificar el *shader* del modelo sobre el cual estamos pintando. El *shader* podría definirse como el conjunto de instrucciones que sigue la computadora para establecer el aspecto de la superficie del modelo. Por tanto, podemos modificar estas instrucciones para colorear, modificar brillos, sombras… Nuestra idea es tan simple como modificar el color de la parte de la superficie sobre la que pintamos. Aquí pueden observarse algunos ejemplos de lo que se puede conseguir manipulando el *shader*.

Como se puede observar, la sensación que obtenemos es diferente. Mientras que en el otro tipo de trazo conseguíamos una pintura con relieve, aquí la pintura es completamente lisa.

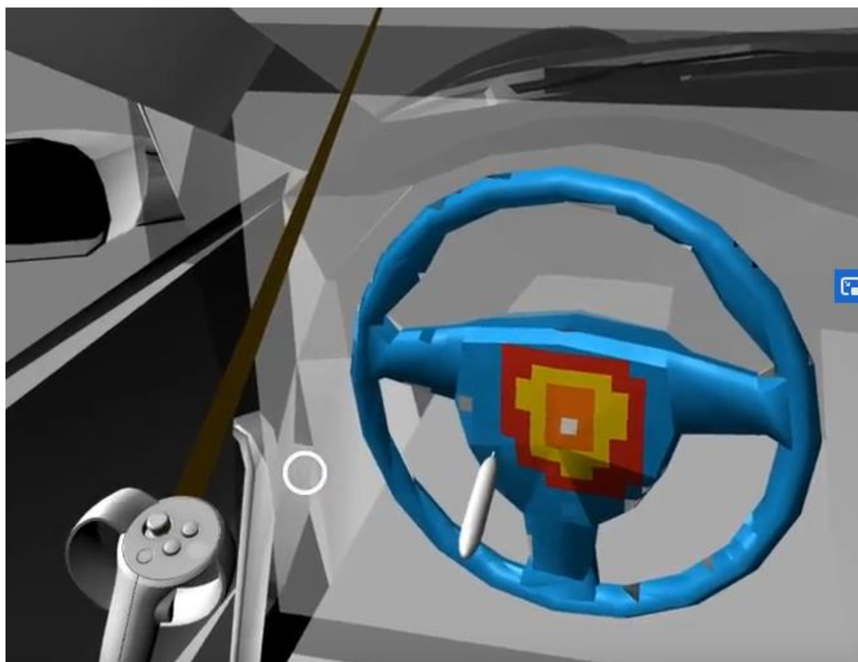

Figura 16. Modelo del interior de un coche. Se está utilizando la aplicación para pintar el volante mediante la modificación del shader del modelo.

5.3 Manipulación de objetos

Como bien se ha explicado anteriormente, los objetos que se encuentran en el espacio virtual pueden ser manipulados utilizando el mando de apoyo. Para ello, necesitábamos generar un mecanismo para seleccionar los objetos. Lo que hicimos fue colocar un rayo que emerge de la representación virtual del mando, en la dirección hacia la que estamos apuntando. Cuando este rayo intersecta con un

objeto, este se ilumina, cambia de color, lo cual quiere decir que podemos seleccionarlo pulsando un botón.

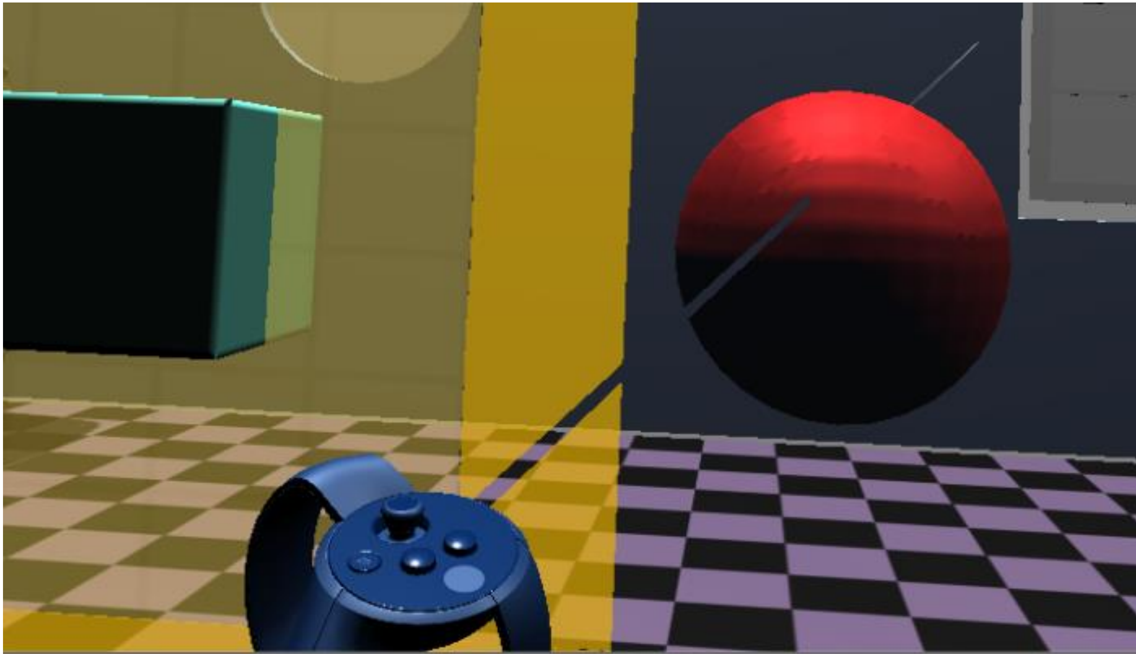

Figura 17. Modelo de una esfera siendo seleccionado, se resalta en color rojo.

Mientras mantengamos pulsado ese botón, el objeto se mantiene seleccionado, entonces podremos hacer lo siguiente con él:

- **Desplazarlo:** Simplemente debemos mover el mando mientras el objeto está seleccionado. Este último realizará un desplazamiento equivalente a ese movimiento.
- **Rotarlo:** De igual modo, si rotamos el mando mientras el objeto esté seleccionado, este último también rotará.
- **Cambiar su escala:** Mientras el objeto está seleccionado, podemos mover el joystick hacia la izquierda para reducir su escala, o hacia la derecha para aumentarla.
- **Cambiar su cercanía:** Si movemos el joystick hacia adelante, el objeto se alejará progresivamente, mientras que, si lo movemos hacia atrás, el objeto se acercará.
- **Clonarlo:** Simplemente debemos pulsar el botón de "Clonar" mientras el objeto está seleccionado. Podremos seguir moviendo el objeto, el clon se mantendrá en el lugar que estaba el objeto en el momento de la clonación.

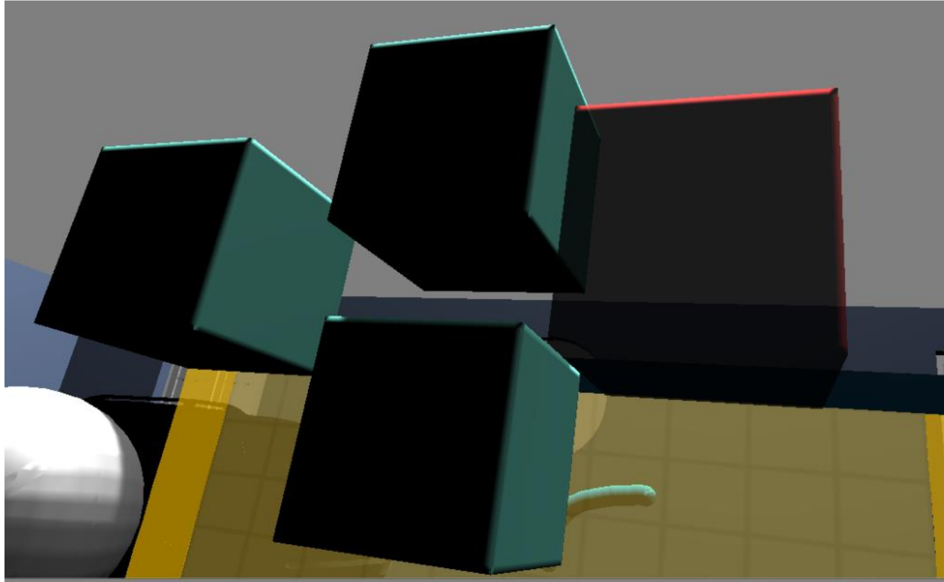

Figura 18. Modelo de un cubo siendo clonado varias veces.

Cabe destacar que hemos generado un modo en el que podemos rotar y desplazar el objeto por separado, utilizando diferentes botones. Esto nos permite hacer una de las dos acciones sin realizar la otra involuntariamente, lo cual puede resultar molesto en algunas ocasiones.

En el caso del lienzo virtual, no es necesario que el rayo intersecte con él para realizar desplazamientos o rotaciones. Simplemente, si el rayo no intersecta con ningún otro objeto seleccionable, podremos pulsar el botón indicado para realizar las acciones antes descritas (el lienzo no puede cambiar su escala ni ser clonado, pero si puede verse afectado por el resto de funciones).

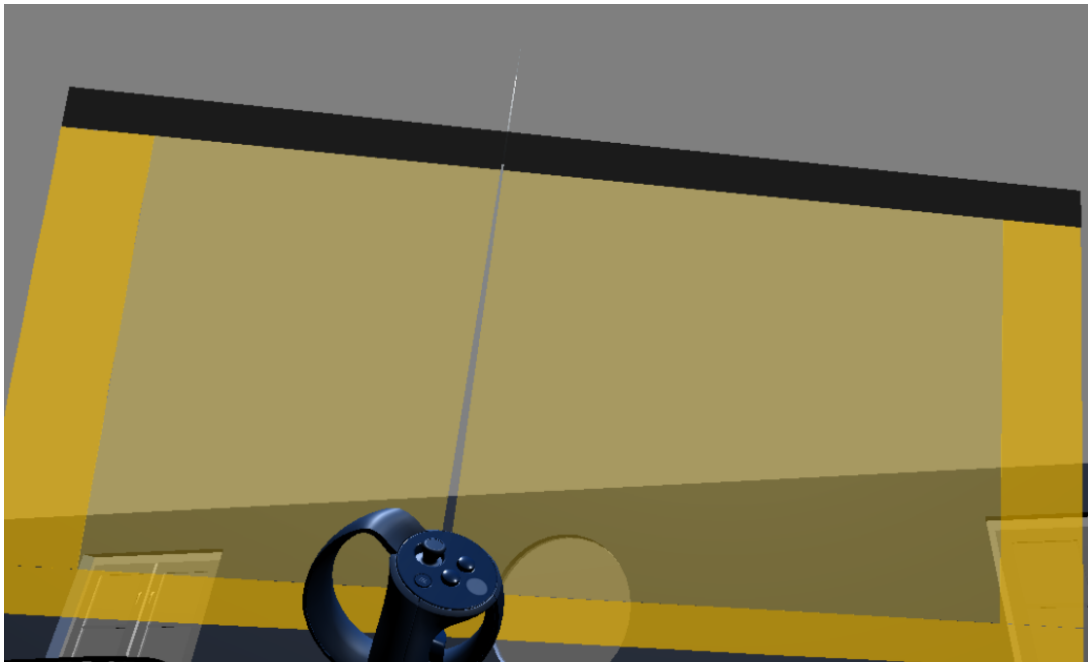

Figura 18. Lienzo virtual siendo desplazado.

5.4 Deformación de modelos

Aprovechando las cualidades de la tableta Wacom, quisimos diseñar un mecanismo de deformación de objetos. Y es que esta tableta es capaz de detectar la presión que realizamos sobre ella con el lápiz táctil. El lápiz tiene además un botón, que nos permite diferenciar cuando queremos pintar (al pulsarlo) o cuando simplemente queremos moldear el objeto que estamos tocando.

El mecanismo consiste en lo siguiente: cuando el lápiz toca al objeto, si ejercemos la presión suficiente sin pulsar el botón (el 50% de la presión máxima detectada) entonces la malla que forma al objeto comienza a hundirse. Cada vértice de la malla se hunde más cuanto más cerca se encuentre del punto en el que está tocando el lápiz.

```
public void DeformarVertices(RaycastHit hit, float presion)
{
    List<Vector3> desplazados = new List<Vector3>(mallaDeformable.vertices);
    int numeroVertices = desplazados.Count;
    for (int i = 0; i < numeroVertices; i++)
    {
        Vector3 vertice = desplazados[i];
        // Calculamos la distancia del vértice al punto de contacto
        float distancia = Vector3.Distance(hit.point, transform.TransformPoint(vertice));
        // Solo hay desplazamiento si el vértice está lo bastante cerca
        if (distancia < radioDeformacion)
        {
            desplazados[i] = DesplazarVertice(vertice, distancia, presion);
        }
    }
    mallaDeformable.SetVertices(desplazados);
    mallaDeformable.RecalculateNormals();
    Destroy(GetComponent<MeshCollider>());
    gameObject.AddComponent<MeshCollider>();
    GetComponent<MeshCollider>().convex = true;
    GetComponent<MeshCollider>().isTrigger = true;
}
```

Lo que hacemos es tomar un radio de deformación, lo cual quiere decir que, dentro de la malla del modelo que estamos deformando, solo podrán desplazarse los vértices que se encuentren a una distancia del punto de presión menor o igual que ese radio.

```
private Vector3 DesplazarVertice(Vector3 original, float distancia, float presion)
{
    /****************************************************************************
     *  El factor de desplazamiento viene dado por la siguiente formula:
     *  min {
     *      maximaProfundidad,
     *      max {
     *          0,
     *          (10 * presión * (radioDeformación - distancia))^2
     *      }
     *  }
     * *********************************************************************/
    //float factor = maximaProfundidad;
    float factor = Mathf.Pow(10 * presion * (radioDeformacion - distancia), 2);
    factor = Mathf.Max(factor, 0);
    if (factor > 0)
        factor = Mathf.Min(factor, radioDeformacion);

    Vector3 nuevo = transform.InverseTransformPoint(transform.TransformPoint(original) - vistaPrevia.up * factor);
    return nuevo;
}
```

Esos vértices, los que se encuentran dentro del radio de deformación, se desplazan según el factor:

$$Mínimo(maximaProfundidad, Máximo(0, presión * (radioDeformación - distancia))^2))$$

Es decir, que tenemos un índice de deformación máxima y el índice de cada vértice depende de la presión ejercida con el lápiz y del cuadrado de la distancia al centro de deformación.

Cabe destacar que los modelos que empleamos tienen que tener una buena densidad de vértices para que el resultado sea el esperado. Por ejemplo, un cubo estándar en Unity está formado por únicamente 8 vértices, resultando 2 triángulos en cada cara. Si realizamos la deformación en ese cubo, al solo moverse los vértices, el resultado sería algo como esto:

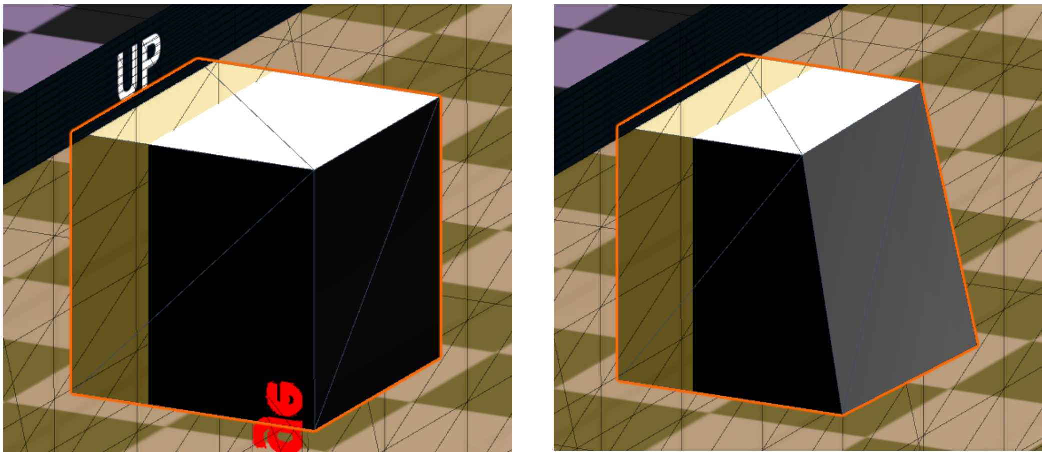

Figura 19. Deformación de un cubo básico.

Se observa que únicamente hemos podido aplastar las esquinas, que es donde se encuentran los vértices. Por ello, hemos creado un script que se encarga de agregar muchos más vértices a los modelos con los que contamos.

```
void DividirTriangulos()
{
    int cuentaIteraciones = 0;
    int mallaCompleta;
    do
    {
        mallaCompleta = 0;
        for (int j = 0; j < malla.triangles.Length; j += 3)
        {
            Vector3 v1 = malla.vertices[malla.triangles[j]];
            Vector3 v2 = malla.vertices[malla.triangles[j + 1]];
            Vector3 v3 = malla.vertices[malla.triangles[j + 2]];
            Triangulo t = new Triangulo(v1, v2, v3);
            if (t.area > areaMaxima)
            {
                ParteTriangulo2(t, j);
                if (t.area / 2 < areaMaxima) mallaCompleta += 2;
            }
            else
            {
                mallaCompleta++;
            }
        }
        cuentaIteraciones++;
        if (cuentaIteraciones > 10)
        {
            break;
        }
    } while (mallaCompleta < malla.triangles.Length);
```

```
void ParteTriangulo2(Triangulo t, int j)
{
    Vector3 nuevo = t.PuntoDivision();

    List<Vector3> verticesActualizados = new List<Vector3>(malla.vertices);
    int antiguoIndice;
    int nuevoIndice;
    antiguoIndice = malla.triangles[j];
    int indice = verticesActualizados.IndexOf(nuevo);
    if (indice > 0)
    {
        nuevoIndice = indice;
    }
    else
    {
        nuevoIndice = verticesActualizados.Count;
        verticesActualizados.Add(nuevo);
    }

    malla.SetVertices(verticesActualizados);

    List<int> triangulosActualizados = new List<int>(malla.triangles);

    triangulosActualizados[j + 2] = nuevoIndice;
    triangulosActualizados.InsertRange(j + 3, new int[] { antiguoIndice,nuevoIndice, antiguoIndice+2 });


    malla.SetTriangles(triangulosActualizados, 0);
}
```

El funcionamiento es muy sencillo: de forma recurrente, se toman los triángulos que forman la malla del modelo y se dividen en dos. Un triángulo está formado por tres vértices. Se agrega uno más y se componen dos triángulos que se sitúan uno

junto a otro. Este proceso se repite hasta que consideramos que los triángulos son lo bastante pequeños.

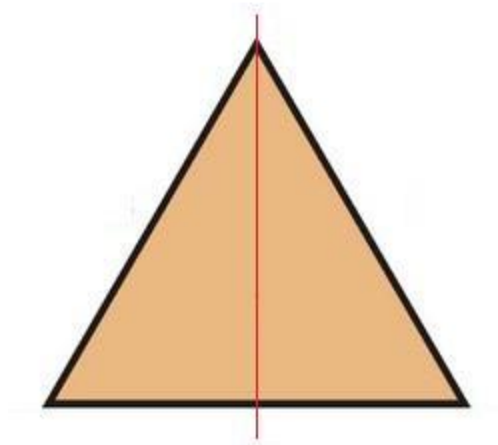

Figura 20. División de un triángulo de la malla.

Si aplicáramos este algoritmo a un cubo sencillo, y después ejerciéramos presión sobre él, obtendríamos el siguiente resultado:

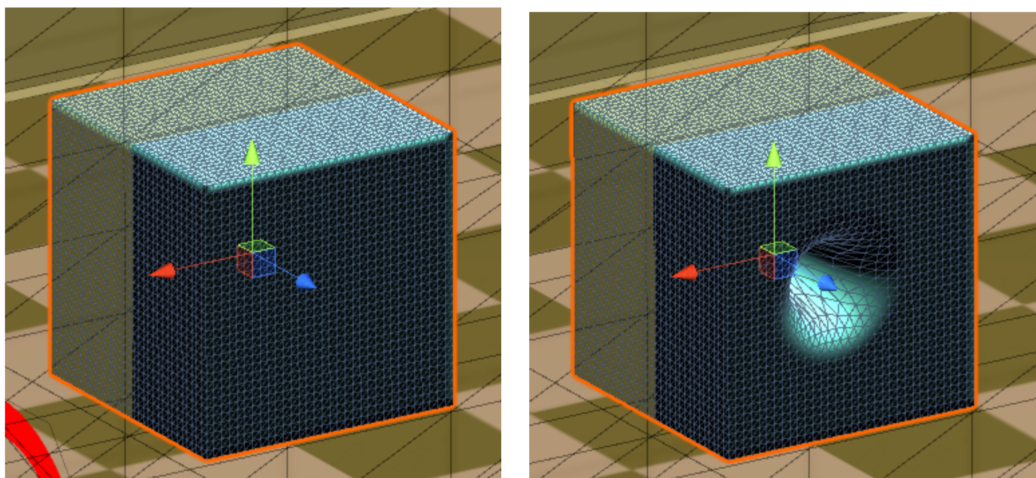

Figura 21. Deformación de un cubo complejo.

El resultado que deseábamos en un principio. Como se observa, hay muchos más vértices y por tanto la deformación es más suave, ya que cada vértice tiene un desplazamiento diferente.

5.5 Borrado de trazos

La aplicación cuenta también con un sistema para borrar tanto los modelos como los trazos que pintamos. Así pues, cada trazo generado se considera también un objeto, sobre el cual podemos aplicar los mecanismos que se explican en el apartado 5.3 Manipulación de objetos. Pero, en el caso de los trazos de pintura, tenemos además la potestad de borrarlos, por si nos equivocamos. Simplemente debemos pulsar el botón de "Borrar" mientras el trazo en cuestión está seleccionado. A nivel de código, simplemente se destruye el objeto.

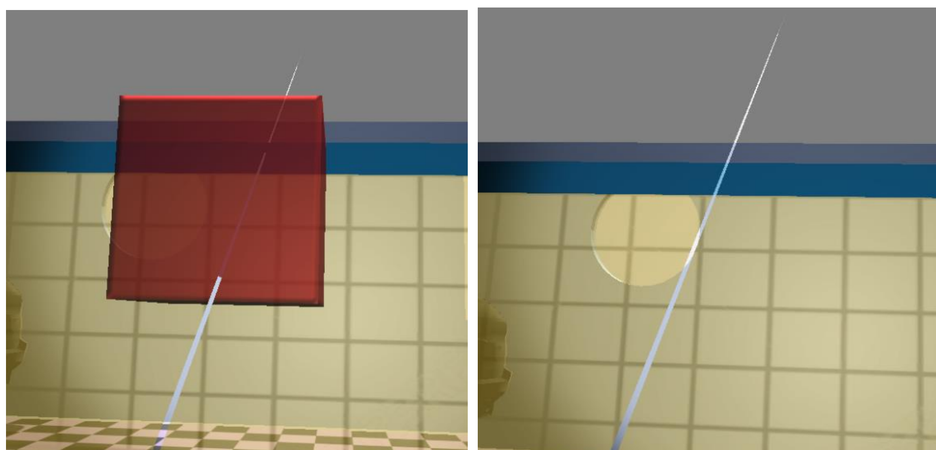

Figura 22. Modelo de un cubo siendo borrado.

Existe también la posibilidad de borrar rápidamente el último trazo creado (sin necesidad de seleccionarlo), pulsando el botón de “Deshacer”. Del mismo modo, podemos pulsar el botón de “Rehacer” para que vuelva a aparecer el último trazo que hemos deshecho. Este mecanismo es posible gracias a que todos los trazos realizados se encuentran en una pila. Al deshacerse, pasan a otra pila que se vacía en cuanto realizamos un nuevo trazo (ya no podríamos rehacer)

5.6 Desplazamiento del jugador

Generalmente, el jugador debe desplazarse por la escena virtual para realizar los trazos que desea de forma cómoda. Cuando el jugador, llevando las Oculus puestas, realiza un desplazamiento en el espacio físico real, este desplazamiento se traslada también al espacio virtual. El problema es que el espacio físico del que disponemos puede no ser lo bastante grande como para realizar el diseño que queremos de forma cómoda.

Por ello, hemos creado un par de mecanismos para poder realizar el desplazamiento del jugador en el espacio virtual sin necesidad de desplazarse en el espacio físico. Contamos con dos mecanismos diferenciados:

- **Desplazamiento mediante rayo:** Utilizamos el rayo que emerge del mando de apoyo. Si apuntamos al suelo de la escena y pulsamos el botón correspondiente, entonces nos desplazaremos progresivamente al punto donde el rayo corta al suelo.

- **Desplazamiento mediante joystick:** Simplemente utilizamos el Joystick para movernos en las diferentes direcciones por el plano del suelo virtual. Hemos observado que este mecanismo es más fluido, pero también genera un mayor mareo.

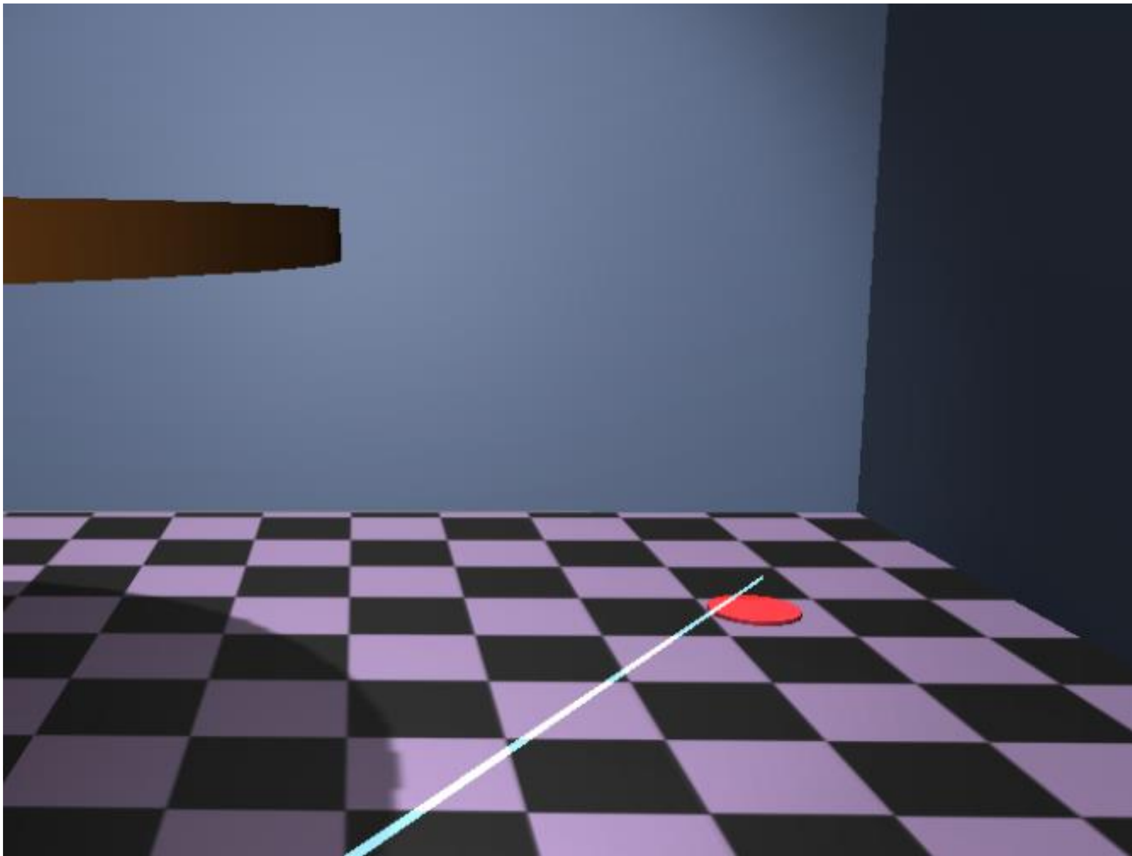

Figura 23. Desplazamiento mediante rayo que apunta al suelo.

5.7 Guardado de escena

Conforme vamos avanzando en nuestro trabajo, la aplicación nos permite guardar los trazos de la escena tal y como se encuentra. El mecanismo que hemos diseñado para ello, consiste en guardar en un fichero la información de los trazos; debemos saber la información de las mallas que los componen: la posición de cada vértice y la organización de los triángulos que forman dichos vértices. El fichero se guarda con el nombre que el usuario decida.

Una vez guardado el fichero con la información de la escena, podemos recargarlo, seleccionando el nombre del fichero correspondiente. De esta manera, podemos realizar un dibujo muy costoso repartiendo el trabajo en diferentes momentos.

5.8 Configurador

Unity nos permite crear un script de configuración, que nos permita hacer ajustes de forma muy rápida mediante variables públicas a las que podemos cambiar el valor antes de realizar la ejecución. En nuestro caso, tenemos unas cuantas variables que podemos modificar gracias al configurador que hemos creado

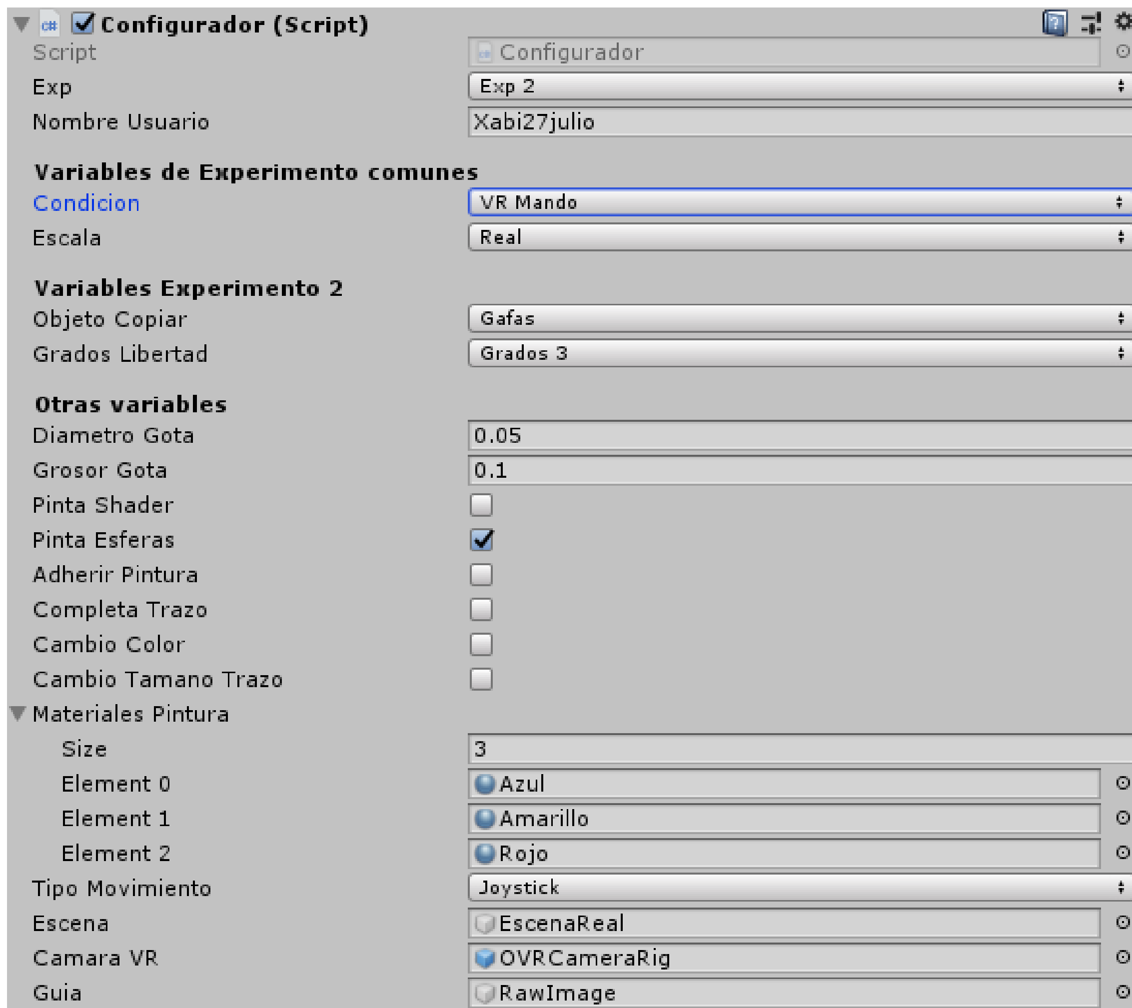

Figura 24. Cuadro de configuración de la aplicación.

Las variables más destacadas que hemos querido configurar son las siguientes:

- **Condición:** Hace referencia al mecanismo de pintura que se utilizará, los mecanismos están descritos en el apartado 4.1 de esta memoria.
- **Amplitud de la gota de pintura:** Podemos ajustar cuál será la amplitud al inicio de la ejecución, pero también es verdad que podremos cambiar el tamaño en el transcurso de la misma. Solo debemos utilizar el mando de apoyo, moviendo el Joystick a la derecha para aumentar el diámetro y hacia la izquierda para disminuirlo.
- **Grosor de la gota de pintura:** Ocurre igual que con la amplitud. Establecemos el grosor inicial, después podemos modificarlo utilizando el Joystick del mando de apoyo. Hacia adelante el grosor aumenta, mientras que, hacia atrás, el grosor disminuye.
- **Pintar shader:** Se trata de una variable booleana. Cuando esté activada, se empleará el tipo de trazo como modificación del shader de la superficie. En caso contrario, se usará el tipo de trazo como objeto.

- **Pintar esferas:** Solo funciona si la opción de “pintar shader” está desactivada. Cuando está activada, el trazo que realizamos está formado por esferas sencillas, en caso contrario realizaremos el modelo de gota más complejo utilizando la proyección.
- **Completar trazo:** Solo es relevante cuando está activada la opción de “pintar esferas”. Hay que tener en cuenta que, según la capacidad de nuestra máquina, las esferas pueden no generarse al mismo ritmo al que estamos realizando el trazo, por tanto, pueden existir huecos dentro del este. Si activamos esta opción, se generarán automáticamente esferas nuevas entre las esferas consecutivas del trazo. En mayor cantidad cuanto mayor sea la distancia entre ellas.
- **Adherir pintura:** Solo funciona si la opción de “pintar shader” está desactivada. Tenemos la opción de que, cuando pintamos sobre la superficie de un modelo, el trazo quede adherido a ese modelo, o bien quede libre.
- **Materiales de la pintura:** Se trata de un array compuesto por objetos de tipo Material. Podemos introducir cuantos materiales queramos, con el objetivo de luego poder alternar dichos materiales para componer los trazos que realizamos. Generalmente los utilizamos para modificar el color del trazo.
- **Tipo de desplazamiento:** Determina si utilizamos el desplazamiento propio mediante Joystick, o bien mediante rayo.

En la imagen se observan también otras variables, que hemos utilizado para realizar experimentos para otra asignatura.

5.8 Otras funcionalidades

La aplicación cuenta con otras funcionalidades más secundarias, pero que hacen que sea más completa:

5.8.1 Cambio de color

Para cambiar el color del trazo que estamos realizando, hemos seguido un mecanismo muy sencillo. Tenemos una lista de colores predeterminados y podemos pasar de uno a otro realizando una acción. Para cambiar de un color al siguiente, tenemos que pulsar dos veces el botón del puntero táctil, en el caso de que estemos utilizando la tableta para pintar. En caso de estar pintando con el mando, utilizaremos el botón del Joystick del mando de apoyo.

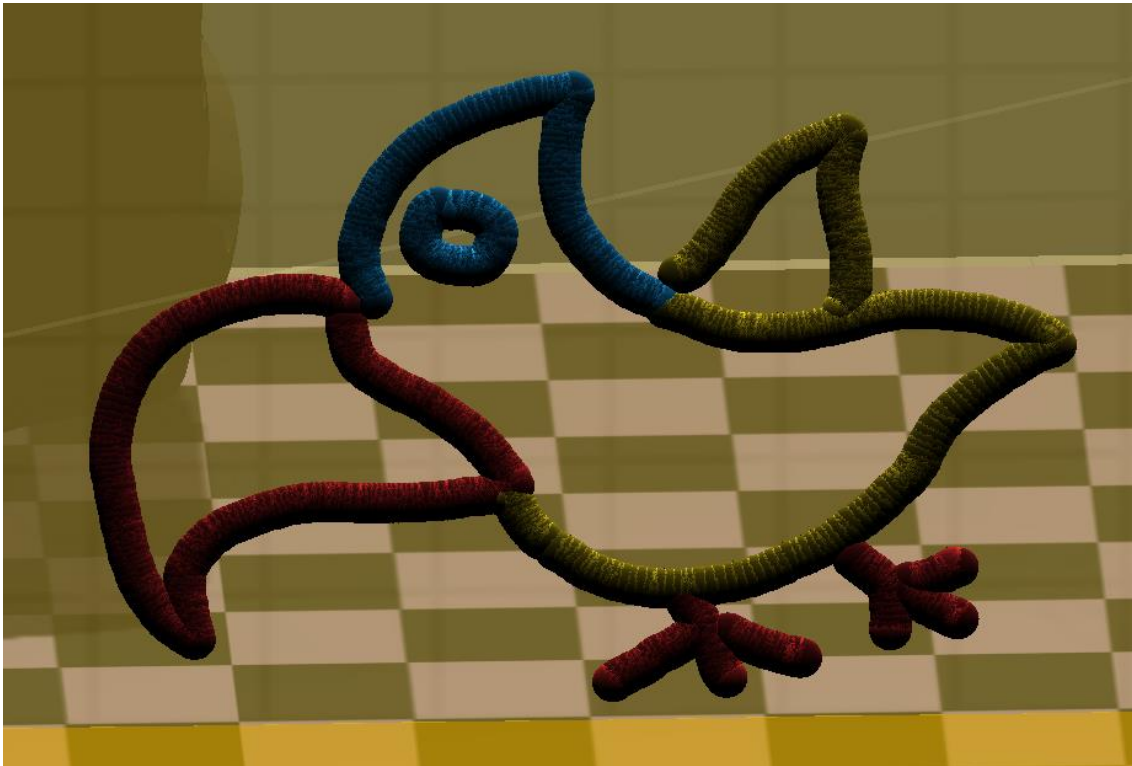

Figura 25. Dibujo realizado con varios colores.

5.8.2 Corte de objetos

Añadimos también una funcionalidad para cortar modelos por la mitad. Para ello debemos pulsar el botón de cortar mientras el lienzo intersecta con el modelo en cuestión. Este se cortará por el plano en el que se encuentra el lienzo. El corte de los modelos es algo ciertamente complejo.

Debemos dividir la malla que forma al modelo en dos nuevas mallas. De la malla original, los triángulos se dividirán en las dos nuevas mallas, dependiendo de a qué lado del plano se encontraban originalmente. El problema es que hay triángulos que tienen algunos vértices a un lado del plano y otros al otro lado. Estos necesitan un tratamiento especial.

```
public int[][] CorteTriangulo(Plane plano, out bool[] ladosSalida, out Vector3[] nuevosVertices)
{
    // Descubrimos a qué lado del plano está cada vértice
    List<int> lados = new List<int>();
    lados.Add(Convert.ToInt32(plano.GetSide(v1)));
    lados.Add(Convert.ToInt32(plano.GetSide(v2)));
    lados.Add(Convert.ToInt32(plano.GetSide(v3)));
    // Descubrimos el vértice que está en el lado diferente (y los otros dos estarán en el mismo)
    int diferente;
    List<int> iguales = new List<int>();
    if (Suma(lados) == 1)
    {
        diferente = lados.IndexOf(1);
    }
    else
    {
        diferente = lados.IndexOf(0);
    }
    iguales.Add(lados.IndexOf(1 - lados[diferente]));
    iguales.Add(lados.LastIndexOf(1 - lados[diferente]));

    // Los dos puntos de corte entre el plano y las aristas que tocan al vértice diferente, serán c
    Vector3 arista = (vertices[iguales[0]] - vertices[diferente]).normalized;   // Arista entre dos
    Ray rayo = new Ray(vertices[diferente], arista);                            // Rayo que coincid

    // Obtenemos la distancia hasta el punto de corte con un Raycast
    plano.Raycast(rayo, out float distancia);

    // Obtenemos el vértice sumando al punto original el vector por la distancia
    Vector3 nuevoVertice1 = vertices[diferente] + distancia * arista;

    // El proceso para el segundo vértice es el mismo
    arista = (vertices[iguales[1]] - vertices[diferente]).normalized;
    rayo = new Ray(vertices[diferente], arista);
    plano.Raycast(rayo, out distancia);
    Vector3 nuevoVertice2 = vertices[diferente] + distancia * arista;
```

Lo que hacemos es coger esos triángulos y dividirlos. Buscamos los puntos por los que el plano corta al triángulo, serán dos. Siempre habrá dos vértices a un lado del plano y el tercero al otro lado. Con los tres vértices, junto a los dos puntos de corte con el plano, generaremos un total de 3 subtriángulos. Como se muestra en la imagen.

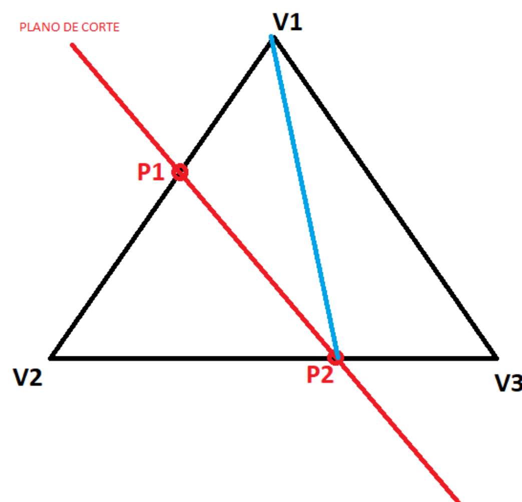

Figura 26. División de un triángulo al realizar el corte. El primer triángulo estaría formado por los vértices V2, P1 y P2. El segundo estaría formado por los vértices V1, P1 y P2. Y el tercero, estaría formado por los

vértices V1, V3 y P2. El primero de ellos formaría parte de una de las nuevas mallas, mientras que los otros dos formarían parte de la otra nueva malla.

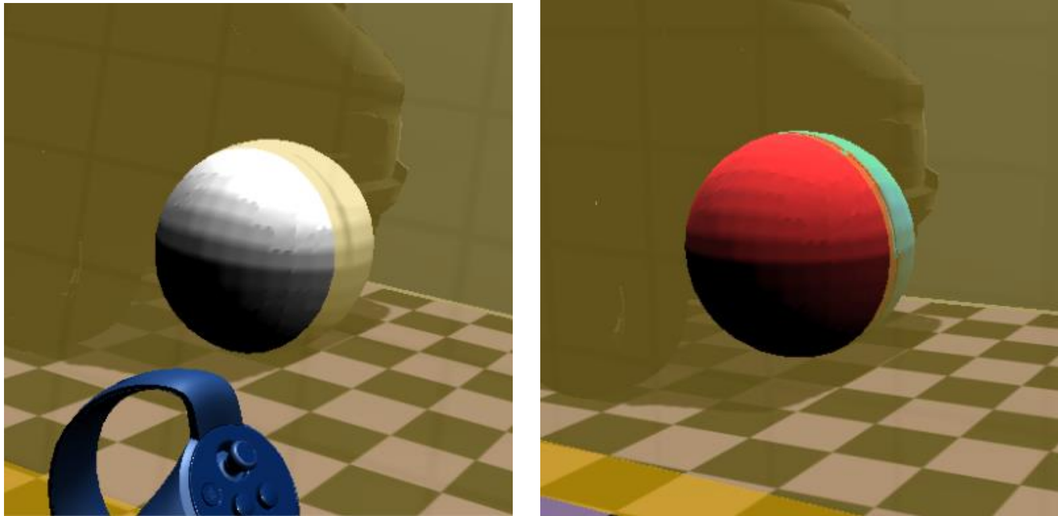

Figura 27. Modelo de una esfera siendo cortada. El corte se genera con dos materiales diferentes para diferenciar.

6. Resultados

En este apartado mostramos resultados que hemos obtenido con la aplicación. Hemos colocado diferentes modelos en la escena, los cuales deben tener siempre las características mencionadas anteriormente.

Figura 28. Ejemplo de diseño de un traje utilizando diferentes colores.

Figura 29. Otro diseño de traje.

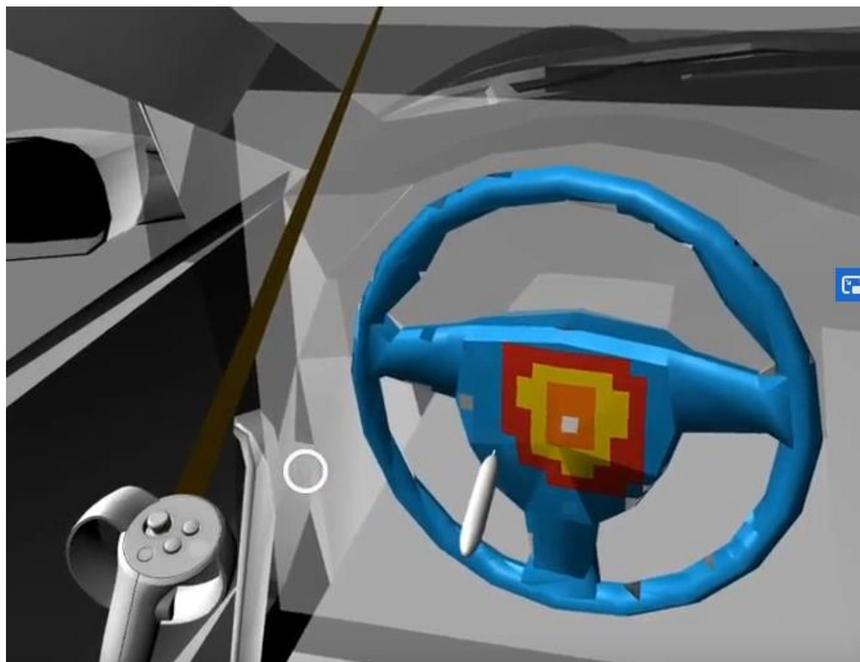

Figura 30. Diseño del volante de un coche.

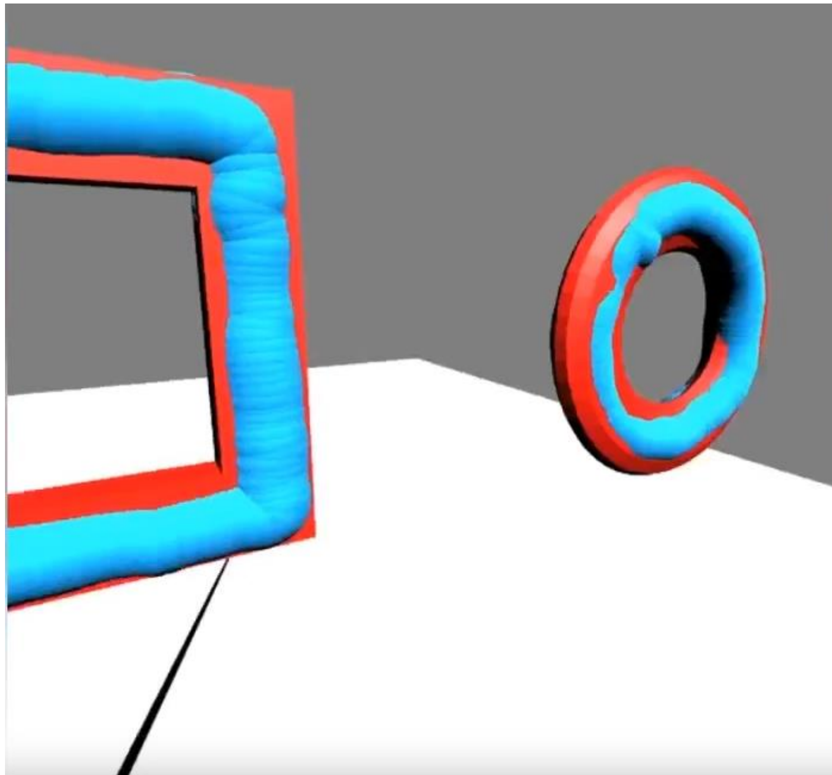

Figura 31. Usuario tratando de repasar con la mayor precisión posibles los modelos que encuentra en la escena.

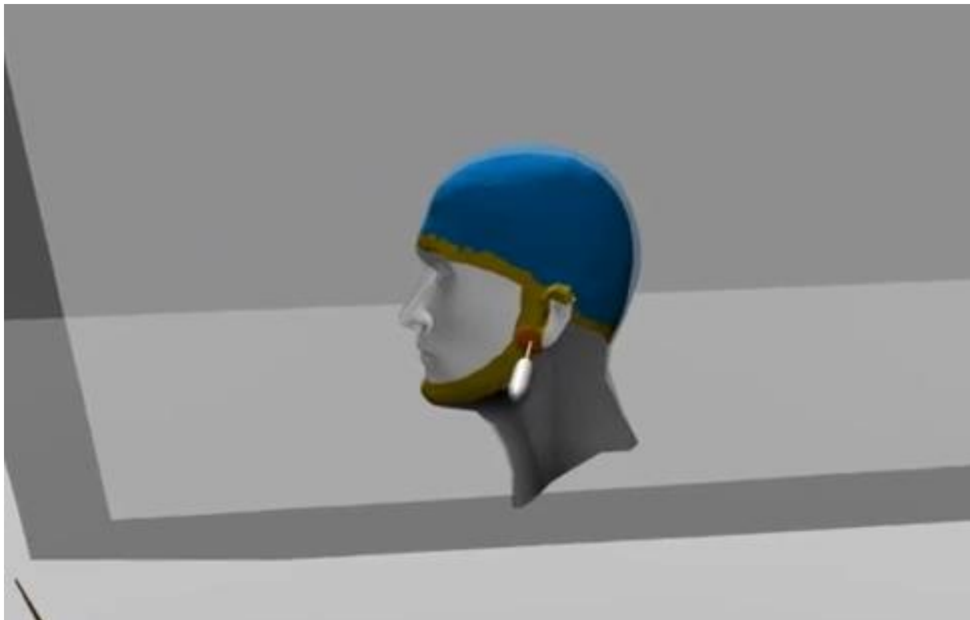

Figura 32. Diseño de un casco de bici.

7. Conclusiones

A partir de la aplicación que hemos desarrollado, es cierto que no hemos podido realizar suficientes pruebas como para sacar conclusiones lo bastante claras, pero es posible que el proyecto siga adelante.

De todos modos, podemos hablar, de forma especulativa, de las posibles ventajas y desventajas con las que contaría esta aplicación.

Teniendo en cuenta la naturaleza del proyecto, considero que su mayor valor de cara al público radica en la posibilidad que ofrece de ejercitar la mente y al mismo tiempo entretenernos de una manera moderna y novedosa. Realmente, se trata de una apuesta de cara al futuro, ya que podemos decir que, actualmente, la mayoría de las personas no están familiarizadas con la realidad virtual. Si visitas la vivienda de algún amigo, lo más probable es que no te encuentres con unas gafas VR. Pero lo mismo podríamos haber dicho hace no tantos años del teléfono móvil y ahora solo hay que abrir los ojos para ver lo que sucede, como ocurre con toda novedad tecnológica. Lo cierto es que cada vez es mayor el porcentaje de teléfonos o tabletas fabricados compatibles con la realidad virtual. Con la modernización de los medios de diseño y la cada vez más fuerte implantación de la tecnología en el mismo, no sería tan descabellado pensar en la utilización de este tipo de aplicaciones en un futuro no muy lejano.

Encontramos también algunos contras y es que no se trata de una aplicación que puedas utilizar durante un intervalo de tiempo muy prolongado. El motivo es que, al cabo de unos minutos, nuestra visión comienza a sentir cansancio y debemos descansar. El peligro es hacer caso omiso de esta sensación de cansancio y terminar dañando nuestro cuerpo. Se debería recomendar el uso moderado de este producto.

En cuanto a la aportación personal que me llevo por trabajar en este proyecto, puedo enumerar varios aspectos. En primer lugar, está claro que el aprendizaje ha sido de gran valor. He seguido formándome en el campo de la realidad virtual, algo que he descubierto, realmente me gusta. Finalmente diré que este confinamiento ha servido al menos para descubrir nuevas tecnologías que podrían quizá aplicarse a nuestro proyecto.

Bibliografía y referencias

Tomas Dorta, Gokce Kinayoglu, Michael Hoffmann. Hyve-3D: A New Embodied Interface for Immersive Collaborative 3D Sketching. SIGGRAPH '14 Studio: Special Interest Group on Computer Graphics and Interactive Techniques Conference (Vancouver, Canadá, 2014)

Hemant Bhaskar Surale, Aakar Gupta, Mark Hancock, Daniel Vogel. TabletInVR: Exploring the Design Space for Using a Multi-Touch Tablet in Virtual Reality. CHI Conference (Glasgow, Escocia, 2019).

Rahul Arora, Rubaiat Habib Kazi, Fraser Anderson, Tovi Grossman. Experimental Evaluation of Sketching on Surfaces in VR. ACM CHI Conference on Human Factors in Computing Systems (Denver, EEUU, 2017).

Rahul Arora, Rubaiat Habib, Tovi Grossman, George Fitzmaurice, and Karan Singh. SymbiosisSketch: Combining 2D and 3D Sketching for Designing Detailed 3D Objects in Situ. CHI Conference on Human Factors in Computing Systems (Montreal, Canadá, 2018).