Nuevas Implementaciones en Neotrie VR: MathItems

New Implementations in Neotrie VR: MathItems

Trabajo Fin de Grado

Autor:
Rubén Hernández Sánchez

Tutor:
José Luis Rodríguez Blancas

Grado en Matemáticas

Junio, 2023
Universidad de Almería

Índice general

Abstract in English

We have been working for over a year and a half as a programmer for the software of dynamic geometry in virtual reality Neotrie VR. During this time we have applied the basis of programming that we have learned in the Degree and perfected them.

In this End-of-Degree project we will show the implementation of one of the key features that we have been working on, the abstraction of the concept of *MathItem*.

First, we will present the development enviroment and the programs that we use. We talk about its graphic motor, *Unity*, and its work philosophy. Then we will do the same with the software itself, *Neotrie VR*.

Secondly, we will explain about the state at we first encountered the software; the struggles we encountered everytime we tried to implement any new object to the game due to the large amount of variables in play. Then we will demonstrate how the concept of *MathItem* can generalize all we already had in the game. Also, it gives the necessary base to let the game keep growing.

We will continue by talking about performance improvements thanks to the new philosophy of work and new developments tools provided by *Unity*. Then we will see how movements are applied to *MathItems* and the most important movements included in *Neotrie VR*.

To finish this project we will be testing the work done by creating some brand new *MathItems* using a new tool that never has been used before in the game, and we will discuss over possible future improvements.

Resumen en español

Durante nuestra colaboración de año y medio en el desarrollo del software de geometría dinámica en realidad virtual *Neotrie VR*, hemos aplicado las bases de programación adquiridas en el grado y las hemos perfeccionado.

En este Trabajo de Fin de Grado expondremos la implementación de una de las mejoras fundamentales en las que hemos trabajado, la abstracción del concepto de *MathItem*.

Comenzamos estableciendo el entorno de desarrollo y los programas que usamos. Hablamos de su motor gráfico, *Unity*. De su filosofía de trabajo. Continuamos haciendo lo propio con el software en cuestión, *Neotrie VR*.

Seguimos entrando en materia, planteando el estado en el que nos encontramos el software; de los problemas con los que nos topamos a la hora de implementar algún objeto nuevo debido a tener una gran cantidad de variables. Vemos cómo los *MathItem* generalizan todo lo que teníamos hasta ahora. Además, aportan la base necesaria para seguir expandiendo el juego.

Continuamos con mejoras en rendimiento gracias a la nueva filosofía de trabajo y a recientes herramientas de desarrollo otorgadas por *Unity*. Vemos cómo se aplican a los movimientos de los *MathItem* y los principales de estos implementados en *Neotrie VR*.

Para acabar probaremos que hemos solucionado el problema que nos encontramos. Crearemos un par de *MathItem* con apoyo de una herramienta nunca usada en el juego. Por último, reflexionamos sobre posibles mejoras futuras.

1 Introducción

Iniciamos este proyecto de colaboración con la empresa *Virtual Dor* a través del programa de inserción laboral, Ícaro. *Neotrie VR* es un software que lleva ya más de cinco años en desarrollo, iniciado por Diego Cangas, actual director de la empresa. El software se ha nutrido de colaboraciones con alumnos en finalización de estudios como nosotros. Esto ayuda a que el juego siga avanzando con ideas y aportaciones nuevas. Por otra parte, cada uno de estos programadores tiene su nivel particular y su forma personal de programar. Esto ha provocado que cada vez sea más difícil añadir nuevas aportaciones o interconectar las existentes.

Neotrie usa la plataforma de *Unity* como motor gráfico. Esta usa la programación orientada a objetos que aprendimos en la asignatura de Programación de Computadoras. Esto nos facilitará el trabajo de adaptación. Aunque aprendimos esta mecánica de trabajo en el lenguaje *Java* y *Unity* usa *C#*, veremos que ambos lenguajes comparten muchas similitudes.

Además, desde los inicios de *Neotrie* hasta ahora, *Unity* ha lanzada nuevas herramientas de desarrollo, dando alternativas más eficientes a las usadas en el software. Cuando empezamos, cada elemento funcionaba por su cuenta y no se comunicaba de ninguna forma con otros elementos. Esto hacía que el flujo de trabajo se atascara al introducir unos pocos objetos.

En este trabajo explicaremos cómo hemos solventado el problema de inconexión en elementos implementados. Trabajaremos con los elementos básicos que definen *Neotrie* y formaremos una base fácil de seguir y de expandir. Para ello, usaremos los conocimientos en programación adquiridos durante toda la carrera y dotaremos a los futuros desarrolladores de una mecánica de trabajo fija. En ella se dictarán unas directrices a seguir para implementar nuevos objetos al juego. Una vez se cumplan las directrices, se dejará libertad al programador para incorporar funcionalidades específicas al objeto implementado.

Para comenzar debemos de conocer las herramientas que manejamos. Para ello veremos en el Capítulo 2 el entorno de desarrollo que utilizaremos. Empezando por el motor que da vida a *Neotrie*, *Unity*. Veremos que es un motor gráfico de videojuegos que tiene como fin facilitar el acceso al desarrollo de videojuegos a pequeñas empresas. Por ello es muy usado en juegos del género *indie*. Aparte de esto, nos familiarizaremos con el software de *Neotrie VR*. Conoceremos sus orígenes y cómo trata de fomentar que las matemáticas, en especial la geometría, resulten una materia más atractiva, entretenida, fácil de visualizar y comprender.

En el Capítulo 3, seguiremos con la aportación central de este Trabajo de Fin de Grado, el concepto de *MathItem*. Veremos la estructura de objetos que seguía *Neotrie* y los problemas de expansión que representaba. La llegada de la clase *MathItem* no solo se traduce en facilitarnos el trabajo a nosotros los desarrolladores, si no que permite un mayor control en las interacciones de objetos. Esto nos otorga la capacidad de brindarle al usuario más opciones.

A continuación, en el Capítulo 4, veremos como la implementación de la clase *MathItem* afecta a los movimientos existentes. Para ello, primero explicaremos cómo

estaba organizado, luego veremos las diferentes opciones que aporta *Unity* a esta materia. Entre ellas se encuentran los *eventos*. Estos resultarán especialmente útiles para la implementación de la clase *MathItem*, en concreto, para controlar las actualizaciones de movimientos. Veremos que esto supone una mejora en la eficiencia del software. Usaremos el conocimiento adquirido para mejorar los distintos modos de movimiento implementados en *Neotrie*.

Acabaremos este proyecto poniendo a prueba todo lo explicado con anterioridad. Así, en el Capítulo 5, nos centraremos en desarrollar dos nuevos tipos de *MathItems*: arcos de circunferencia (y sectores circulares) y cónicas. Además, usaremos una herramienta nueva para *Neotrie* que le brindará muchas posibilidades en el futuro.

Concluimos en el Capítulo 6 con unas reflexiones y posibles mejoras futuras. Ya que hemos comprobado que el problema de escalabilidad está resuelto, podemos comenzar a trabajar en el siguiente gran interés del software: la importación de archivos de construcciones matemáticas realizadas en otros softwares de geometría dinámica, como *GeoGebra*. Comprobaremos así que la capacidad de abstracción de conceptos de un matemático son útiles y más que bienvenidos en un sector tan en auge como lo es el de la informática.

2 Entorno de desarrollo

Iniciarse en un nuevo trabajo siempre tiene sus dificultades y requiere dedicar cierto tiempo de adaptación y aprendizaje. Veremos cómo los conocimientos aprendidos durante el grado tienen su aplicación en el mundo laboral, en este caso, en el sector informático donde colaboraremos como programadores en el software de geometría dinámica *Neotrie VR* [3].

2.1 Motor gráfico. Unity

El proyecto de *Neotrie* está basado en *Unity*, un motor de videojuegos diseñado por la empresa *Unity Technologies* [6] en el año 2005. El principal objetivo de este producto es acercar el mundo de la programación y diseño, tanto de aplicaciones como de videojuegos, a las personas poco o nada especializadas en el sector. Ofrece soporte para la mayoría de las plataformas principales del mercado, Android, Windows, iOS, hasta diseño Web. Además, cuenta con dos versiones, una personal gratuita, y otra de pago para los empresarios.

Figura 2.1: Logo de Unity.

La plataforma de desarrollo *Unity* brinda a sus usuarios toda clase de herramientas para facilitar la entrada al mundo de la programación, desde videotutoriales guiados para aprender los conceptos más básicos, que sirven además de material visual a modo de clases [7], hasta una extensa documentación de todas sus clases, métodos y variables, para dejar claro el uso de cada elemento. Finalmente, la herramienta con la que trabajaremos es su editor.

Editor de Unity

El modelo de programación por el que optó *Unity* es la programación basada en componentes. Para el desarrollo del software solo tendremos que crear diferentes *GameObjects*, que son objetos vacíos dentro de la *escena*. Posteriormente, asignaremos a esos objetos diferentes piezas llamadas *componentes*, las cuales les otorgan toda clase de comportamientos y propiedades para obtener lo que deseemos. Una vez creado un *GameObject* complejo, con los componentes que queremos para cierto comportamiento, *Unity* nos da la opción de guardar este objeto complejo en forma de *prefab*. Esto nos permite instanciar copias de ese objeto sin tener que repetir todo el trabajo realizado.

Este proceso de creación se llevará a cabo en su editor. En la Figura 2.2 se muestran todos los conceptos elementales señalados en el párrafo anterior. Este editor está formado por un grupo de ventanas, cada una con un propósito marcado y una finalidad específica. En la imagen podemos ver las más relevantes, en la parte de abajo nos encontramos la ventana de proyecto, donde encontraremos todos los archivos generados o utilizados por nuestro software. Arriba a la izquierda tenemos la jerarquía que muestra una lista, como podemos ver en primer lugar nos encontramos la escena, dentro de ella podemos visualizar todos los *GameObjects*. Esta ventana nos será de gran utilidad cuando veamos el concepto *padre-hijo*. En el centro podemos ver la escena, donde todos los cambios que hagamos se mostrarán en tiempo real. Por último, a la derecha, nos encontramos el inspector. Cuando seleccionemos un *GameObject* o algún objeto en particular, esta ventana nos mostrará toda la información sobre él, ya sea sus componentes o diversas opciones relacionadas.

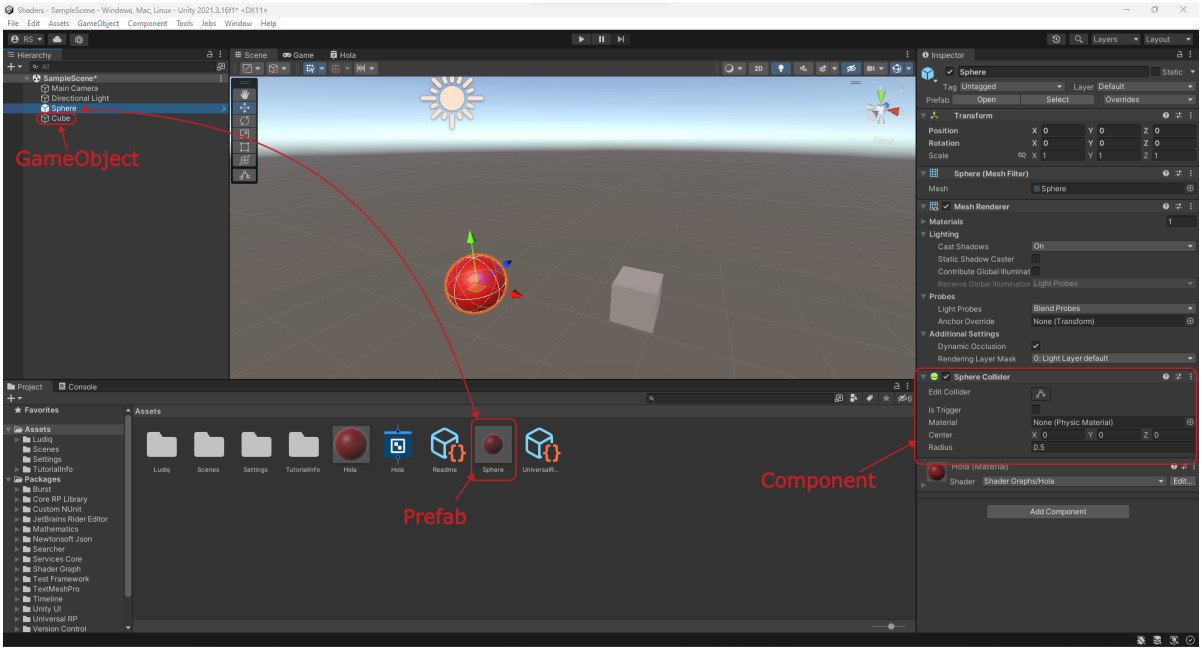

Figura 2.2: Editor de Unity y sus conceptos elementales.

Lenguaje de programación. C#

Unity cuenta con una batería de diversos componentes, los cuales aportan funcionalidades pensadas para usos genéricos en cualquier desarrollo. Lo interesante de este motor, es que nos da la posibilidad de programar nuestros propios componentes. Es decir, para conseguir el comportamiento que queramos en cualquier objeto, solo necesitamos programarlo apoyándonos en los demás componentes que ya nos proporciona *Unity*. Estos componentes propios los llamaremos *scripts*.

Aunque *Unity* internamente use como lenguaje *C++*, los componentes propios se desarrollaran en *C#*, un lenguaje de programación de alto nivel con una curva de aprendizaje más sencilla que el primero. Además, podemos poner en práctica nuestros conocimientos de programación en *Java*, adquiridos en la asignatura de Programación

Figura 2.3: Editor de Unity y sus conceptos elementales.

de Computadores. *Java* y *C#* comparten muchas similitudes, pues ambos provienen de *C++*. Aun así, *C#* es preferido por muchos dado que es más moderno y adoptó muchas características de Java que se consideraron avances en su tiempo.

2.2 Neotrie VR

Las matemáticas, a pesar de que en estos últimos años han ido ganando cada vez más adeptos, siguen siendo una materia compleja, lo que la hace ser rechazada. Una de las ramas en las que la didáctica tiene mucho margen de mejora es la geometría. Esta rama favorece a aquellas personas que tienen un gran sentido de visión espacial, dejando a las demás que no poseen esta habilidad en clara desventaja.

A menudo ciertos resultados tienen demostraciones gráficas sencillas y fáciles de comprender, pero se recurre a sus versiones analíticas por su rigurosidad y estandarización. Esto hace que los alumnos que no consigan entenderlo pierdan interés por la materia. A pesar de sus ventajas, las demostraciones gráficas apenas son usadas en los procesos de enseñanza por ser tediosas de realizar en un aula estándar de enseñanza.

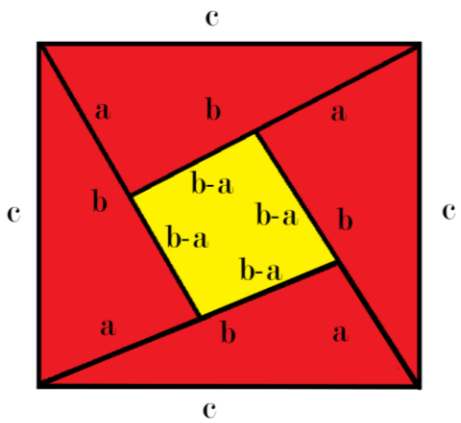

Figura 2.4: Demostración gráfica del Teorema de Pitágoras por Bhaskara Acharia.

La incorporación de la tecnología en las aulas está cada vez más extendida y su uso como herramienta para el aprendizaje no tiene igual. En la rama de la geometría, conocidos software como *GeoGebra* ayudan diariamente a facilitar la tarea de explicar a profesores de todo el mundo. Sin embargo, el problema de la visión espacial no se ve del todo solventado con la inclusión de dibujos realizados a ordenador o a mano, ya que nosotros vivimos en un mundo de tres dimensiones, mientras que los dibujos en uno de dos, ya sea una pantalla de ordenador o una pizarra, respectivamente.

El siguiente paso para la tecnología era dar ese salto a la tercera dimensión. La realidad virtual es la tecnología que lo consigue. Estos dispositivos, como las populares Meta Quest 2, consiguen trasladar nuestra visión y movimientos a un entorno virtual donde podremos vivir toda clase de experiencias. Aprovechando esta nueva herramienta nace *Neotrie VR*, se trata de un software de geometría en realidad virtual, desarrollado por la empresa *Virtual Dor* spin-off de la Universidad de Almería.

Figura 2.5: Logos de *Neotrie VR* y Virtual Dor.

En *Neotrie* tenemos la posibilidad de experimentar la geometría de manera intuitiva y dinámica. Algunas de sus características son el modelado de objetos en tres dimensiones, realización de construcciones geométricas con su sistema de herramientas y su recientemente añadido modo multijugador (véase artículo de prensa [4]).

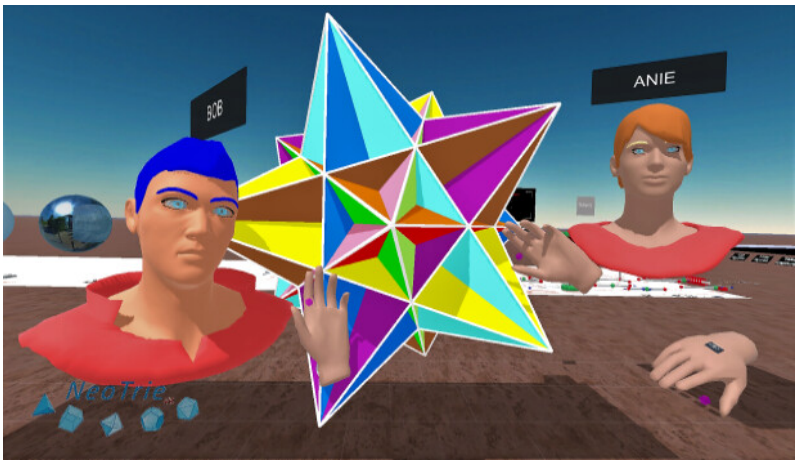

Figura 2.6: Modo Multijugador

3 Creación del concepto MathItem

La manera en la que *Neotrie VR* nos brinda la posibilidad de realizar construcciones geométricas es mediante una serie de objetos prefabricados. Cada objeto representará un elemento matemático bien definido. Por ejemplo: vértices, aristas, caras, etc.

Antes de nuestro trabajo, cada elemento tenía su clase específica, es decir, cada uno de ellos tenía sus propios métodos, que no interactuaban entre sí (salvo dependencias). Esto dificultaba en gran medida la tarea de incorporar nuevos elementos más complejos.

3.1 Vértices

Cuando empezamos el proyecto, nos encontramos con una serie de clases con muchas similitudes, necesitábamos algún tipo de orden o de elemento básico que usar, para conseguir que todo lo demás dependiera de ello. La respuesta la encontramos en los vértices, pues son los elementos más básicos que se pueden crear en el juego. Ahora tenemos que cambiar la mentalidad a la hora de programar. En vez de que cada elemento matemático fuera representado por un objeto que trabaja independientemente, tenemos que encontrar una manera de representarlo a través de vértices. La forma de representarlo no tiene por qué ser única, en cuyo caso podemos ofrecer al usuario distintos métodos de crear el mismo elemento.

3.2 Clase abstracta: MathItem

Ahora ya tenemos una idea clara de cómo queremos que los elementos geométricos funcionen. Todos tienen que depender de vértices. El siguiente paso es encontrar la forma de programarlo sin perder la flexibilidad que nos ofrece tener cada elemento independiente. Veamos cual es la solución.

La clase *MathItem* nace con la intención de abstraer todos los métodos y funciones comunes a todos los elementos matemáticos implementados. En programación existe el concepto de clase abstracta. Esta es una clase "madre" que dota a sus "hijos" de todos los métodos y propiedades públicos y protegidos definidos en ella. En *Java*, cuando queríamos heredar una clase, usábamos la palabra reservada *extends*, en *C#* basta con poner dos puntos *:*.

Definiendo la clase como abstracta, *C#* nos da acceso a dos nuevas palabras clave, *virtual* y *abstract*. Ambas son modificadores de métodos, la primera hace que el método definido en la clase "madre" pueda ser sobrescrito, por tanto, definido de otra forma, en las clases "hijo" que lo necesiten. Por otra parte, la palabra *abstract* obliga a los "hijos" a implementar un método con la misma cabecera. En este caso, la clase "madre" no implementará el método. Para sobrescribir los métodos se usará la palabra clave *override* en la cabecera del método deseado dentro de la clase "hijo". Sin olvidarnos de que si no usamos ninguna de estas nuevas palabras clave, los métodos de la clase "madre" se

heredan directamente a los hijos, tenemos así, la flexibilidad que necesitábamos para generalizar cualquier elemento matemático a nuestro alcance.

```
1 public class Madre
2 {
3     public abstract void MetodoAbstracto(); //Sin implementacion
4     public virtual void MetodoVirtual(int a)
5     {
6         //Implementacion para la mayoria de los hijos
7     }
8     public void MetodoNormal()
9     {
10        //Implementacion del metodo
11    }
12 }
13 public class Hijo1 : Madre
14 {
15    public override void MetodoAbstracto()
16    {
17        //Implementacion obligada por clase madre
18    }
19    //Este hijo no necesita modificar el MetodoVirtual, por lo que no
         hace falta modificarlo
20    //MetodoNormal sigue siendo accesible por herencia de Madre
21 }
22 public class Hijo2 : Madre
23 {
24    public override void MetodoAbstracto()
25    {
26        //Implementacion obligada por clase madre
27    }
28    public override void MetodoVirtual(int a)
29    {
30        base.MetodoVirtual(a); //Si se necesitara se puede ejecutar la
             implementacion de Madre con la palabra clave base
31        //Implementacion unica de Hijo2
32    }
33    //MetodoNormal sigue siendo accesible por herencia de Madre
34 }
```

Código 3.1: Clase abstracta, métodos *virtual* y *abstract*.

Implementación

En esta sección iremos implementando la clase a la vez que lo razonamos. Hemos concluido que debe ser una clase abstracta por lo que la cabecera de la clase nos queda,

```
1 public abstract class MathItem : MonoBehaviour, IDefRemovable,
     IRestaurable, ISelectable, IEquatable<MathItem>
```

Nótese que la clase implementa una serie de interfaces. La clase *MonoBehaviour* es una clase interna de *Unity* utilizada por los *GameObjects*.

Por la necesidad de tener que llevar un registro de todos los *MathItems* en escena, crearemos un diccionario. Este será estático, es decir, será independiente al objeto creado. Como clave tendremos una variable de tipo *string* que indica el tipo de *MathItems* que queremos. Asociado a ella podremos acceder a una lista de *MathItems*, si necesitamos que sea específicamente una lista del tipo deseado podremos hacer un *cast* (ver método Cast de la Figura 3.2), ya que nos aseguraremos de que esa lista contiene exclusivamente elementos de un único tipo.

Con la misma dinámica guardaremos un diccionario estático que nos irá proporcionando un entero simbolizando la ID, del tipo proporcionado en la clave. Además, guardaremos dos listas estáticas, una con los tipos de *MathItems* disponibles y otra con aquellos que puedan ser transparentes.

```
1 public static Dictionary<string, List<MathItem>> MathItems= new();
2 public static Dictionary<string, int> UniqueIDStatic = new();
3 public static List<string> MathItemsTypes = new() { "Vertice", "Edge", "
    FaceScript", "Ellipse", "Disk", "Cylinder", "Cone", "Sphere", "
    Quadric", "Conic", "Circle"};
4 public static List<string> CanBeTransparent = new() { "FaceScript", "
    Cylinder", "Sphere", "Quadric" };
```

Ahora trataremos las propiedades de cada *MathItem*. Cada objeto debe guardar su ID, un entero con su "caso" por si fuera necesario, su color, un entero por si es seleccionado, una lista con los vértices de los que depende, un booleano que controla si está actualizándose y un entero con en número de vértices que están pidiendo que se actualice este *MathItem*.

```
1 public int ID;
2 public int caso = 0;
3 public Color preferentColor;
4 public int selectionGroup = 0;
5 public List<Vertice> Vertices;
6 public bool isUpdating = false;
7 public int vertsUpdating = 0;
```

Siguiendo la base establecida de que todos los *MathItem* dependen de vértices, podemos ir abstrayendo e implementando todos los métodos comunes anteriormente implementados en cada objeto por separado. El diagrama de clase lo podemos ver en la Figura 3.1, los métodos en cursiva indican que tienen la palabra *abstract* en su cabecera.

Con este trabajo realizado, seguir añadiendo *MathItem* resulta una tarea sencilla, ya que toda la base ya se encuentra implementada y los métodos abstractos nos indican los métodos necesarios para su correcto funcionamiento. Solo quedaría preocuparse por el comportamiento específico de ese *MathItem*.

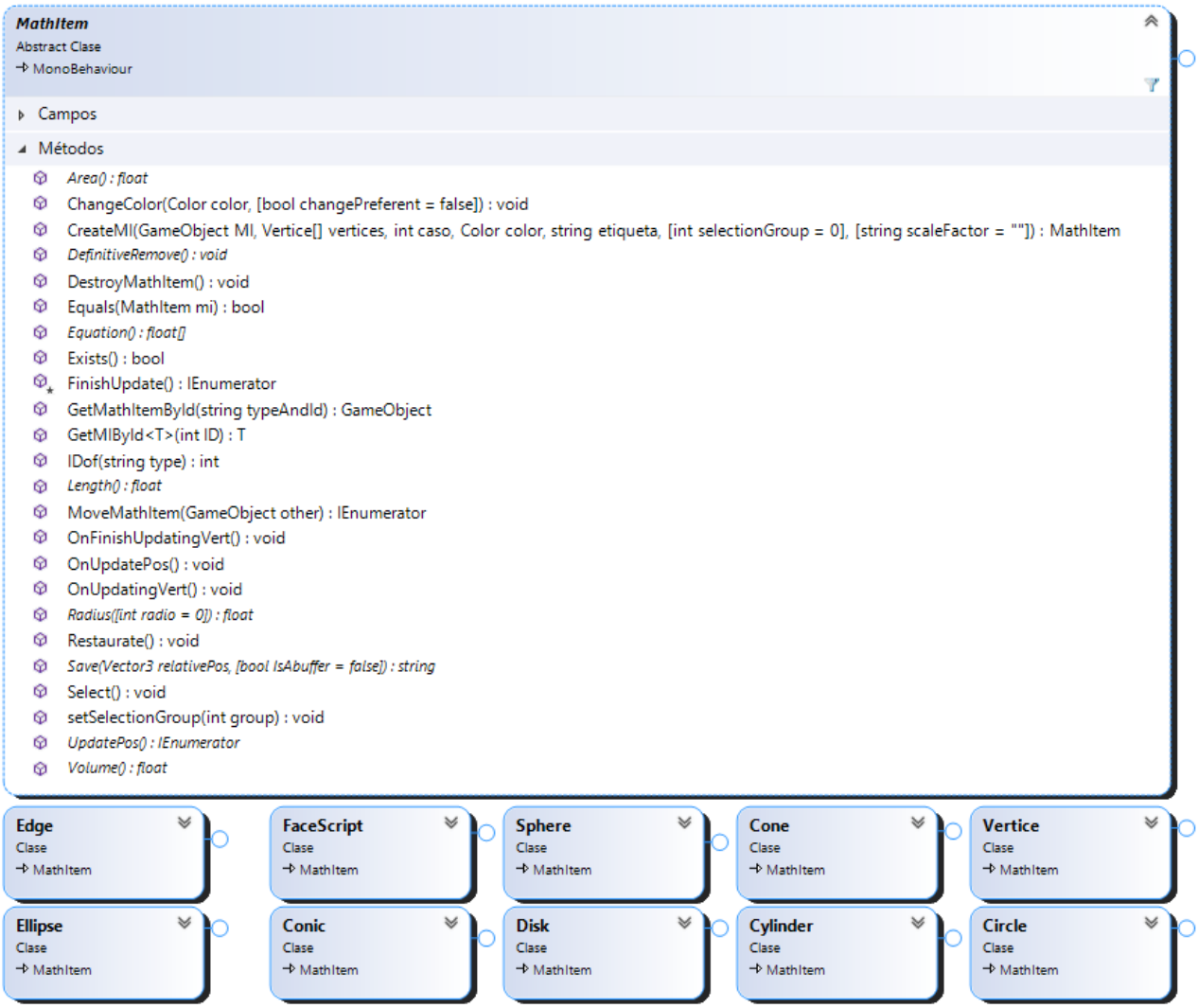

Figura 3.1: Clase MathItem

Por motivos de limpieza de código, se ha programado una clase estática aparte que aporta funcionalidades adicionales al manejo de estructuras de datos relacionadas con *MathItem*. Su diagrama de clase lo podemos ver en la Figura 3.2.

La ventaja de separar estos métodos en una clase estática aparte es que evita que cada instancia del objeto tenga que cargar con una clase más pesada. Una clase estática no está asociada a ningún objeto, es decir, instancia, se trata de un conjunto cerrado de instrucciones. Entre sus principales desventajas está el solo poder usar métodos estáticos, es decir, a menos que tengas una referencia estática al objeto que necesites, no podrás acceder a sus variables. Además, todos los datos que necesites deberás pasarlos como parámetros.

Para solucionar el problema de la referencia de objeto, en *C#* podemos usar el modificador de parámetro *this* para indicar el tipo de objeto que podrá acceder a este método. Así, aunque la clase y el método sean estáticos, se accederá a ellos a través de los objetos del tipo correspondiente en lugar de llamando a la clase, luego al método y pasando el objeto como parámetro. Esta es una técnica muy usada por los programa-

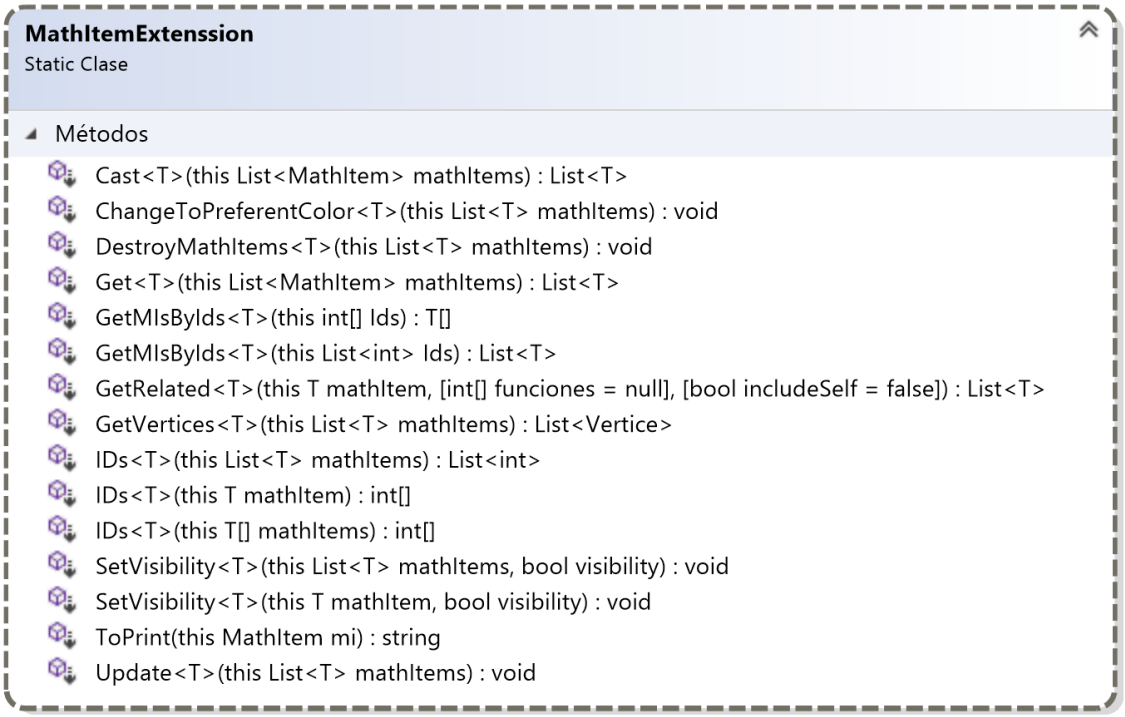

Figura 3.2: Clase MathItemExtenssion.

dores para dotar de funcionalidades extra a cierto tipo de objetos.

Por último, podemos observar en la Figura 3.2, el uso de genéricos. En programación, un genérico es una variable de tipo variable. Con un ejemplo se entiende mejor, imaginemos que queremos un método que, al pasarle dos objetos, nos los compare. Podríamos hacer el mismo método para todos los tipos de variables, *int*, *float*, *bool*... mejor no. Si usamos genéricos, podemos hacerlo con un único método. Normalmente, definimos los tipos de los parámetros. Con genéricos, el tipo del parámetro se define cuando se llame al método.

Por defecto, un genérico hereda de objeto, por lo que no podríamos acceder a sus métodos de objeto. Para solucionarlo usaremos la palabra clave *where*, la cual nos permite restringir los tipos que adopta ese genérico. Recopilando toda esta información, ya podríamos hacer el método mencionado en el ejemplo.

```
1 //<T> indica que T es un generico
2 public int Compare<T>(T object1, T object2) where T : IComparer
3 {
4     //En una misma llamada T sera de un unico tipo especifico
5     //T sera de un tipo que implemente ICompare
6     return object1.Compare(object2);
7 }
```

Código 3.2: Ejemplo genérico.

Esta es una técnica avanzada de programación que se escapa a los contenidos de la asignatura de Programación de Computadoras. Además, es de las más útiles en la generalización.

Fuera Updates, bienvenidos eventos

La clase interna de *Unity*, *MonoBehaviour*, aporta un flujo de trabajo a los *GameObjects* al que tendremos que adaptarnos para que adquiera el comportamiento deseado; podemos ver un esquema del flujo en [10]. Entre ellos se encuentra el método *Update*. Se trata de un simple método que es llamado una vez por frame.

```
1 void Update()
2 {
3     //Codigo llamado cada frame
4 }
```

Código 3.3: Método Update.

Como podemos observar, el flujo de trabajo consiste en un bucle, es decir, para avanzar a la siguiente etapa debe finalizar los procesos actuales. Esto quiere decir que, si ponemos una tarea demasiado extensa en un *Update*, la tasa de frames se verá gravemente afectada, llegando al punto de detener el juego. Esto se debe a que *Unity* está hecho para ejecutarse en un único núcleo, por lo que perdemos mucha potencia a nuestros equipos actuales. Por desgracia, *Unity* todavía no nos aporta a los desarrolladores demasiadas herramientas de paralelización del trabajo, pero están trabajando en ello como podemos ver en [9].

Aun así, *Unity* se ha encargado de optimizar el método *Update*, es decir, si la tarea es liviana, todavía podemos optar por usar este método sin mayor preocupación. En nuestro caso, la escena enseguida se llena con numerosos *MathItems*, cada uno con su *Update* para asegurarse que está en la posición que le corresponde y de si existe. Esto conlleva un costo computacional demasiado elevado, pero en los inicios de *Neotrie VR* no había otra alternativa.

Esto cambió con la llegada de los *eventos*. Un evento es un objeto que guarda acciones, es decir, métodos. Estos métodos se ejecutarán una vez que ocurra el evento, esto es, se invoque. Si mezclamos esta utilidad con nuestra estructura centrada en vértices, podemos deducir que un *MathItem* debe actualizarse cuando uno de sus vértices esté siendo actualizado. Solo resta un problema, necesitamos una estructura que no interrumpa el hilo de ejecución y permita al juego seguir otorgándonos frame a pesar de volver a estar haciendo las tareas que antes teníamos en el *Update*.

Para solucionar este último problema, *Unity* implementa las corrutinas, siendo su primera aproximación a la paralelización que hemos mencionado antes. Una corrutina es un método que sigue el tiempo de ejecución normal, hasta que llega a una instrucción *yield*. Esta detiene la ejecución del método hasta que se cumpla lo establecido en la instrucción. Existen variedad de instrucciones, la mayoría se pueden ver en el diagrama de flujo de trabajo [10]. Al llegar a una instrucción *yield*, *Unity* nos puede seguir aportando frames sin obstruir el hilo principal y comportarse como un falso método *Update*.

```
1 //Una corrutina se crea devolviendo IEnumerator en un metodo
2 IEnumerator FalsoUpdate()
3 {
```

```
4     while(true)
5     {
6         //Codigo llamado cada frame
7         yield return new WaitForEndOfFrame();
8     }
9 }
```

Código 3.4: Falso Update.

Ahora estamos preparados para implementar este nuevo sistema. Los responsables de llevar a cabo actualizaciones en los *MathItem* son los vértices, por lo que crearemos un par de *eventos* como parte de sus propiedades. El primer evento marcará el inicio de la actualización, el segundo su final. Cabe destacar que estos *eventos* no pueden ser estáticos, ya que cada vértice necesita tener el propio para evitar que los *MathItems* no relacionados con el vértice en cuestión no se actualicen al hacerlo este.

```
1     public class Vertice : MathItem, IDefRemovable, IRestaurable,
          ISerializable, IComparer<Vertice>, IComparable<Vertice>
2     {
3         public UnityEvent UpdateDependantMI = new();//Inicio
              actualizacion
4         public UnityEvent FinishUpdateMI = new();//Fin actualizacion
5     ]
```

Código 3.5: Eventos en la clase Vertice.

Los *MathItems* se pueden suscribir a estos *eventos* ya sea a la hora de su creación o realizando operaciones como la del pegado. Para evitar problemas de ejecutar múltiples veces la corrutina de actualización, usaremos el campo *vertsUpdating* de los *MathItem* y un método previo a la corrutina. Además, con otro método y el campo *isUpdating* controlaremos cuando detenerla.

```
1 public abstract class MathItem : MonoBehaviour, IDefRemovable,
      IRestaurable, ISelectable, IEquatable<MathItem>
2 {
3     public void OnUpdatingVert() //Suscrito a UpdateDependantMI
4     {
5         vertsUpdating++;
6         if (vertsUpdating > 1) return; //Control de llamada multiple
7         isUpdating = true;
8         OnUpdatePos();
9     }
10    public void OnUpdatePos() { StartCoroutine(UpdatePos()); }
11    public abstract IEnumerator UpdatePos(); //Corrutina de actualizacion
          , necesita ser abstracta ya que cada MathItem tiene su propio
          proceso para colocarse
12    public void OnFinishUpdatingVert() //Suscrito a FinishUpdateMI
13    {
14        if(Vertices.Count==1 && !Vertices[0].GetComponent<
              Herramienta_GeneradoraVertices>())
15        {
16            isUpdating = false;
```

```
17                vertsUpdating = 0;
18                return;
19            }
20            StartCoroutine(FinishUpdate());
21        }
22        protected IEnumerator FinishUpdate()
23        {
24            yield return new WaitForEndOfFrame();
25            foreach (Vertice v in Vertices)
26                yield return new WaitUntil(() => !v.isUpdating); //Espera a
                      que todos los Vertices de los que depende este MathItem
                      dejen de actualizarse
27            isUpdating = false;
28            vertsUpdating = 0;
29        }
30    }
```

Código 3.6: Nuevo sistema de actualización por eventos.

Con esto habríamos terminado de implementar el nuevo sistema de actualización de *MathItems*. La principal ventaja de este sistema es que, en escenas con muchos *MathItems*, si no se requiere que se actualicen, es decir, realicen un movimiento, el consumo de sus *scripts* es prácticamente nulo. Otra ventaja es su escalabilidad. Al estar los métodos centralizados, no es necesario saber el tipo específico de *MathItem* para ordenar su actualización. Esto facilita la implementación de nuevos *MathItems*, ya que prácticamente no hay que preocuparse de su movimiento, ni de las demás funcionalidades comunes abstraídas en la clase *MathItem*.

4 Movimientos

El mayor orgullo de *Neotrie VR* es su capacidad de realizar geometría dinámica. Para ello, es necesario dotar a nuestros *MathItems* de movimiento. En este capítulo trataremos las distintas formas de trabajo con movimientos incorporadas en *Unity* y los distintos modos de movimiento disponibles en el juego.

4.1 Movimientos en Unity, Componente Transform

Ya hemos visto que *Unity* trabaja con escenas (véase Figura 2.2) . Dentro de ellas podremos crear objetos que directamente implementarán la clase MonoBehaviour, convirtiéndose así en *GameObjects*. A esta clase también le acompaña un componente, el *Transform*. Este guarda todo lo relativo a la situación espacial del objeto. Las más relevantes son un vector con su posición, un cuaternión con su posición (aunque en el editor veremos su conversión a un vector con los ángulos en cada eje), un vector con su escala y una referencia al *transform* del objeto padre.

Antes de continuar, haremos un inciso para saber más sobre el concepto de objetos *padre-hijo* y por qué es tan importante a la hora de analizar el movimiento y diseñar la jerarquía de la escena. Si nos fijamos arriba a la izquierda en la Figura 2.2, podemos observar una pestaña que nos muestra la jerarquía. En ella siempre tendremos la escena como un cajón que encierra a todos sus *GameObjects*. Pero no todos los objetos dentro de ella tienen por qué estar al mismo nivel. Un *GameObject* se puede volver un "cajón" para otros objetos. Con esto, ese *GameObject* se volvería un *padre* y todos los que estén dentro de él, sus *hijos*.

Aparte de para mantener la jerarquía más ordenada, los objetos padres tienen la propiedad de transmitir el movimiento a sus hijos. Además, los hijos pueden trabajar con dos tipos de coordenadas, locales (respecto a su padre) o globales. Haciendo uso de esta propiedad nace el concepto de *Figura*. Una agrupación de *MathItems* dependientes entre sí, donde aplicar ciertas transformaciones se simplifica al solo tener que aplicárselas al objeto padre, es decir, la figura.

Retomando el componente *Transform*, cualquier movimiento que queramos se reduce a calcular su posición, rotación y/o escala e insertarlo en las variables del componente correspondiente. Es importante optimizar este proceso, ya que estos cálculos deben realizarse en tiempo real para darle al usuario una sensación de continuidad, a pesar de ser cálculos discretos.

4.2 Movimientos en Neotrie VR

Entre las diversas funcionalidades implementadas en *Neotrie VR*, sin duda la más atractiva es poder diseñar y controlar tus construcciones con tus propias manos. Para ello se le otorga al usuario un menú por el que podrá acceder a distintos modos, ver Figura 4.1. Así cada mano tendrá funcionalidades muy diversas, dependiendo del modo en el que esté. Para seleccionarlo solo tendremos que apretar un botón del mando

para abrir el menú y acercar la mano al símbolo correspondiente, veremos como en el reverso de nuestra mano el modo cambia al que hayamos elegido.

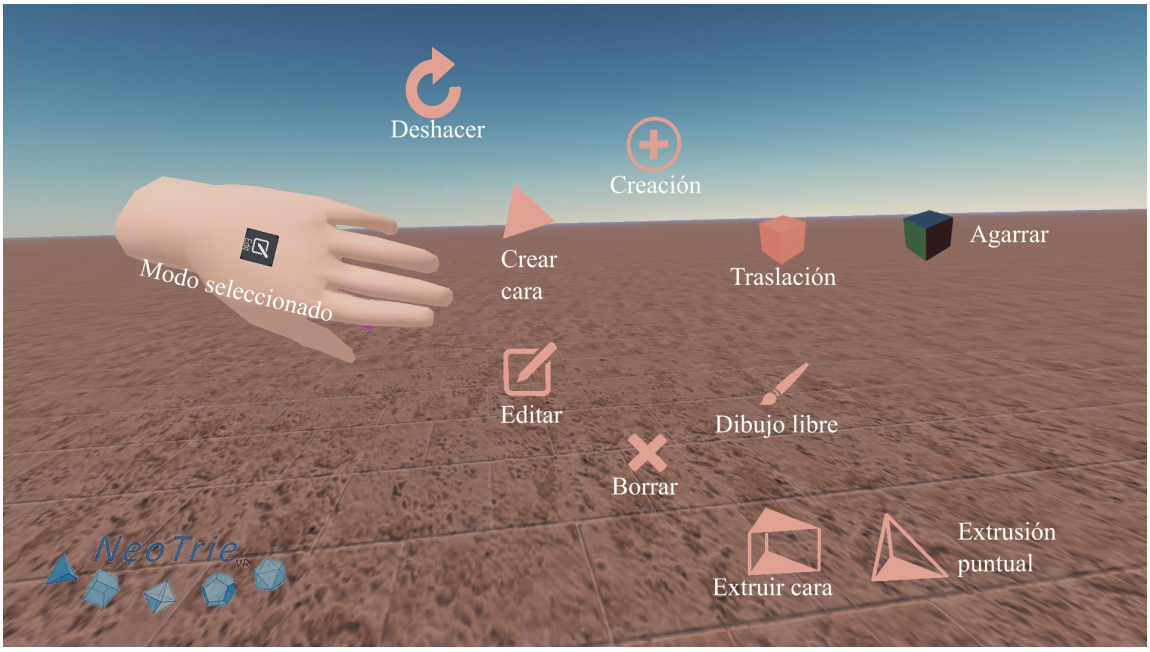

Figura 4.1: Menú de modos de la mano.

En esta sección trataremos los modos relacionados con el movimiento de *MathItems*. En concreto los de edición, translación y agarre.

Modo Edición

La geometría, como una de las ramas de las matemáticas, requiere de cierta habilidad para ser capaz de visualizar sus conceptos, lo que ayuda a comprenderla. *Neotrie VR* facilita esta tarea manteniendo una perspectiva modular sobre sus objetos matemáticos, cada uno de los cuales cumplen una función y combinando diversos elementos podemos construir formas más complejas.

Esta forma de trabajo también nos aporta una ventaja: ser capaz de editar cada componente por separado. Hasta ahora, esto se había conseguido dependiendo del tipo de *MathItem* que estuvieras editando. Cada uno tenía su método de movimiento. Con la idea de centralización en vértices hemos optimizado este proceso. Todos los *MathItems* se mueven bajo los mismos criterios. Veamos como lo hemos conseguido. En primer lugar, veamos cómo se selecciona un *MathItem*, en la clase que gestiona nuestra mano, *Neotrie_Controller*, tenemos un método que se ejecuta cuando apretamos el gatillo del mando, *UseControllerMode*. Este elegirá la acción a realizar dependiendo del modo de la mano.

```
1 public class Neotrie_Controller : MonoBehaviour
2 {
3     //Suscrito a Evento "pulsar gatillo"
4     public void UseControllerMode()
```

```
5     {
6         switch (useMode.MenuNum)
7         {
8             case 2: //Modo edicion
9                 StartCoroutine(MoveMathItem());
10                break;
11        }
12    }
13 }
```

Código 4.1: Modo Edición.

Este comenzará una corrutina donde podremos controlar cuando queremos vaciar las variables o por si queremos ejecutar otro método en un orden específico. Veremos que la corrutina es muy sencilla. Primero escogeremos el *MathItem* que queramos mover de una lista guardada de antemano con los *MathItems* que estemos tocando en el momento de pulsar el gatillo. Esta elección se basa en lo que entendemos que es más difícil de tocar en la vida real. Sin duda lo más difícil de tocar es un vértice, ya que representa un punto en el espacio. Le sigue la arista, circunferencia, elipse, cara, círculo, esfera, cono, cilindro y, por último, cónica. En caso de no elegir ninguno, indica que no estamos tocando ningún *MathItem* y no haría nada.

```
1 public class Neotrie_Controller : MonoBehaviour
2 {
3     IEnumerator MoveMathItem()
4     {
5         MathItem mi = MathItemPriority();//MathItem seleccionado por
              prioridad
6         if (mi != null)
7             yield return StartCoroutine(mi.MoveMathItem(transform.
                  GetChild(0).gameObject));//Pasamos la bolita rosa
8         yield return new WaitForEndOfFrame();
9         ClearTouching();
10    }
11 }
```

Código 4.2: Selección de MathItem.

Si nos fijamos en las manos de los jugadores en la Figura 2.6, veremos que en sus palmas tienen una pequeña bolita rosa. Esta hará la función de puntero de nuestra mano al interactuar con nuestras construcciones. Además, la usaremos como punto de referencia para nuestros movimientos.

Por último, llegamos a la corrutina más importante, la que nos permite mover cualquier *MathItem*. Al igual que a la hora de la creación hemos tenido en cuenta que un *MathItem* viene determinado por sus vértices, también podemos decir que la posición de sus vértices determina la posición del *MathItem*. Lo podemos ver en el siguiente diagrama.

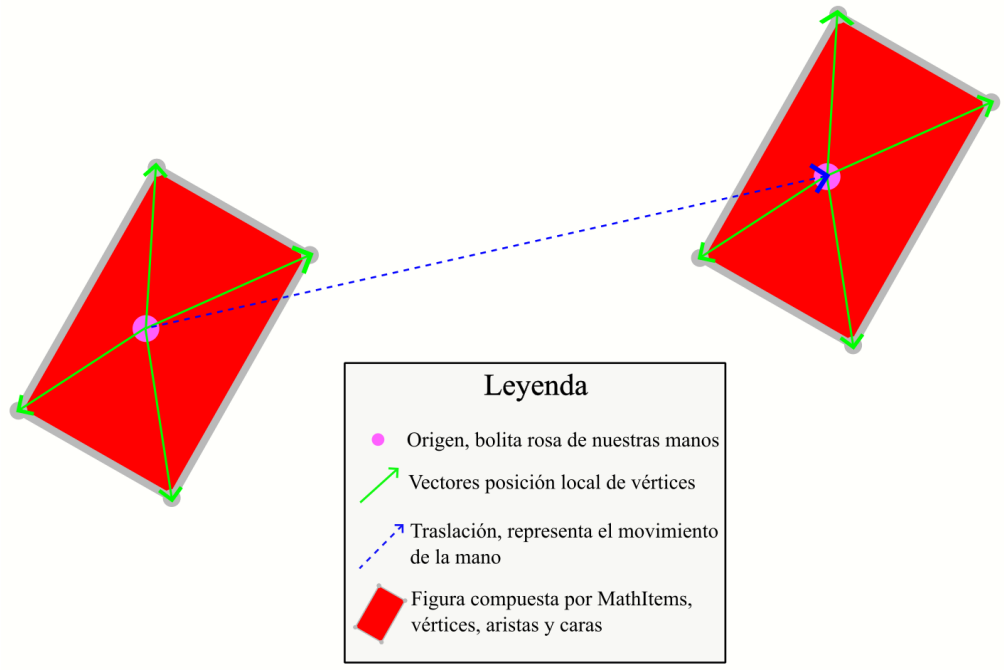

Figura 4.2: Traslación de figura.

Por tanto, solo necesitamos colocar los vértices en las posiciones correctas. Posteriormente, el *MathItem* se actualizará debido al evento y se colocará donde corresponda a partir de sus vértices. Veámoslo:

```
1 public abstract class MathItem : MonoBehaviour, IDefRemovable,
      IRestaurable, ISelectable, IEquatable<MathItem>
2 {
3     public IEnumerator MoveMathItem(GameObject other)//Other = bolita
          rosa
4     {
5         List<Vector3> dif = new();//Lista de vectores vertice-> bolita
6         Vector3 otherOrigin = other.transform.position;//Variables de
              control
7         Vector3 otherCurrent;
8         foreach (Vertice v in Vertices)
9         {
10            dif.Add(other.transform.position - v.transform.position);
11            v.UpdateDependantMI.Invoke();//Indicamos que el vertice se
                  esta actualizando, los mathitems relacionados se
                  actualizaran tambien
12        }
13        ControllerManager cm = other.GetComponentInParent<
              ControllerManager>();
14        while (cm.trigger.press)
15        {
16            otherCurrent = other.transform.position;
17            if ((otherCurrent - otherOrigin).magnitude > 0.007f)
18            {
19                otherOrigin = otherCurrent;
```

```
20                  for (int i = 0; i < Vertices.Count; i++)
21                      Vertices[i].transform.position = otherCurrent - dif[i
                            ];
22              }
23              yield return null;//Siguiente frame
24          }
25          for (int i = 0; i < Vertices.Count; i++)
26          {
27              Vertices[i].FinishUpdateMI.Invoke();//El vertice deja de
                    actualizarse, los mathitems relacionados pueden dejar de
                    actualizarse
28          }
29      }
30  }
```

Código 4.3: Corrutina de movimiento general de MathItem, traslación.

Primero guardaremos los vectores diferencia entre nuestro origen y la posición inicial de los vértices. Esto nos dará las coordenadas locales de los vértices en nuestro sistema de referencia. Después, indicaremos que los vértices que vamos a mover empiecen a actualizarse con su evento. Esto hará que todos los *MathItems* implicados, es decir, suscritos a esos *eventos*, empiecen a actualizarse a su vez. Luego, mientras mantengamos pulsado el gatillo, comprobaremos con las variables de control si ha habido suficiente movimiento como para transmitirlo a los vértices, una vez por frame. Esto se establece para evitar que las personas con menos pulso realicen movimientos erráticos. Además corrige ciertas imperfecciones en el rastreo de la posición del propio hardware con el que estemos jugando. Acabaremos el movimiento marcando el fin de la actualización de los vértices.

Modo Traslación

Antes hemos mencionado el concepto de *Figura*. En este modo lo usaremos, ya que su objetivo no es mover elementos específicos dentro de una *Figura*, sino ella misma. Con este modo podremos aplicar traslaciones.

Hasta ahora, no hemos tratado como interactúan unos objetos con otros. *Unity* pone a nuestra disposición un sistema de colisiones a través de componentes llamados *Colliders*. Hay dos tipos, el primero son los primitivos que constan de una forma predefinida, cubos, esferas y cápsulas. Estos están altamente optimizados y es recomendable su uso siempre que sea posible. El otro tipo lo forman los *MeshColliders*. Estos colisionadores se calculan en tiempo real basados en la malla del objeto, es decir, su forma específica. Son mucho más pesados en cuanto a costo de rendimiento. Además, si necesitáramos modificar la malla del objeto, también necesitamos recalcular su *MeshCollider*.

Cada *MathItem* posee su *Collider*, ya que son elementos con los que necesitamos interactuar. Las figuras no lo poseen, ya que no sabemos qué componentes la van a formar. Además, el costo de unir todas las mallas de los componentes que la forman para hallar su *MeshCollider* en tiempo real sería demasiado alto. Por consiguiente, en este modo, para seleccionar la figura que deseemos mover, solo tendremos que irnos

al objeto padre del *MathItem* que estemos seleccionando. Para simplificar la elección de la figura, cogeremos el primer vértice del *MathItem* con mayor preferencia, y luego, cogeremos su objeto padre.

Al igual que en el modo edición, el evento "pulsar gatillo" ejecuta este método. En este caso, el modo Traslación de la mano es el 4.

```
1 public class Neotrie_Controller : MonoBehaviour
2 {
3     //Suscrito a Evento "pulsar gatillo"
4     public void UseControllerMode()
5     {
6         switch (useMode.MenuNum)
7         {
8             case 4: //Modo Traslacion
9                 MoveObjectMode();
10                break;
11        }
12    }
13 }
```

Código 4.4: Modo Traslación.

Ahora realizaremos la elección de figura a mover. Como hemos dicho, esta figura será el padre del primer vértice del *MathItem* con mayor prioridad, de esto se encargará el método *ReSelectVert*. El vértice elegido se guardará en la variable *SelectVert*. Si no se elige ninguno, se suspenderá la ejecución del método, ya que no estaremos tocando ninguna *figura*.

```
1 public class Neotrie_Controller : MonoBehaviour
2 {
3     public Vertice SelectVert = null;
4     void MoveObjectMode()
5     {
6         ReSelectVert();//vertice elegido -> SelectVert
7         if (SelectVert == null) return;//Control seleccion
8         StartCoroutine(MoveParent());//Corrutina de movimiento del padre
9     }
10 }
```

El procedimiento restante es exactamente el mismo seguido para mover los vértices en el modo edición, mostrado en la Figura 4.2. En este caso, el *transform* de la *figura* hace el papel de vértice a mover, mientras que el vector que une la bolita rosa de nuestras manos y el *transform* de la *figura* es la posición local de la *figura* respecto de la mano.

```
1 public class Neotrie_Controller : MonoBehaviour
2 {
3     public Figure ParentMoving = null;
4     public IEnumerator MoveParent()
5     {
```

```
6         ParentMoving = SelectVert.transform.GetComponentInParent<Figure
              >();//Tomamos la figura del vertice
7         Vector3 ObjMoveVector = ControlSphere.transform.position -
              ParentMoving.transform.position;//Posicion relativa de la
              figura con respecto de la mano
8         Vertice[] verts = ParentMoving.Get<Vertice>();//Vertices de la
              figura
9         Vector3 originPos = ControlSphere.transform.position;//Variables
              de control
10        Vector3 currentPos;
11        foreach (Vertice v in verts)
12            v.UpdateDependantMI.Invoke();//Vertices inician actualizacion
13        while (cm.trigger.press)
14        {
15            currentPos = ControlSphere.transform.position;
16            if ((originPos - currentPos).magnitude > 0.007f)
17            {
18                ParentMoving.transform.position = currentPos -
                      ObjMoveVector;//Asignamos nueva posicion
19                originPos = currentPos;
20            }
21            yield return null;
22        }
23        foreach (Vertice v in verts)
24            v.FinishUpdateMI.Invoke();
25        ClearTouching();
26        ParentMoving = null;
27    }
28 }
```

Código 4.5: Corrutina de movimiento de la figura, traslación.

Con esto ya tenemos tratadas las traslaciones de Figuras.

Modo Agarre

El último de los modos de movimiento que trataremos en este trabajo es el modo Agarre. Este modo, como el anterior, también se centra en las *Figuras* y permite al jugador experimentar la sensación de agarrar cualquier construcción geométrica. Esto es, podrá aplicar movimientos de traslación y rotación a las *Figuras*.

Seguiríamos el mismo procedimiento para iniciar el movimiento. En este caso pulsaremos el gatillo con el modo agarre seleccionado (modo 9).

```
1 public class Neotrie_Controller : MonoBehaviour
2 {
3     //Suscrito a Evento "pulsar gatillo"
4     public void UseControllerMode()
5     {
6         switch (useMode.MenuNum)
7         {
8             case 9: //Modo Agarre
9                 GrabObjectMode();
```

```
                    break;
            }
        }
}
```

Código 4.6: Modo Agarre.

El siguiente paso es idéntico al del modo traslación, salvo que ahora iniciamos otra corrutina distinta de movimiento.

```
public class Neotrie_Controller : MonoBehaviour
{
    public Vertice SelectVert = null;
    void GrabObjectMode()
    {
        ReSelectVert();//vertice elegido -> SelectVert
        if (SelectVert == null) return;//Control seleccion
        StartCoroutine(MoveParentWithRotation());//Corrutina de
            movimiento del padre
    }
}
```

Ahora bien, calcular el movimiento de rotación y traslación ya no resulta tan sencillo como un simple cálculo de vectores. *Unity* implementa las llamadas matrices TRS, donde sus siglas en inglés significan traslación, rotación y escala.

```
public static Matrix4x4 TRS(Vector3 pos, Quaternion q, Vector3 s);
```

Se trata de una matriz que sitúa el objeto en la posición dada por la variable *pos*, con la orientación dada por q y la escala definida por s según su documentación [8]. Desde el punto de vista matemático, consiste en una de una matriz ampliada que representa una transformación afín,

$$\begin{pmatrix} \vec{v'} \\ 1 \end{pmatrix} = \begin{pmatrix} \mathbf{A} & \vec{pos} \\ \vec{0} & 1 \end{pmatrix} \begin{pmatrix} \vec{v} \\ 1 \end{pmatrix}.$$

Usando esta herramienta y realizando los cálculos adecuados de productos de matrices lograremos obtener la matriz TRS *move* que guardará la posición y rotación finales que deberán asignarse al *transform* de la *figura*.

Aparte del uso de matrices, la implementación de la corrutina de movimiento sigue la misma estructura que la del modo traslación (código 4.5).

```
public class Neotrie_Controller : MonoBehaviour
{
    public Figure ParentMoving = null;
    public IEnumerator MoveParentWithRotation()
    {
        ParentMoving = SelectVert.transform.GetComponentInParent<Figure
            >();
```

```
7         Matrix4x4 q = Matrix4x4.TRS(ControlSphere.transform.position,
              ControlSphere.transform.rotation, new Vector3(1, 1, 1));
8         Matrix4x4 p = Matrix4x4.TRS(ParentMoving.transform.position,
              ParentMoving.transform.rotation, new Vector3(1, 1, 1));
9         Matrix4x4 pq = p.inverse * q;
10        Vertice[] verts = ParentMoving.Get<Vertice>();
11        foreach (Vertice v in verts)
12            v.UpdateDependantMI.Invoke();
13        Matrix4x4 move = Matrix4x4.identity;
14        while (cm.trigger.press)
15        {
16            move = Matrix4x4.TRS(ControlSphere.transform.position,
                  ControlSphere.transform.rotation, new Vector3(1, 1, 1)) *
                   pq.inverse;
17            ParentMoving.transform.SetPositionAndRotation(move.GetColumn
                  (3), move.rotation);
18            yield return null;
19        }
20
21        foreach (Vertice v in verts)
22            v.FinishUpdateMI.Invoke();
23        ClearTouching();
24        ParentMoving = null;
25    }
26 }
```

Código 4.7: Corrutina de movimiento de la Figura, traslación y rotación.

5 Nuevos MathItem

En lo que llevamos de trabajo solo hemos hablado de los cambios que hemos tenido que aplicar en diversos puntos del software para generalizar todos los tipos de *MathItems*. Ahora, nos centraremos en crear dos nuevos tipos de *MathItems*. El primero de ellos incluirá un caso especial de la circunferencia. Se trata de los arcos de circunferencia y sectores circulares. El segundo será totalmente nuevo en el juego: las cónicas en el espacio.

Para construir estos nuevos tipos, usaremos la herramienta de desarrollo *ShaderGraph* de *Unity* que no se ha usado con anterioridad en el software.

5.1 ShaderGraph

Un *GameObject* como tal no tiene parte gráfica. La visualización de un *GameObject* en la escena depende de dos componentes claves. El primero es su malla, *mesh* en inglés. Esta guarda la forma del objeto con una serie de vértices y triángulos. La malla de un objeto se encuentra en su componente *Mesh Filter*. Este solo actúa de contenedor y todavía no dota a la malla de gráficos. El segundo es su *material*, el cual aporta textura a la malla, ya sea mediante un color fijo o imágenes prediseñadas de antemano conocidas como texturas.

Los materiales a su vez dependen de un *shader*. Estos son programas un tanto especiales que se ejecutan directamente en la GPU (Graphics Processing Unit) de nuestro dispositivo. Esto lo hace trabajar mucho más rápido, pero está muy limitado en cuanto a funcionalidades. La principal falencia es su escasa capacidad de comunicación entre los scripts que podemos estar usando como componentes y los *shaders*. Con esto nos referimos a que, por ejemplo, no podemos pasar un array como variable. Además, estos programas usan *HLSL*, un lenguaje de alto nivel desarrollado por *Microsoft*. Aunque esté diseñado para parecerse lo más posible a *C#*, no deja de ser un lenguaje nuevo y presenta sus pequeñas peculiaridades. Sortearemos este problema usando la documentación aportada por *Microsoft* en [11].

Por otra parte, podemos aprovechar la potencia del procesador gráfico para hacer cálculos en cada píxel. Así conseguiremos resultados gráficos interesantes sin tener que realizar operaciones costosas, como generar o modificar una malla.

Unity, en su afán de facilitar el acceso a todo lo posible, nos aporta la herramienta *ShaderGraph*. En vez de tener que aprender un nuevo lenguaje de programación, esta herramienta nos permite generar *shaders* a través de pseudocódigo con estilo de nodos y aristas. Cada nodo representa una función o método y las entradas y salidas de datos de ellos se controlan mediante aristas. En la Figura 5.1 mostramos la interfaz de esta nueva herramienta.

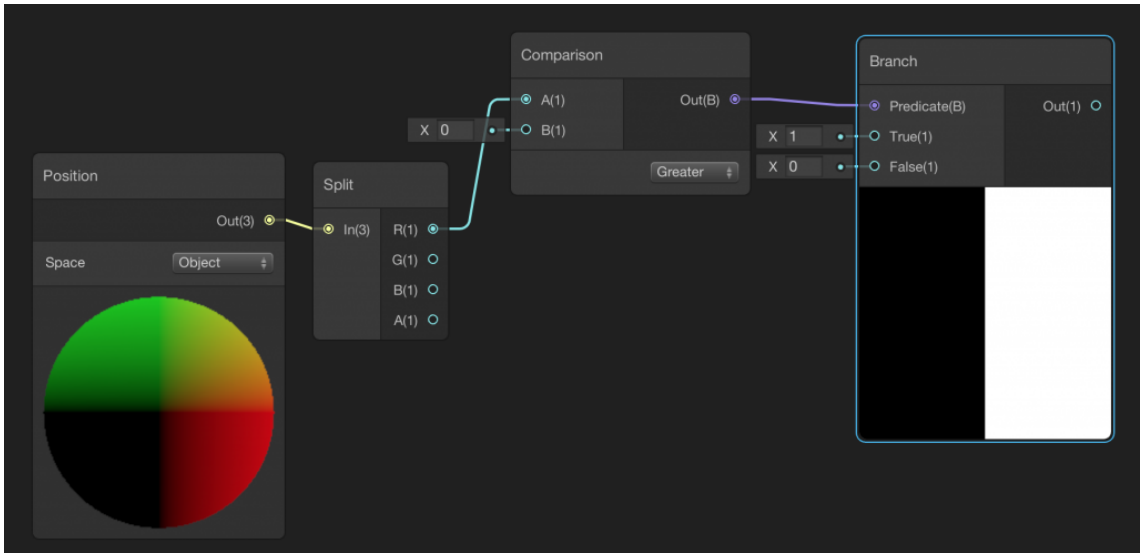

Figura 5.1: Herramienta *ShaderGraph*.

Cabe destacar que esta es solo una herramienta y, como cualquier otra, no llega a ofrecernos la libertad de programar directamente en su lenguaje, tal y como comprobaremos en la sección 5.3.

5.2 Arcos de circunferencia y sectores circulares

Los modelos de circunferencia y círculo ya se encontraban implementados en el juego. Además, en el caso de la circunferencia, esta puede ser determinada de distintas formas: (a) circunferencia que pasa por tres puntos, (b) indicando centro, radio y plano que la contiene, indicando la normal o (c) cogiendo un tercer punto fijando el plano.

Figura 5.2: Casos de circunferencia (a), (b) y (c).

En cualquier caso, estas referencias vienen dadas por vértices, tal y como hemos definido en la nueva filosofía de trabajo.

Para la introducción del arco de circunferencia se pensó en una primera versión la cual constaba en la unión de 360 pequeñas aristas que conformaban la circunferencia, las cuales se irían activando y desactivando según el ángulo permitido del arco. Por motivos de rendimiento, esta idea se descartó, ya que eran demasiados objetos en escena solo para representar un único *MathItem*.

En este punto, investigando otras opciones, descubrimos los *shaders* y la herramienta *ShaderGraph*, con la cual pudimos diseñar un *shader* capaz de aportar al objeto el comportamiento que deseábamos.

La idea a la hora de crear un *shader* es tener el menor número de entradas posibles. Nos bastó con cuatro, las tres primeras definían la circunferencia, optamos por la opción de centro, radio y normal. La última sería el ángulo del arco. Ya que tenemos cuatro dependencias, la primera versión o forma de hacer un arco en *Neotrie VR* dependerá de cuatro vértices. El primero define el centro, el segundo el radio de la circunferencia, el tercero el ángulo del arco y el cuarto la normal. Luego, a la hora de actualizar la circunferencia, haremos los cálculos necesarios para situar el *MathItem* en su posición correcta y al mismo tiempo, le pasaremos los datos al *shader* para que solo dibuje los píxeles que nos interesan.

```
1  //Actualizamos variables
2  center = Vertices[0].transform.position;
3  normal = Vertices[3].transform.position - Vertices[0].transform.position;
4  radius = (Vertices[0].transform.position - Vertices[1].transform.position
       ).magnitude;
5  angle = Vector3.SignedAngle(Vertices[2].transform.position - center,
       center - Vertices[1].transform.position, normal);
6  //Pasamos los datos al shader del material (mat)
7  mat.SetVector("_Vertex1Pos", Vertices[1].transform.position);
8  mat.SetVector("_CenterPos", center);
9  mat.SetVector("_Normal", normal);
10 mat.SetFloat("_Angle", angle);
```

Código 5.1: Arco determinado por 4 puntos.

Para terminar, solo nos queda ver el *shader* que realiza el trabajo. La primera parte la podemos ver en la Figura 5.3. El nodo *Position* nos aporta la posición global del píxel a dibujar. Con dicha posición, podemos calcular el ángulo que forman los vectores *CenterPosPosition* y *CenterPosVertex1Pos*, con el vector normal como eje de giro.

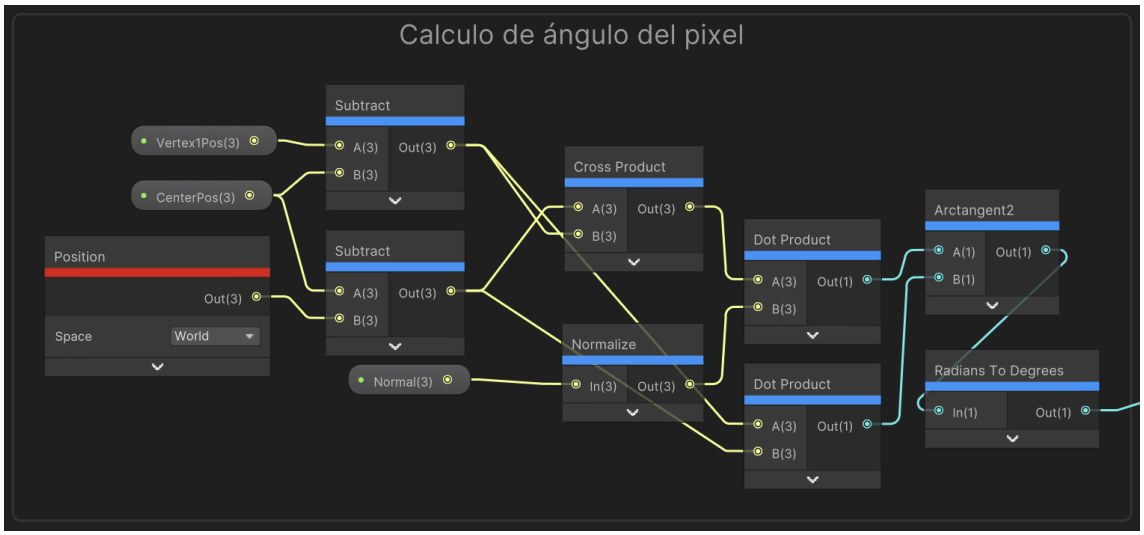

Figura 5.3: Cálculo del ángulo hasta el píxel.

Por último, compararemos el ángulo obtenido del píxel y el ángulo que le hemos pasado. Si es menor, dejaremos la transparencia intacta del color multiplicando el canal de transparencia alfa por la unidad. Puede que el color tenga transparencia independientemente si el píxel debiera dibujarse o no, por eso tomamos esta consideración. En caso contrario, multiplicamos por 0, volviendo invisible cada píxel que se pase del ángulo estipulado.

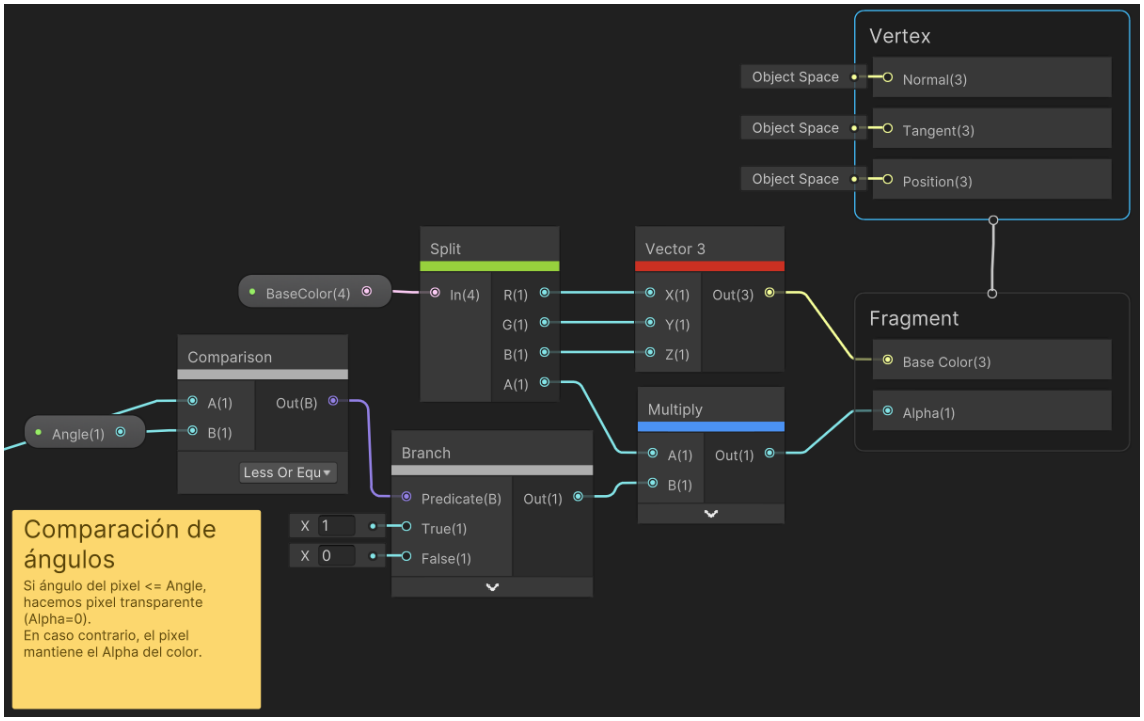

Figura 5.4: Control de transparencia según ángulo.

Como hemos visto en la Figura 5.2, existen varias formas de determinar una circunferencia y por tanto el arco. Queríamos buscar una forma más simple de determinar el arco de circunferencia, con solo tres vértices. La solución la obtuvimos basándonos en el caso de circunferencia que pasa por tres puntos.

```
1  //Calculos previos
2  Vector3 N1 = Vertices[1].transform.position - Vertices[0].transform.
       position;
3  Vector3 N2 = Vertices[2].transform.position - Vertices[1].transform.
       position;
4  normal = Vector3.Cross(N1, N2);
5  Vector3 D = new(Vector3.Dot(N1, (Vertices[0].transform.position +
       Vertices[1].transform.position) / 2), Vector3.Dot(N2, (Vertices[1].
       transform.position + Vertices[2].transform.position) / 2), Vector3.
       Dot(normal, Vertices[0].transform.position));
6  Vector3 A = new(N1.x, N2.x, normal.x);
7  Vector3 B = new(N1.y, N2.y, normal.y);
8  Vector3 C = new(N1.z, N2.z, normal.z);
9  center = Geometry_calcs.CramerSolution(A, B, C, D);
10 radius = Vector3.Distance(Vertices[0].transform.position, center);
11 //Elegimos angulo mayor
12 angle = Mathf.Max(
```

```
13              Vector3.SignedAngle(Vertices[2].transform.position - center,
                    center - Vertices[0].transform.position, normal),
14              Vector3.SignedAngle(Vertices[1].transform.position - center,
                    center - Vertices[0].transform.position, normal)
15              );
16 //Pasamos los datos al shader del material (mat)
17 mat.SetVector("_Vertex1Pos", Vertices[0].transform.position);
18 mat.SetVector("_CenterPos", center);
19 mat.SetVector("_Normal", normal);
20 mat.SetFloat("_Angle", angle);
```

Código 5.2: Arco por 3 puntos.

Además, ya que este *shader* no depende de casos específicos de circunferencias, realizará su función mientras se le pasen las variables adecuadas. Por ello, será capaz de representar también sectores circulares. Los resultados dentro del juego los podemos ver en las Figuras 1 y 2.

5.3 Cónicas

Al igual que con los arcos, la implementación de las cónicas ha constado de varias etapas. La primera idea consistió en modelar todos los tipos de cónica y hallar sus métodos de posicionamiento posteriormente. Esta idea quedó descartada ya que no podíamos pasar de un tipo de cónica a otro fácilmente.

La idea que surgió después fue más interesante, consistía en usar el componente *LineRenderer*. Este se trata de un componente nativo de *Unity* que dibuja segmentos que unen una lista de puntos. Para calcular los puntos nos basaríamos en el Teorema de Pascal.

Teorema 5.1 (Pascal, 1639)**.** *Si un hexágono cualquiera de vértices $ACEBFD$, se encuentra inscrito en una cónica, entonces los puntos de corte $P = AB \cap DE$, $Q = AF \cap CD$ y $R = BC \cap EF$ son colineales.*

La demostración la podemos encontrar en [2, pág. 74 – 76]. Lo interesante de este teorema es su corolario, que nos permitirá construir cualquier cónica a través de cinco puntos en el espacio.

Corolario 5.1.1. *Dados un pentágono de vértices $ABCDE$ y un punto arbitrario X, definimos $P = AE \cap BD$, $Q = BX \cap CE$, $R = CD \cap PQ$, y por último $F = BX \cap AR$. Entonces, existe una única cónica que pasa por los puntos A, B, C, D, E y F.*

Variando el punto X de la Figura 5.5, podemos obtener todos los puntos que conforman la cónica. Hemos preparado un applet de *GeoGebra* para que podamos ver este resultado [12]. Con respaldo de la teoría, la idea del *LineRenderer* se hace viable. Su implementación resulta sencilla, ya que solo consta de intersecciones de rectas. Por desgracia, la práctica no siempre respalda la teoría. Debido a que las cónicas siempre están contenidas en un plano y *Neotrie VR* trabaja en el espacio, ocurren errores de precisión que hay que tener en cuenta. Obviamente, *Unity* no está pensado para hacer

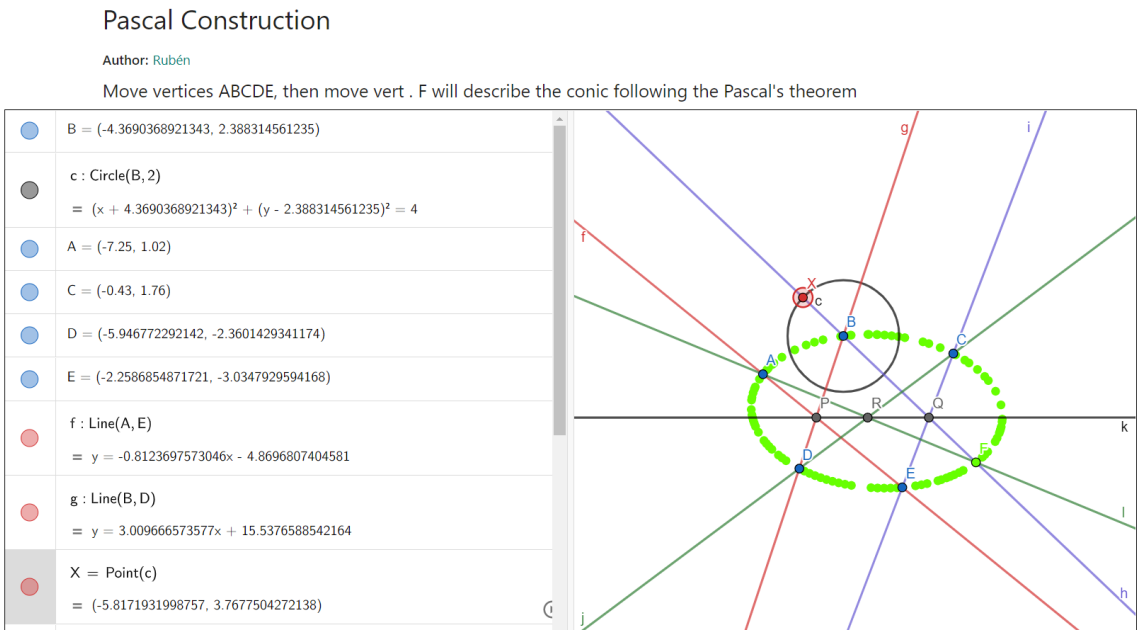

Figura 5.5: Construcción de cónica por el Corolario del Teorema de Pascal en *GeoGebra*.

cálculos tan precisos como otros softwares como *Matlab* o *Mathematica*. No obstante, el trabajo no fue en vano y pudimos introducir los puntos de Pascal en el juego.

Por último, logramos representar las cónicas de manera fluida y dinámica mediante un *shader*. Recordemos la definición original de cónica:

Definición 5.1. *Llamaremos cónica a todas las curvas resultantes entre la intersección de un cono completo y un plano.*

El plano lo podemos obtener a partir de tres puntos. Después, podemos calcular la distancia del píxel al plano y la usaremos para resaltar la curva.

Usaremos una simple tolerancia para darle un pequeño grosor a la intersección. Esto la hará visible al usuario. Cabe destacar que al contrario que en el caso de arcos, ahora no queremos hacer completamente transparente el cono, ya que el usuario podría resultarle molesto poder perderlo en la escena. Por ello, en vez de multiplicar por 0, lo haremos por 0,1, para así conservar solo un 10 % de la transparencia original.

Esta idea de implementación ha logrado integrarse en el juego, ya que resulta interesante ver la definición básica de cónica en acción, ver Figuras 3, 4, 5 y 6 del Apéndice. Aun así, decidimos intentar un enfoque más directo. Podemos determinar una cónica con solo tres puntos y algunas restricciones.

¿Por qué hemos decidido usar tres puntos? Porque estos definen un plano, el que usaremos como lienzo para, con la ayuda de un *shader*, dibujar la cónica correspondiente. Además, si solo queremos cónicas de un tipo específico podremos fijarlo de antemano. Sabemos por el Corolario 5.1.1 que necesitamos cinco puntos, estos los hallaremos realizando los cálculos necesarios y luego se los pasaremos al *shader*, siguiendo la dinámica de trabajo usada en el caso de los arcos de circunferencia, ver Figuras 7, 8 y 9.

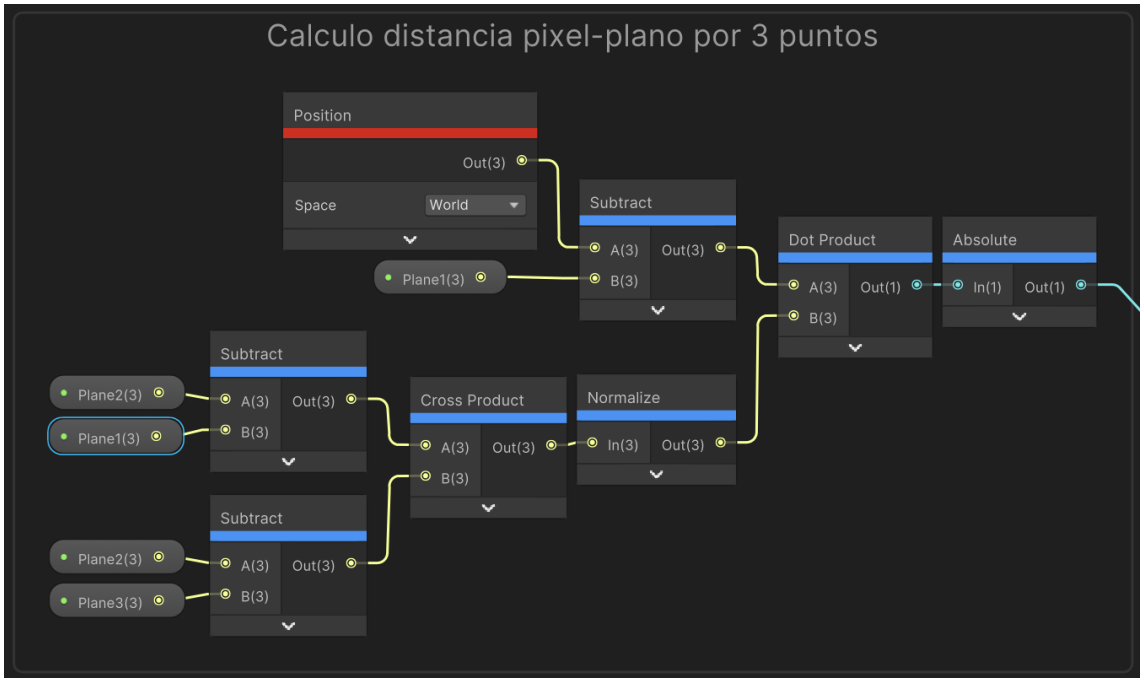

Figura 5.6: Distancia píxel-plano.

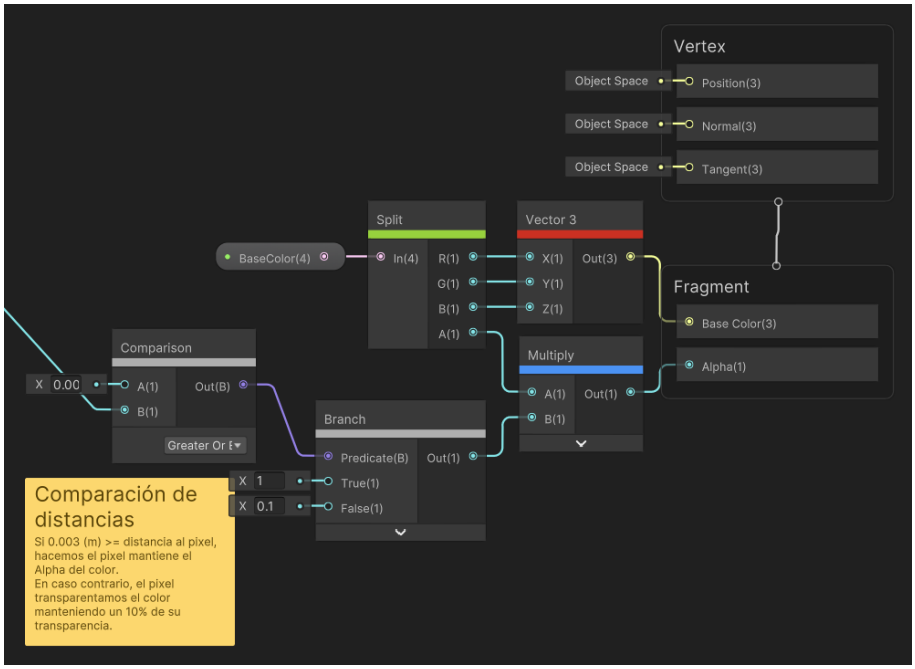

Figura 5.7: Control de transparencia según distancia al plano.

En caso de que no elijamos el tipo especifico, no serán necesarios cálculos adicionales ya que contaremos con los cinco puntos. Los tres primeros, en cualquier caso, se usarán además para situar el plano que utilizaremos como lienzo. En el caso general, los dos últimos puntos no tienen por qué estar en el plano. Esto no es problema ya que el método *WorldToUV* se encarga de trabajar con las proyecciones en el plano.

```
Vector4[] ConicPoints = new Vector4[5];
Vector3 P;
Plane p = new(Vertices[0].transform.position, Vertices[1].transform.position, Vertices[2].transform.position);
switch (caso)
{
    case 0://Ellipse
        ConicPoints[4] = Vertices[2].transform.position;
        focus1 = Vertices[0].transform.position;
        focus2 = Vertices[1].transform.position;
        P = Vertices[2].transform.position;
        center = (focus1 + focus2) * 0.5f;
        normal = Vector3.Cross(focus1 - center, P - center).normalized;
        radius1 = ((P - focus1).magnitude + (P - focus2).magnitude) * 0.5f;
        radius2 = Mathf.Sqrt(Mathf.Abs(Mathf.Pow(radius1,2) - Mathf.Pow((focus1-center).magnitude,2)));
        ConicPoints[0] = center + radius1 * (focus1 - center).normalized;
        ConicPoints[1] = center - radius1 * (focus1 - center).normalized;
        ConicPoints[2] = center + radius2 * Vector3.Cross(normal, focus1 - center).normalized;
        ConicPoints[3] = center - radius2 * Vector3.Cross(normal, focus1 - center).normalized;
        break;
    case 1://Parabola
        ConicPoints[4] = Vertices[2].transform.position;
        focus1 = Vertices[0].transform.position;
        center = focus1;
        focus2 = Vertices[1].transform.position;
        float PF1 = Vector3.Distance(Vertices[2].transform.position, focus1);
        Vector3 Pr = Vertices[2].transform.position + (focus2 - focus1).normalized * PF1;
        Vector3 F1r = Geometry_calcs.ProjectOnLine(Pr, focus1, focus2);
        ConicPoints[0] = (focus1 + F1r) * 0.5f;
        P = Vector3.Cross(p.normal, focus1 - F1r);
        ConicPoints[1] = focus1 + P;
        ConicPoints[2] = focus1 - P;
        ConicPoints[3] = focus1 - F1r;
        ConicPoints[3].w = 1;
        break;
    case 2://Hyperbola
        ConicPoints[4] = Vertices[2].transform.position;
        P = Vertices[2].transform.position;
        focus1 = Vertices[0].transform.position;
        focus2 = Vertices[1].transform.position;
        center = (focus1 + focus2) * 0.5f;
        float a = Mathf.Abs(Vector3.Distance(P, focus1) - Vector3.Distance(P, focus2)) * 0.5f;
        Vector3 focus = Vector3.Distance(P, focus1) > Vector3.Distance(P, focus2) ? focus1 : focus2;
        ConicPoints[0] = (focus - center).normalized * a + center;
        ConicPoints[1] = -(focus - center).normalized * a + center;
        float r = Mathf.Pow(Vector3.Distance(focus1, focus2),2) / (4 * a) - a;
        Vector3 cross = Vector3.Cross(p.normal, focus1 - focus2).
```

```
            normalized;
47          ConicPoints[2] = cross * r + focus;
48          ConicPoints[3] = -cross * r + focus;
49          break;
50      default:
51          center = Vertices.ToArray().GetMiddlePoint();
52          for (int i = 0; i < Vertices.Count; i++)
53              ConicPoints[i] = Vertices[i].transform.position;
54          break;
55  }
56  for (int i = 0; i < 5; i++)
57  {
58      ConicPoints[i] = WorldToUV(ConicPoints[i]);//Por motivos del shader,
            tenemos que transformar las coordenadas de los puntos
59      thisMat.SetVector("_p" + i, ConicPoints[i]);
60  }
```

Código 5.3: Selección de puntos en la cónica según caso.

En esta ocasión, nos ha sido imposible usar la herramienta *ShaderGraph*. El *shader* se ha conseguido adaptando el que encontramos en [13], desde el lenguaje *GLSL* en el que estaba programado al de *HLSL* que usa *Unity*, siguiendo las tablas de conversión de [14]. El *shader* completo lo podemos ver en el Apéndice 6.

Por último, como podemos apreciar en las líneas 177 – 181 del *shader* 6, a los cinco puntos que definen la cónica se les ha modificado el color. Esto nos permite identificar, dentro del juego, los cinco puntos que definen la cónica. Estos puntos son los resultados de los cálculos realizados en caso de requerir un tipo de cónica específico, ver Figuras 7, 8 y 9. En el caso general, estos puntos coinciden con los vértices que la definen o con sus proyecciones en el plano, ver Figuras 10, 11, 12 y 13. En cualquier caso, vemos que la línea 181 se encuentra comentada, esto se debe a que, por construcción, se ha decidido que el tercer vértice que toquemos siempre se encuentre en la cónica.

6 Futuras mejoras y conclusiones

Neotrie VR está en constante expansión gracias a la estrecha colaboración entre los equipos de desarrollo, profesores e investigadores [1]. El software ya cuenta con una gran variedad de herramientas y opciones para desempeñar sus funciones de enseñanza y aprendizaje de la geometría, como se puede comprobar por el gran número de publicaciones citadas en [5].

Ahora bien, el rendimiento es un problema latente en el juego. La optimización es esencial, y para que esta sea viable se ha de tener una base sólida sobre la que trabajar. Poco a poco se va estableciendo esta base, por ejemplo, con la mejora de la unificación de los *MathItems* que le aporta una plantilla de trabajo, que además de permitir expandir el software, permite que la programación sea más eficiente al no tener que separar por todos los tipos, por ejemplo, a la hora de iterar sobre una lista de *MathItems* para aplicarles cierto método común o la mejora en el movimiento con la introducción de los eventos. Está claro que en el diseño de algo nuevo se van a cometer errores, o se va a obtener una versión rudimentaria que funcione. Pero en un software de estas características es complicado llegar a una versión final, cuando hay falta de optimización de base.

Por otra parte, no todo es culpa de falta de optimización. El motor gráfico de *Unity* trabaja únicamente usando la potencia de un solo núcleo de nuestros procesadores. Actualmente, estos mejoran su potencia añadiendo núcleos extra para realizar tareas en paralelo. Esto *Unity* no lo aprovecha, pero están trabajando en ello con su proyecto *DOTS*, [9]. Por ahora se encuentra en fase beta, pero tiene unos resultados sorprendentes. Básicamente trata de cambiar la dinámica de trabajo, adaptando un nuevo estilo de programación, la programación orientada a datos. Junto con las herramientas del sistema de trabajos de *C#* y un compilador extremadamente eficiente. Nos permite paralelizar tareas que se ejecutarán en los núcleos adicionales de nuestro procesador cuando estén disponibles.

Existen muchos procesos que se pueden paralelizar en *Neotrie VR*, por ejemplo, los de movimiento. Por lo que estaremos a la espera de noticias de este proyecto de *Unity* para seguir optimizando el juego.

Una vez implementemos estas mejoras de rendimiento, con la generalización construida a lo largo de este trabajo, podríamos introducir las superficies cuádricas y otros tipos de *MathItems*, como los que tiene *GeoGebra* incluidos en su catálogo. Después de completar esa tarea, el software estaría listo para trabajar en la importación de ficheros de *GeoGebra* en *Neotrie*, combinando las ventajas de ambos softwares.

Bibliografía

[1] J.L. Rodríguez, I. Romero, A. Codina, *The Influence of NeoTrie VR's Immersive Virtual Reality on the Teaching and Learning of Geometry*, Mathematics 9 (2021), 2411. https://doi.org/10.3390/math9192411

[2] H.S.M. Coxeter, S.L. Greitzer, *Geometry Revisited*, 1996.

[3] Página web de *Neotrie VR*: https://www2.ual.es/neotrie/.

[4] Página web del anuncio del modo multijugador de *Neotrie VR*: https://www2.ual.es/neotrie-vr-multijugador-en-la-prensa/

[5] Página web de artículos sobre *Neotrie VR*: https://www2.ual.es/research/

[6] Página web de *Unity*: https://unity.com/es.

[7] Página web de aprendizaje de *Unity*: https://learn.unity.com.

[8] Documentación de *Unity*: https://docs.unity3d.com/ScriptReference/.

[9] Proyecto DOTS de *Unity*: https://unity.com/dots.

[10] *Unity* MonoBehaviour Flujo de trabajo: https://docs.unity3d.com/flowchart.svg.

[11] Documentación de HLSL por *Microsoft*: https://learn.microsoft.com/en-us/hlsl.

[12] Applet de GeoGebra de construcción de cónicas a partir del Teorema de Pascal: https://www.geogebra.org/m/jynaffnx.

[13] Shader de cónicas en geometria proyectiva: https://www.shadertoy.com/view/lljBWz.

[14] Página web con tablas de conversión entre *GLSL* y *HLSL*: https://learn.microsoft.com/glsl-to-hlsl-reference.

Apéndice

Shader: Cónica por cinco puntos

```
Shader "Unlit/ConicPlane"
{
    Properties
    {
        [MainColor] _BaseColor ("Color", Color) = (1,1,1,1)
        _p0 ("p0", Vector) = (0,0,0,0)
        _p1 ("p1", Vector) = (0,0,0,0)
        _p2 ("p2", Vector) = (0,0,0,0)
        _p3 ("p3", Vector) = (0,0,0,0)
        _p4 ("p4", Vector) = (0,0,0,0)
        _lwidth ("Curve Width", float) = 0.0006
        _pwidth ("Point Width", float) = 0.0015
    }
    SubShader
    {
        Tags {"Queue"="Transparent" "RenderType"="Transparent" }
        LOD 100

        ZWrite Off
        Blend SrcAlpha OneMinusSrcAlpha
        Pass
        {
            Cull Off
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #include "UnityCG.cginc"

            struct appdata
            {
                float4 vertex : POSITION;
                float2 uv : TEXCOORD0;
            };

            struct v2f
            {
                float2 uv : TEXCOORD0;
                float4 vertex : SV_POSITION;
            };
            v2f vert (appdata v)
            {
                v2f o;
                o.vertex = UnityObjectToClipPos(v.vertex);
                o.uv = v.uv;
                return o;
            }
            static const float PI = 3.141592654;
            static const float eps = 1e-4;
            CBUFFER_START(UnityPerMaterial)
                float3 _p0, _p1, _p2, _p3, _p4;
```

```
51                  float4 _BaseColor;
52                  float _lwidth;
53                  float _pwidth;
54              CBUFFER_END
55              // Represent a projective conic as a 3x3 matrix:
56              //
57              // M = (a,d,e,
58              //      d,b,f,
59              //      e,f,c)
60              //
61              // is: axx + byy + czz + 2(dxy + exz + fyz) = 0
62              // calculated as p'Mp for p = (x,y,z) and p' = transpose(p)
63              //
64              // We can treat this as a distance field, scaled by the
65              // (x,y) derivative in order to get correct line widths.
66
67              // With this representation, the dual conic is just the
                   inverse;
68              // if the determinant is zero then there is no dual and the
69              // conic is degenerate.
70
71              // Distance from the conic
72              float dist(float3 p, float3x3 m) {
73                  return dot(p, mul(p , m));
74              }
75
76              // The gradient uses the same matrix.
77              // Don't homegenize the result!
78              float3 grad(float3 p, float3x3 m) {
79                  return mul(p , m) * 2.0;
80              }
81
82              float conic(float3 p, float3x3 m) {
83                  float d = dist(p, m);
84                  float3 dd = grad(p, m);
85                  d = abs(d / (p.z * length(dd.xy))); // Normalize for
                       Euclidean distance
86                  return 1.0 - step(_lwidth, d);
87              }
88              float3x3 inverse(float3x3 m) {
89                  float invdet = 1 / determinant(m);
90                  float3x3 minv;
91                  minv[0][0] = (m[1][1] * m[2][2] - m[2][1] * m[1][2]) *
                       invdet;
92                  minv[0][1] = (m[0][2] * m[2][1] - m[0][1] * m[2][2]) *
                       invdet;
93                  minv[0][2] = (m[0][1] * m[1][2] - m[0][2] * m[1][1]) *
                       invdet;
94                  minv[1][0] = (m[1][2] * m[2][0] - m[1][0] * m[2][2]) *
                       invdet;
95                  minv[1][1] = (m[0][0] * m[2][2] - m[0][2] * m[2][0]) *
                       invdet;
96                  minv[1][2] = (m[1][0] * m[0][2] - m[0][0] * m[1][2]) *
                       invdet;
97                  minv[2][0] = (m[1][0] * m[2][1] - m[2][0] * m[1][1]) *
                       invdet;
```

```
            minv[2][1] = (m[2][0] * m[0][1] - m[0][0] * m[2][1]) * invdet;
            minv[2][2] = (m[0][0] * m[1][1] - m[1][0] * m[0][1]) * invdet;
            return minv;
        }
        // Find a projective mapping taking p0,p1,p2,p3 to
        // triangle of reference and unit point, ie:
        // p0 -> (1,0,0), p1 -> (0,1,0), p2 -> (0,0,1), p3 -> (1,1,1)
        // No three points collinear.
        float3x3 rproject(float3 p0, float3 p1, float3 p2, float3 p3) {//PUNTO DE GUARDADO
            // Just an inverse for the first three points
            // (the triangle of reference). No inverse if collinear.
            float3x3 m = inverse(float3x3(p0, p1, p2)); // column major!
            float3 p3a = mul(p3 , m);
            // Then scale each row so the unit point (1,1,1) is correct
            m = transpose(m);
            // zero components here only if not collinear
            m[0] /= p3a[0];
            m[1] /= p3a[1];
            m[2] /= p3a[2];
            m = transpose(m);
            return m;
        }

        // Construct the conic defined by 5 points.
        // Method taken from "Geometry", Brannan, Esplan & Gray, CUP, 2012
        float3x3 solve(float3 p0, float3 p1, float3 p2, float3 p3, float3 p4) {
            // p takes p0,p1,p2,p3 to triangle of reference and unit point
            float3x3 p = rproject(p0, p1, p2, p3);
            // Now construct a conic through the images of p0-p4,
            float3 p4a = mul(p4 , p);
            float a = p4a.x, b = p4a.y, c = p4a.z;
            float d = c * (a - b);
            float e = b * (c - a);
            float f = a * (b - c);
            float3x3 m = float3x3(0, d, e,
                d, 0, f,
                e, f, 0);
            // And combine the two.
            return mul(mul(p, m),transpose(p));
        }
        float punto(float3 p, float3 q) {
            if (abs(p.z) < eps) return 0.0;
            if (abs(q.z) < eps) return 0.0;
            p /= p.z; q /= q.z; // Normalize
            return 1.0 - step(_pwidth, distance(p, q));
        }

        float linea(float3 p, float3 q) {
            // Just treat as a degenerate conic. Note factor of 2.
```

```
                // We could do this more efficiently of course.
                return conic(p, float3x3(0, 0, q.x,
                    0, 0, q.y,
                    q.x, q.y, 2.0 * q.z));
            }

            float3 join(float3 p, float3 q) {
                // Return either intersection of lines p and q
                // or line through points p and q, r = kp + jq
                return cross(p, q);
            }
            fixed4 frag(v2f i) : SV_Target
            {
                float3 p = float3(i.uv - float2(.5,.5), 1);
                float3 p0 = float3(_p0[0], _p0[1], _p0[2]);
                float3 p1 = float3(_p1[0], _p1[1], _p1[2]);
                float3 p2 = float3(_p2[0], _p2[1], _p2[2]);
                float3 p3 = float3(_p3[0], _p3[1], _p3[2]);
                float3 p4 = float3(_p4[0], _p4[1], _p4[2]);
                float3x3 M = solve(p0, p1, p2, p3, p4);
                float det = determinant(M);
                if (!isnan(det)) {
                    _BaseColor.a = conic(p, M);
                }
                //if (abs(det) > 1e-10) {
                //    // Inverse is dual conic.
                //    // In fact, the adjoint would be better
                //    _BaseColor.a = conic(p, inverse(M));
                //}
                float2 uv = i.uv;
                _BaseColor = lerp(_BaseColor, float4(1,0,0,_BaseColor.a),
                    punto(p, p0));
                _BaseColor = lerp(_BaseColor, float4(1,0,0,_BaseColor.a),
                    punto(p, p1));
                _BaseColor = lerp(_BaseColor, float4(0,1,0,_BaseColor.a),
                    punto(p, p2));
                _BaseColor = lerp(_BaseColor, float4(0,1,0,_BaseColor.a),
                    punto(p, p3));
                //_BaseColor = lerp(_BaseColor, float4(0,0,1,1), punto(p,
                    p4));
                return _BaseColor;
            }
            ENDCG
        }
    }
}
```

Imágenes

Arcos de circunferencia y sectores circulares

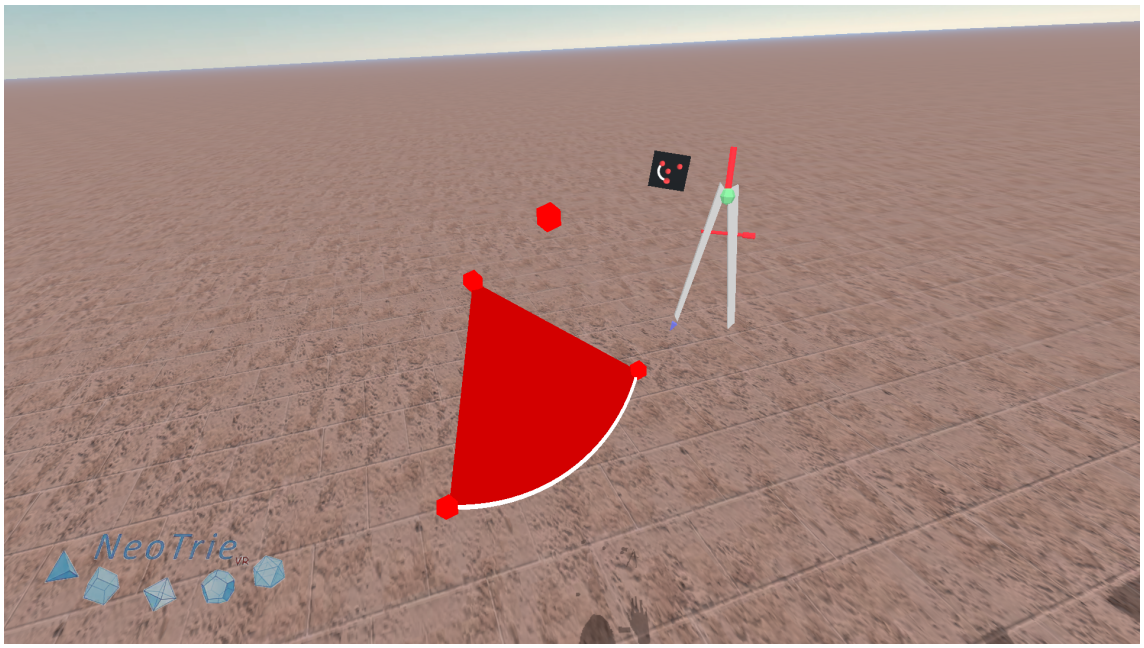

Figura 1: Arco de circunferencia y sector circular por 4 puntos.

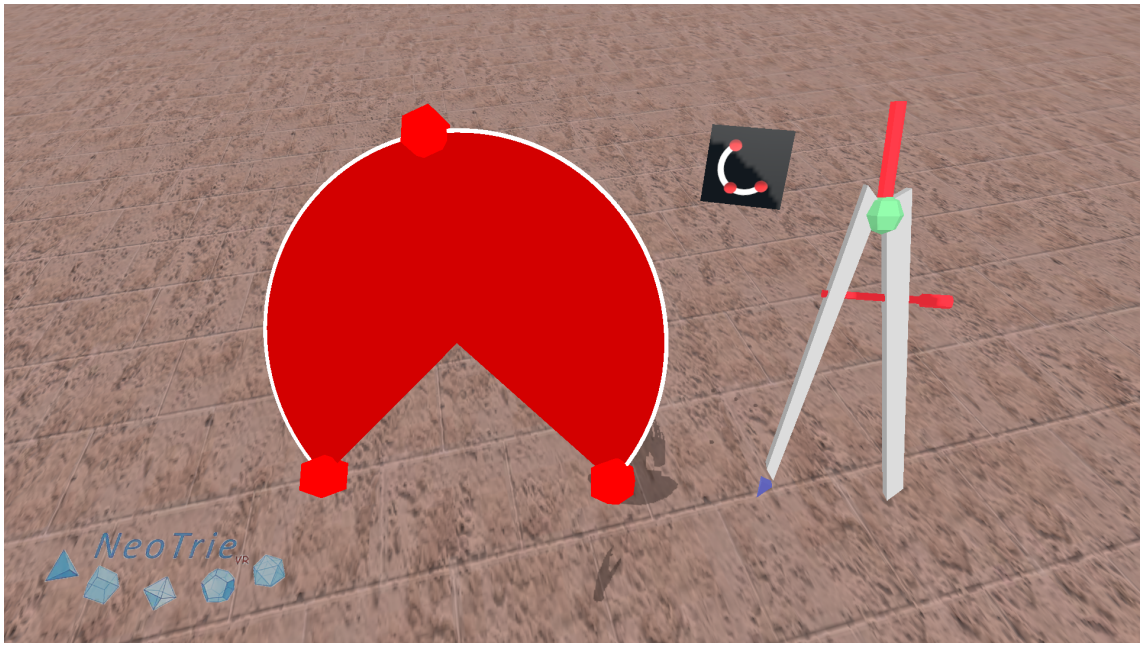

Figura 2: Arco de circunferencia y sector circular por 3 puntos.

Cónicas

Cónicas por definición

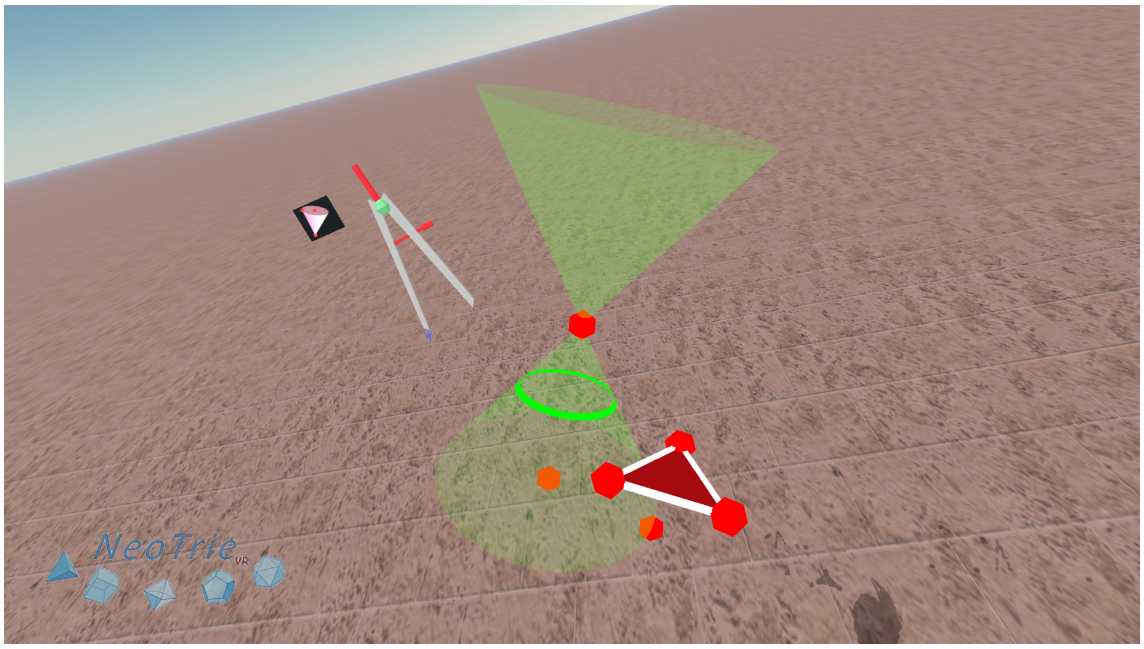

Figura 3: Cónica por definición, Circunferencia.

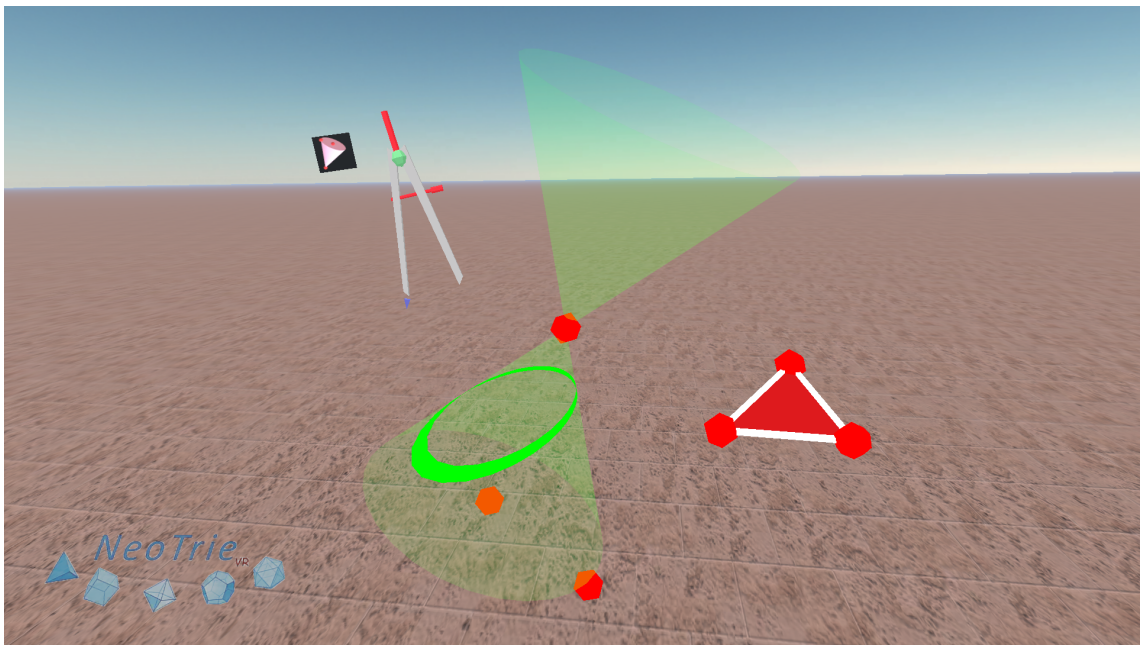

Figura 4: Cónica por definición, Elipse.

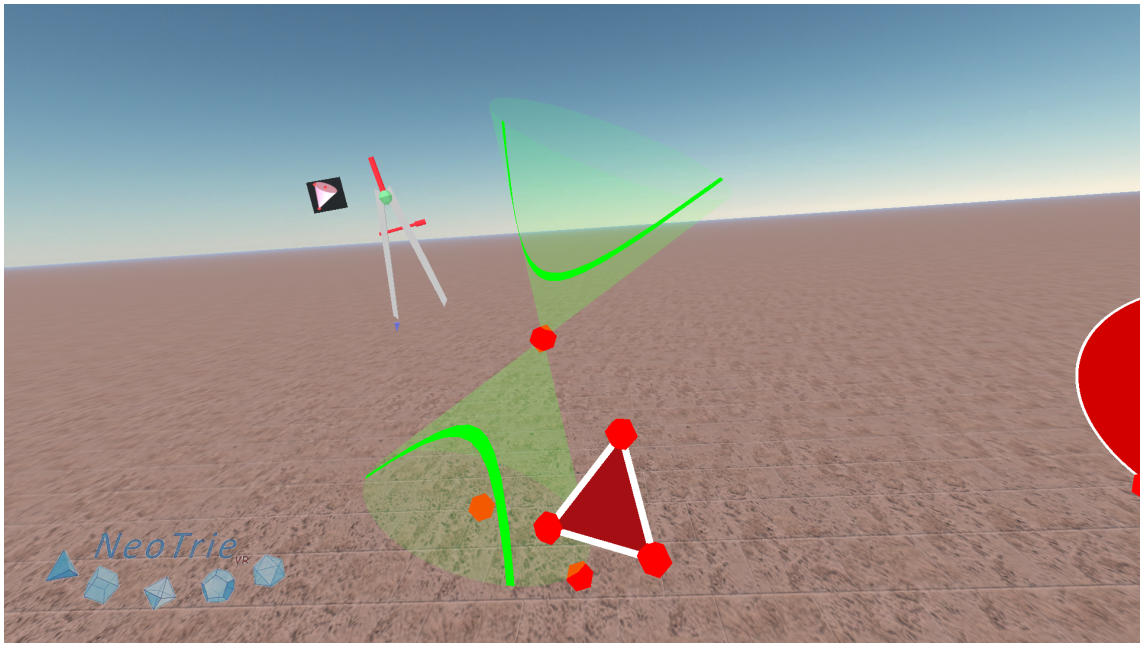

Figura 5: Cónica por definición, Hipérbola.

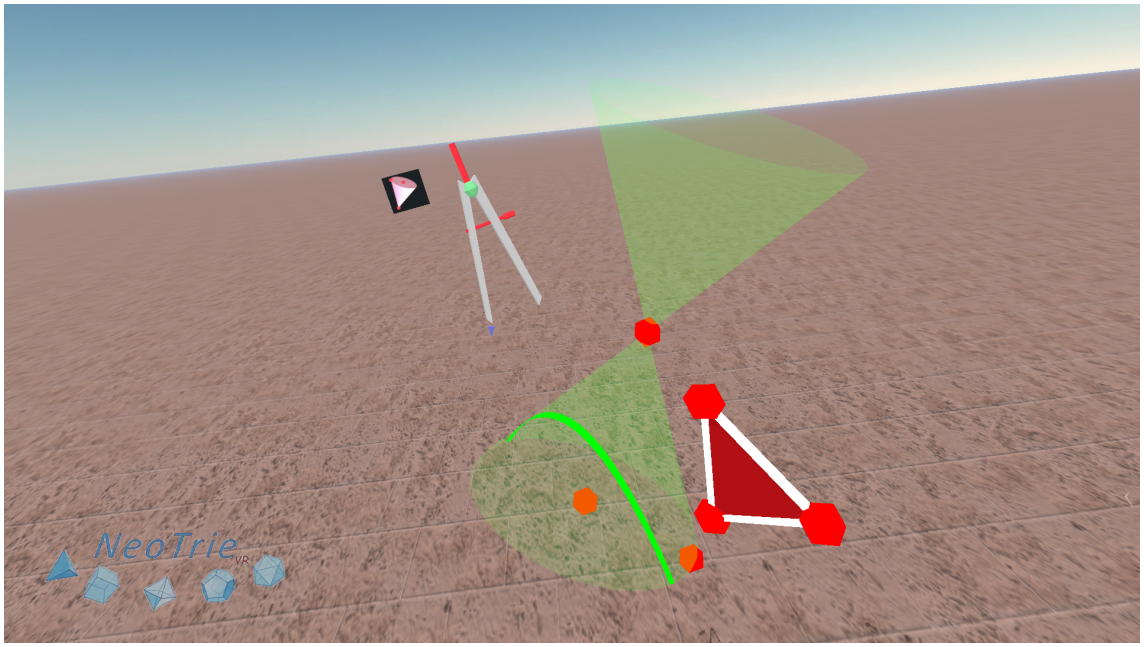

Figura 6: Cónica por definición, Parábola.

Cónicas: MathItem

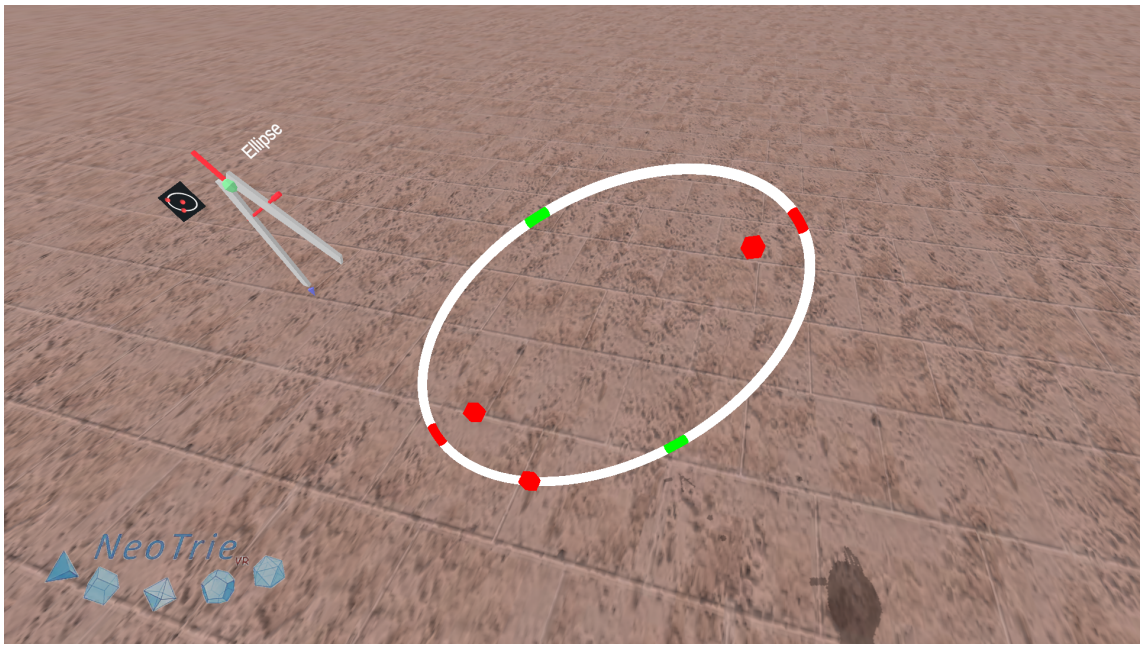

Figura 7: Elipse dada por focos y punto en esta.

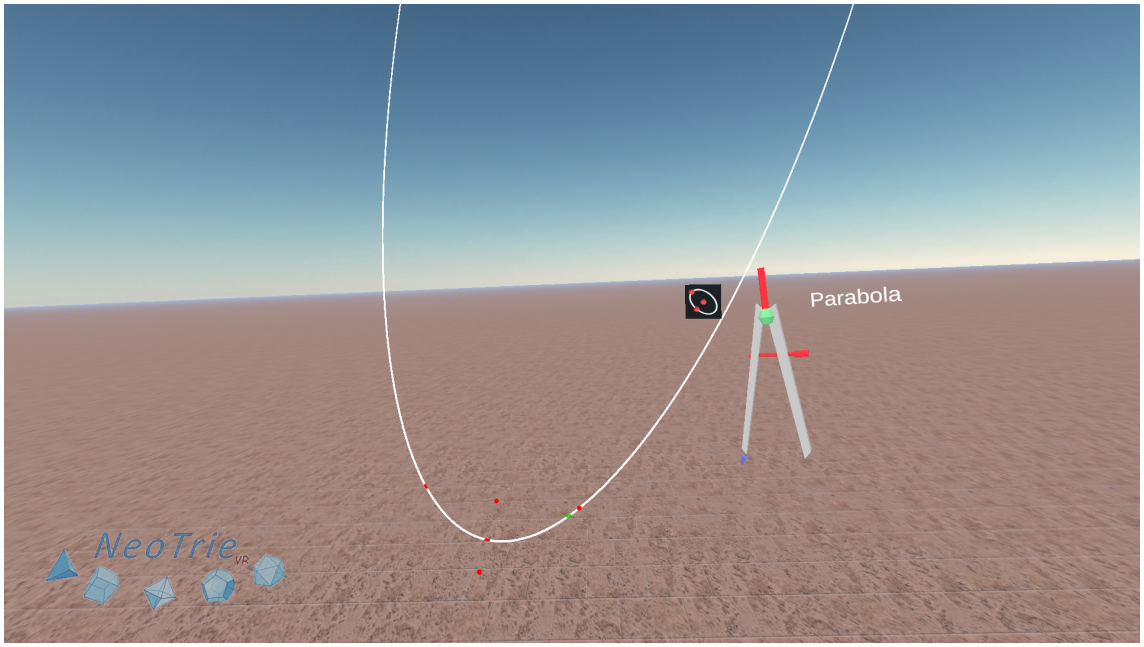

Figura 8: Parábola dada por foco, dirección perpendicular a recta directriz y punto en esta.

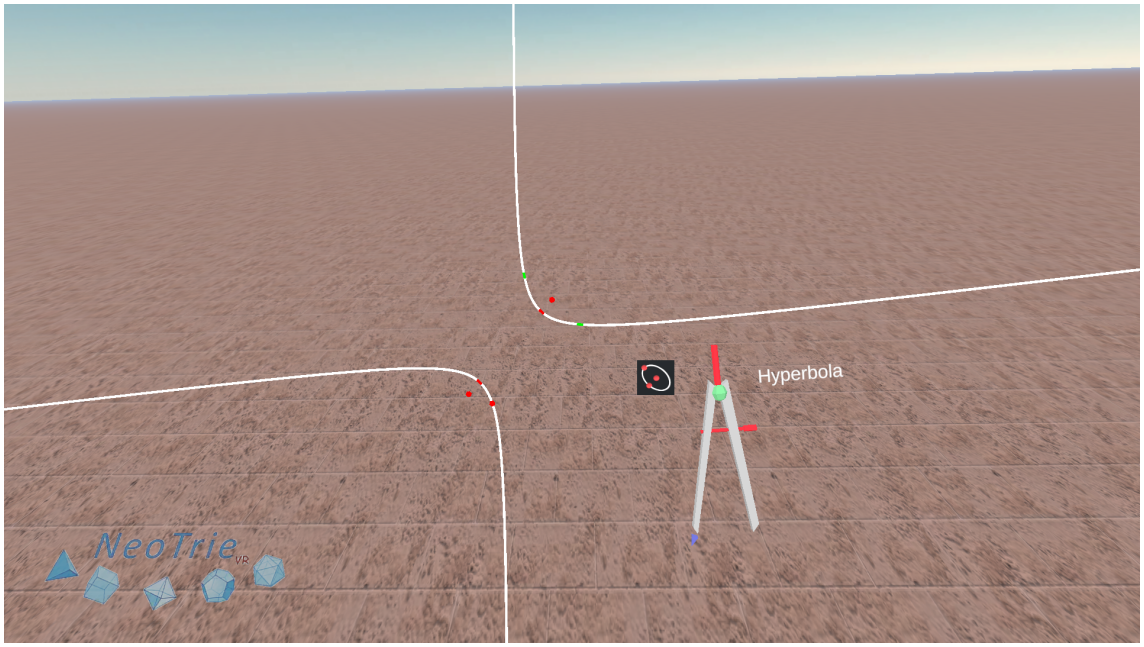

Figura 9: Hipérbola dada por focos y punto en esta.

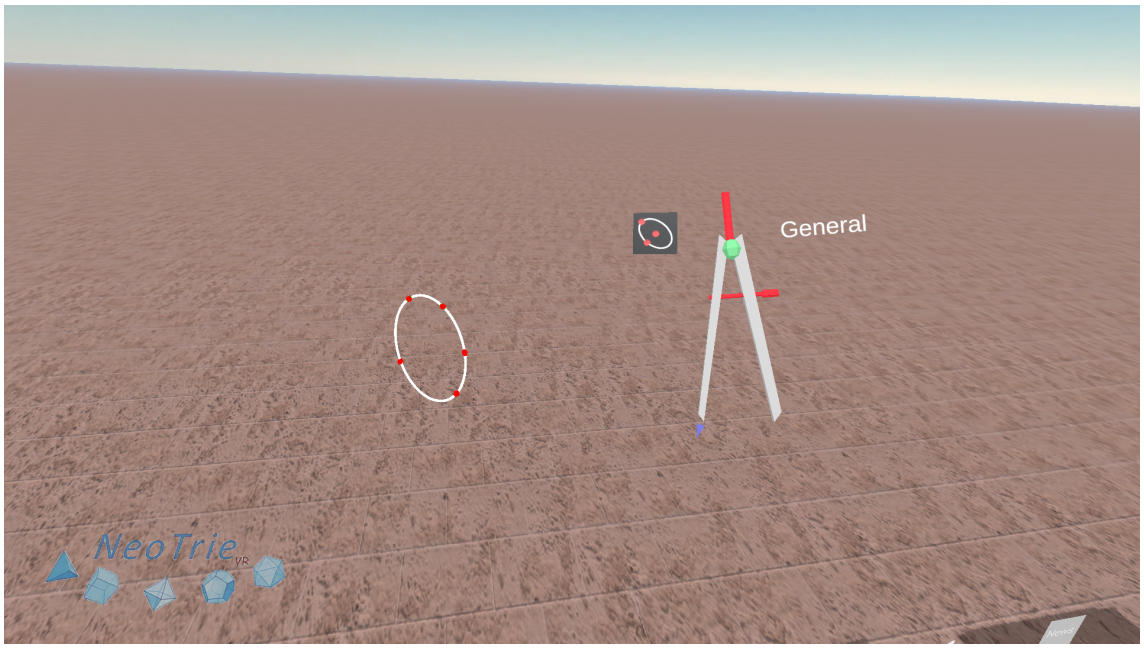

Figura 10: Cónica por 5 puntos, Elipse.

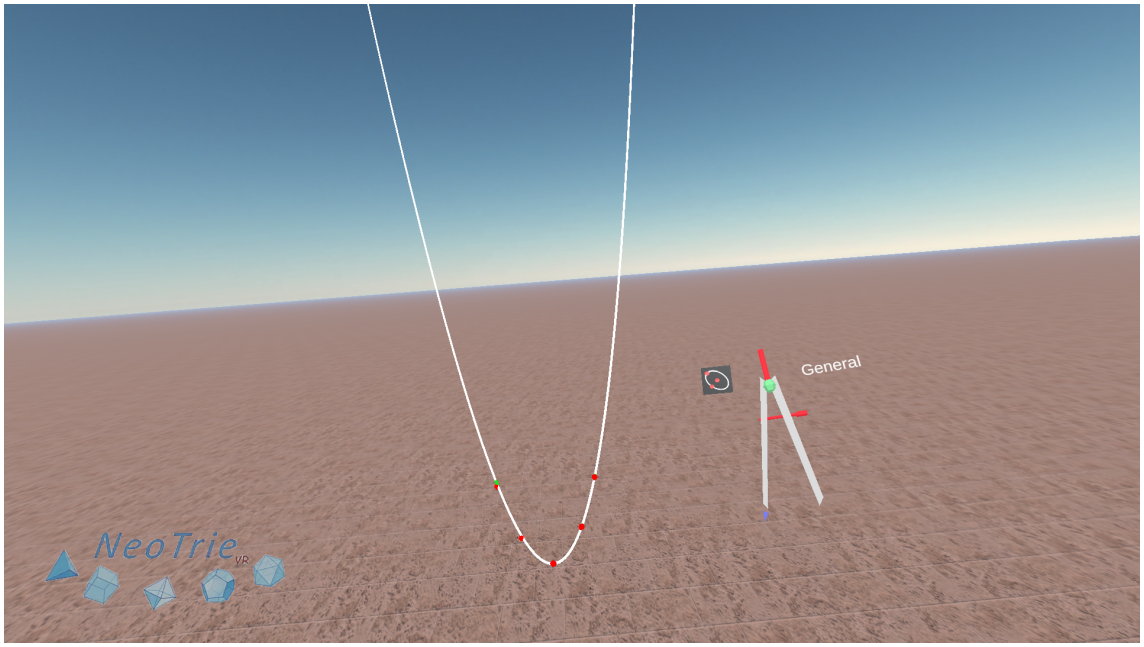

Figura 11: Cónica por 5 puntos, Parábola.

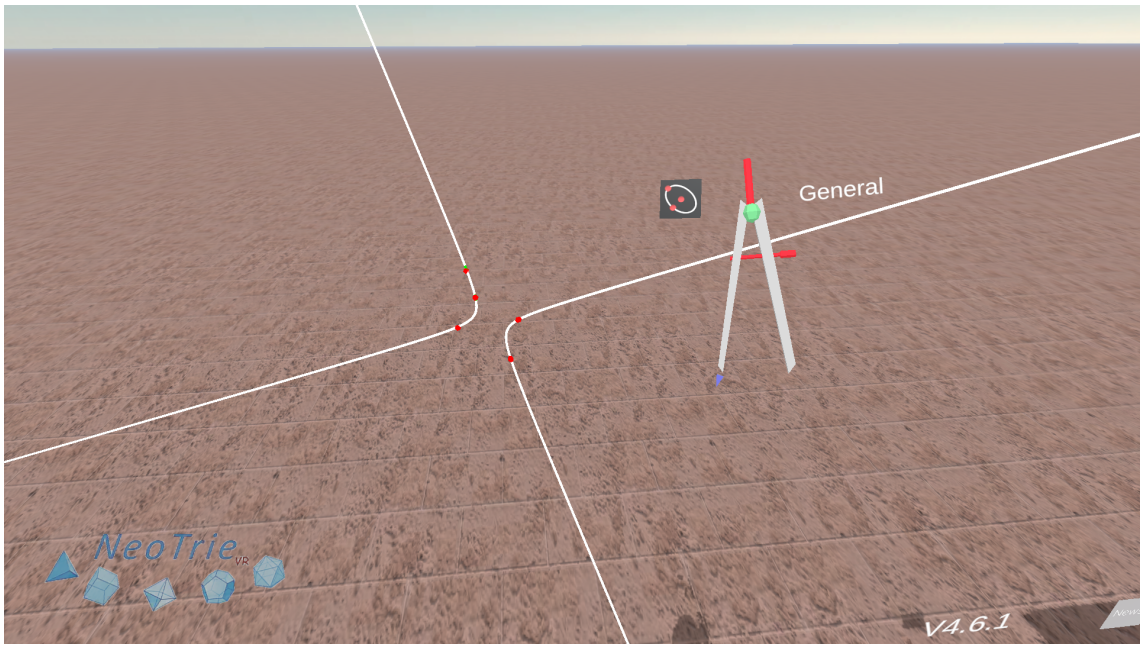

Figura 12: Cónica por 5 puntos, Hipérbola.

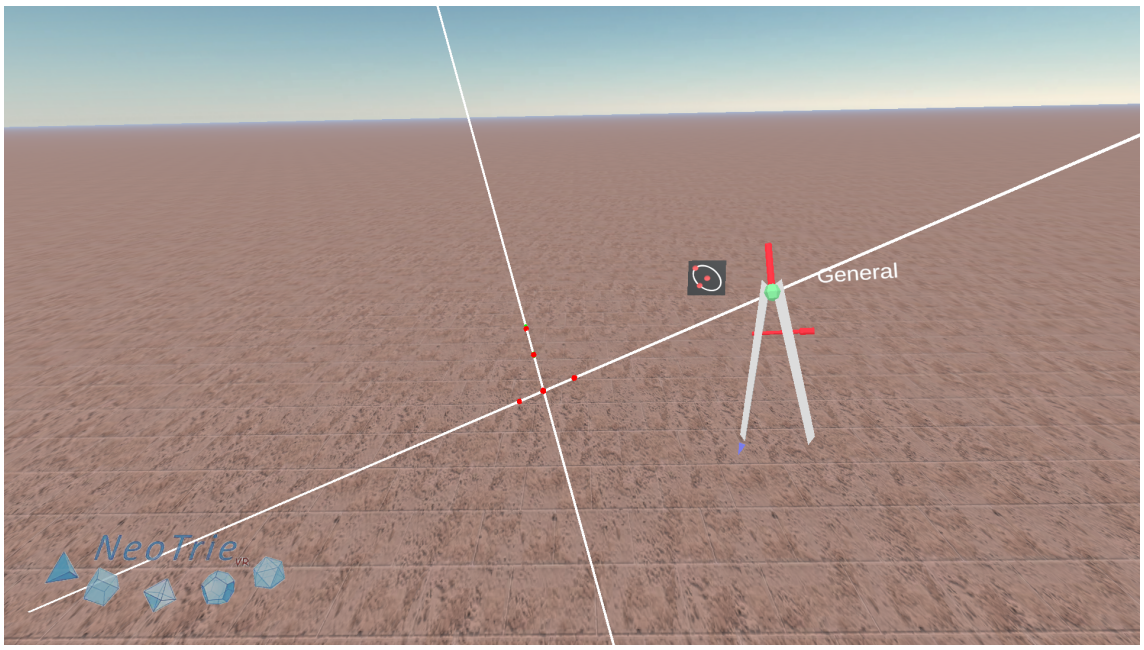

Figura 13: Cónica por 5 puntos, Par de rectas.