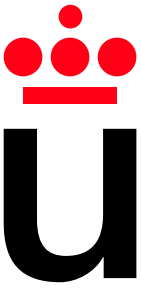


Universidad
Rey Juan Carlos

Escuela Técnica Superior
de Ingeniería Informática

Grado en Ingeniería Informática

Curso 2021-2022

Trabajo Fin de Grado

VIRTO: ENTORNO DE PROGRAMACIÓN EN REALIDAD VIRTUAL

Autor: Julián Sánchez Fernández
Tutor: Micael Gallego Carrillo
Cotutor: Jesús M. González Barahona

Agradecimientos

Cuando se consigue algo, se suele premiar y alabar a aquellos que han dado la cara, pero se tiende a olvidar el enorme equipo que tienen detrás y sin el cual, si no imposible, les habría resultado mucho más difícil alcanzar sus objetivos. Por ello, quiero agradecer a mi gran equipo todo su apoyo y ayuda.

A mi familia, por su inagotable capacidad de escucha, comprensión y paciencia, que me han permitido levantarme cada vez que me he caído.

A mis amigos, que han estado conmigo en todo momento, ayudándome en los malos momentos y celebrando los nuevos.

A los profesores con los que he coincidido durante todas las etapas educativas, cuya vocación y afán por enseñar, me han ayudado a llegar a donde estoy ahora.

Gracias a todos.

Resumen

Este trabajo tiene por objeto explorar las posibilidades que ofrece la realidad virtual para crear un entorno de desarrollo sencillo y extensible que permita la realización de programas destinados al control de un móvil. Para ello, se procederá a la creación de prototipos de manera incremental, con el objetivo de desarrollar una interfaz persona-ordenador que permita realizar las tareas propias de un entorno de desarrollo sin replicar el teclado y ratón.

Como resultado, se ha desarrollado un producto mínimo viable llamado `ViRto`, pensado para ser ejecutado en un visor de realidad virtual a través de un navegador web. `ViRto` permite a los usuarios crear programas para mover y rotar un *drone* a través de programas construidos con secuencias, bucles y decisiones. Adicionalmente, se ha creado un blog[1] en el que se ha ido escribiendo acerca del proceso de desarrollo, incluyendo muestras de los resultados que se iban obteniendo.

Para desarrollar `ViRto`, se han empleado tecnologías *OpenSource* que posibilitan la creación de escenas de realidad virtual en el navegador, como *A-Frame*, *Three.js* o *Ammo.js*. El código de este trabajo y el del blog asociado son públicos y accesibles a través del repositorio creado para llevar a cabo un control de versiones, de manera que otras personas se puedan beneficiar de ellos.

Palabras clave: Realidad Virtual, Programación, A-Frame, JavaScript, WebXR

[1] `https://jdjuli.github.io/virto/`

Índice de contenidos

Índice de figuras

Índice de códigos

1 Introducción

Desde la aparición de los primeros computadores, hemos tenido que aprender a instruirlos en la tarea que queremos que realicen. En este capítulo se presenta un breve repaso de las herramientas de programación gráfica que se han ido creando a lo largo del tiempo, así como los objetivos de este trabajo.

1.1. Contexto

Inicialmente la programación se realizaba modificando las conexiones eléctricas de los computadores o de forma menos invasiva, haciendo uso de tarjetas perforadas. Según han ido avanzando los computadores y surgiendo nuevos periféricos, la forma de programar ha ido cambiando, apareciendo los editores de texto, en los que inicialmente solo se podía usar el teclado y posteriormente, también el ratón, lo cual demostró Douglas Engelbart en 1968 durante una demostración que ha pasado a la historia como '*la madre de todas las demostraciones*'[1][1].

Especialmente en el siglo XXI, se puede ver como ha ido ganando fuerza la programación gráfica, que permite programar sustituyendo parte de los elementos de los lenguajes de programación por figuras geométricas que se pueden componer como si de un puzle se tratara para crear programas funcionales. Aunque esta idea se remonta a 1991, cuando Microsoft lanzó su producto *Microsoft Visual Basic*[2], el cual dividía la programación de aplicaciones gráficas en dos fases, en la primera,

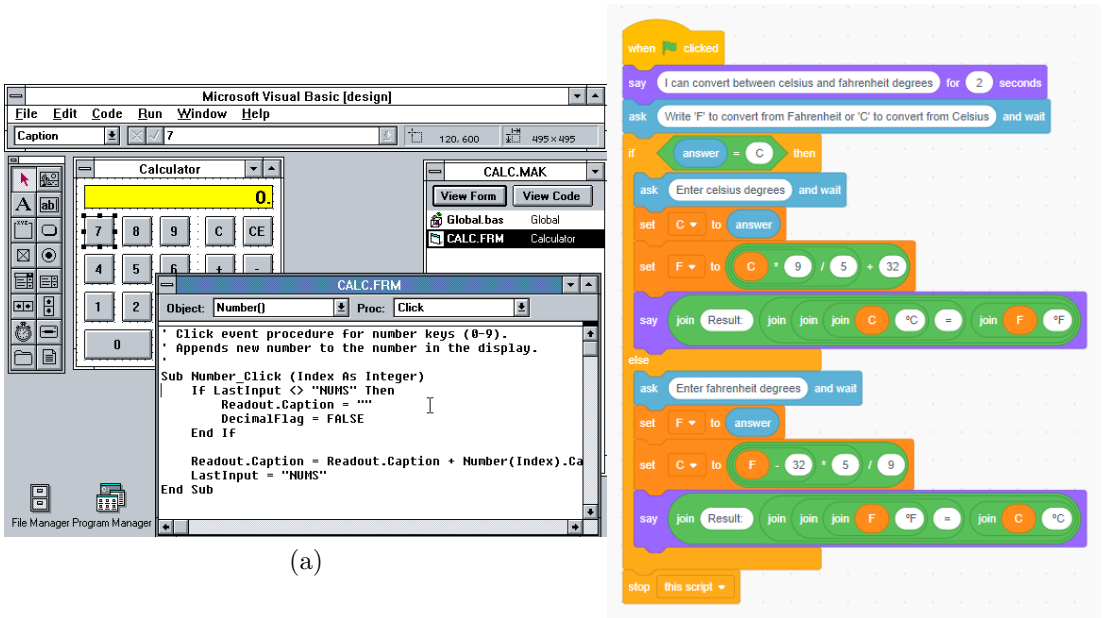
[1] '*The mother of all demos*'

[2] https://winworldpc.com/product/microsoft-visual-bas/10

se construye la interfaz a través de un editor *WYSIWYG*[3] y, posteriormente, se añade código para definir su comportamiento.

Más adelante, en 2003 el MIT comenzó a trabajar en '*Scratch*'[2], un lenguaje de programación visual diseñado para enseñar a programar, por lo que aunque no se pueden diseñar interfaces de usuario como en *Visual Basic*, a diferencia del producto de *Microsoft*, la lógica del programa se define de manera completamente gráfica, eliminando la necesidad de escribir código y por tanto, conocer un lenguaje de programación para poder utilizar el entorno.

En la figura 1.1 se puede ver a la izquierda una captura de pantalla de *Microsoft Visual Basic 1.0*, en la cual se está construyendo una calculadora y se puede apreciar cómo el comportamiento del programa se define mediante código y, a la derecha, un programa creado con *Scratch*, cuyo comportamiento se ha definido uniendo bloques de diferentes formas y colores como si fuera un puzle.



(a) (b)

Figura 1.1: Capturas de pantalla de la creación de programas usando (a) *Microsoft Visual Basic 1.0* y (b) *Scratch*

En los últimos años se está viviendo una creciente popularización de los dispositivos de realidad virtual, principalmente gracias a los videojuegos, pero que, teniendo en cuenta lo expuesto anteriormente, nos plantea algunas preguntas como si esta tecnología dará pie a una nueva forma de programar y, en caso de que lo haga, ¿Cómo serán los programas que lo permitirán?.

[3] '*What You See Is What You Get*', que se traduce como '*Lo que ves es lo que obtienes*'

1.2. Objetivos

Este trabajo tiene por objetivo explorar cómo se podría programar en realidad virtual a través de un lenguaje visual sencillo, construyendo para ello una interfaz inspirada en el entorno *Scratch* para obtener un producto mínimo viable que permita evaluar su utilidad y usabilidad a la hora de construir programas.

Por tanto, se plantea la construcción de un prototipo de entorno de programación en realidad virtual que permita crear programas destinados a mover un móvil por la escena, haciendo uso del paradigma de programación estructurado. A continuación se enumeran los objetivos específicos del trabajo:

1. Crear un intérprete que ejecute los programas creados, actuando sobre el móvil de acuerdo a aquello que se haya indicado en ellos.
2. Explorar formas de utilizar la realidad virtual para desarrollar entornos de programación escalables y extensibles.
3. Construir una aplicación de realidad virtual utilizando tecnologías web como *Three.js* y *A-Frame*
4. Establecer una relación directa entre la estructura de los programas creados y cómo se representan en el DOM, de manera que éstos puedan ser serializados.
5. Crear un lenguaje sencillo para programar que incorpore los tres tipos de instrucciones de la programación estructurada: secuencia, decisión y repetición.
6. Documentar el proceso de desarrollo software realizado a través de un blog.
7. Diseñar y realizar un experimento que permita evaluar de forma cualitativa el prototipo final.

2

Estado del arte y tecnologías relacionadas

En este capítulo se hará un breve repaso por las herramientas y programas previos que han explorado el uso de los gráficos por ordenador como vía de comunicación entre el computador y el usuario, así como las tecnologías relacionadas con este trabajo y los componentes *A-Frame* creados por la comunidad y utilizados en este trabajo.

2.1. Lenguajes de programación gráficos

Esta sección hace un repaso cronológico por los proyectos que previamente han explorado el uso de los gráficos por ordenador para enseñar, facilitar tareas visuales o crear lenguajes de programación visuales.

2.1.1. Sketchpad

El sistema *Sketchpad* se creó en 1963 como una nueva forma de interfaz persona-ordenador para crear gráficos vectoriales. Se apoyaba en el uso de un lápiz luminoso para trazar líneas que el programa procesaba y convertía en figuras geométricas como rectas, circunferencias o polígonos, sobre los cuales se podían establecer restricciones.

Sketchpad no se limitó a ser un editor de gráficos vectoriales, sino que con los elementos necesarios, era posible incluso diseñar circuitos eléctricos y evaluar

dicho diseño en un software simulador, a modo de ejemplo, en la figura 2.1 se puede ver a Ivan Sutherland, su creador, haciendo una demostración de su sistema en la que se aprecia un diseño técnico en la pantalla, lo que recuerda a los software CAD actuales (*Computer Aided Design*)

Sus creadores acabaron concluyendo que su software solamente sería de utilidad cuando la información obtenida a través del ordenador fuera más allá de los gráficos creados, como ocurriría en el ejemplo del diseño y simulación de circuitos eléctricos. [3]. Casi al mismo tiempo, se presentó una nueva versión de *Sketchpad* llamada *Sketchpad III* que permitía la creación y manipulación de primitivas tridimensionales de la misma manera [4].

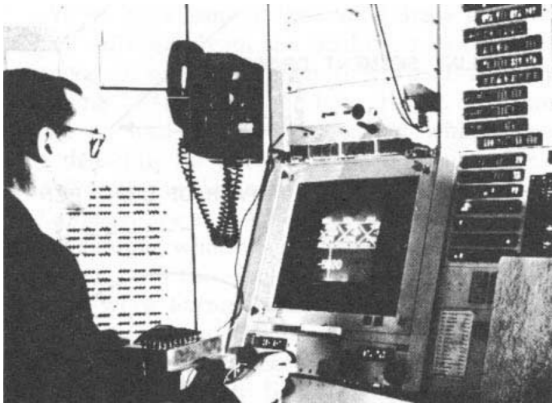


Figura 2.1: Ivan Sutherland realizando una demostración del sistema Sketchpad (1963)

Fuente: Artículo [3], figura 1

2.1.2. Logo y Turtle graphics

Logo es un lenguaje de programación que se empezó a diseñar en 1966 como una versión simplificada de LISP, con el propósito de enseñar matemáticas a través de la programación a niños.

Uno de los componentes del entorno de Logo es el llamado *turtle graphics*, que consiste en un objeto gráfico en pantalla que tiene tres atributos: posición, orientación y 'lápiz' (o *pen* en inglés), el cual tiene como mínimo un atributo que indica si está levantado o no, además de otros muchos opcionales como pueden ser el grosor, el color o el estilo de línea que trazará. El valor de *turtle graphics* reside en la posibilidad de trazar figuras geométricas arbitrariamente complejas mediante la modificación de los tres atributos del objeto gráfico [5]. Posteriormente, se crearon robots capaces de realizar las acciones descritas en un programa Logo a los cuales se llamó *'tortugas'* y son la razón por la que la representación gráfica del objeto gráfico que pinta en la pantalla suela ser una imagen de una tortuga.

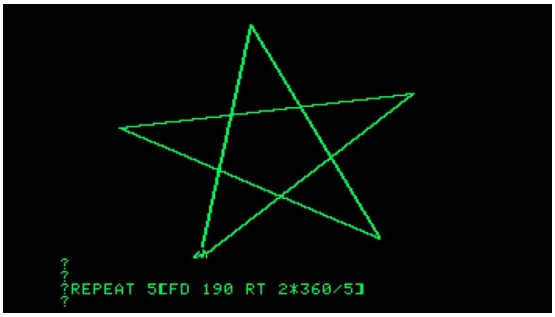


Figura 2.2: Secuencia de instrucciones LOGO para dibujar una estrella de cinco puntas usando el componente *turtle graphics*

2.1.3. CUBE

CUBE[6] es un lenguaje de programación visual propuesto en 1991, que hace uso de gráficos tridimensionales para representar los programas, tiene un tipado estático y su semántica se basa en la lógica de Horn, tratando los predicados como elementos de primera clase y con ello permitiendo que estos puedan ser pasados a otros predicados de igual manera que un tipo simple (es similar al lenguaje JavaScript actual, en el cual las funciones pueden ser pasadas como parámetro a otras funciones).

Este lenguaje aspiró a ser utilizado para programar en entornos de realidad virtual, de hecho, en 1992 los mismos autores publicaron otro artículo[7] en el que mostraban el resultado de su implementación de un IDE para validar y ejecutar programas escritos en *CUBE*, de dicho artículo se ha extraído la figura 2.3, que muestra el resultado de renderizar la función factorial junto con el resultado de su ejecución para el valor de entrada '3'.

En este último artículo incidieron en la necesidad de crear un editor para los programas y la idoneidad de la realidad virtual sobre el ratón convencional como forma de interactuar con dicho editor.

2.1.4. Alice

Alice es un sistema de prototipado rápido presentado en 1994 e ideado para crear escenas dinámicas con objetos tridimensionales. Su rapidez reside en que hace uso del lenguaje interpretado *Python* para establecer cómo alteran la escena las interacciones del usuario.

Utiliza un lenguaje visual basado en la composición de figuras geométricas y orientado a prototipos (similar a la POO pero sin la posibilidad de crear clases), de forma que cada elemento de la escena tiene una entidad asociada que pertenece a una jerarquía de entidades cuya raíz es la escena, denominada *this*. En la figura

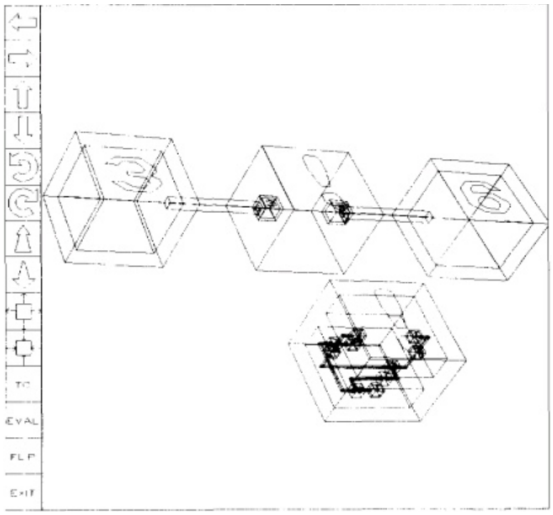


Figura 2.3: Renderizado de la operación matemática *factorial* implementada en CUBE (abajo) y resultado de realizar dicha operación sobre el número entero 3 (arriba), cuyo resultado es 6 como puede verse en el cubo superior derecho
Fuente: Artículo [7], figura 4

2.4 se puede apreciar la interfaz de *Alice 3.6*, en la cual, al no haber objetos tridimensionales en la escena, el visor está vacío y solo se pueden implementar funciones y procedimientos de *'this'*

Los creadores indican que pese a la lentitud de *Python* frente a otros lenguajes compilados como *C* o *C++*, se obtiene un aumento de rendimiento importante al evitar la compilación, que es especialmente notable cuando se hacen constantemente pequeños cambios en el prototipo y por tanto, en el programa que genera dicha escena.

Desde su inicio, este sistema ha ido evolucionando, enfocándose en el ámbito de la educación y añadiendo la posibilidad de crear proyectos de realidad virtual, asociando la cámara a la posición del visor y permitiendo usar los controladores en caso de haberlos. Si el programador necesita obtener información del visor o los controladores, puede acceder a ella de la misma manera que si se manipulara un objeto 3D, ya que también cuentan con su propia entidad dentro de la escena.

2.1.5. Scratch

El proyecto *Scratch* se inició en 2003[2] y se hizo público en 2007, siendo un entorno de programación con el objetivo de que los niños y adolescentes (especialmente los pertenecientes a comunidades socialmente desfavorecidas) puedieran desenvolverse mejor con la tecnología mediante su uso, a la vez que desarrollan otras competencias como son la resolución de problemas, el pensamiento ma-

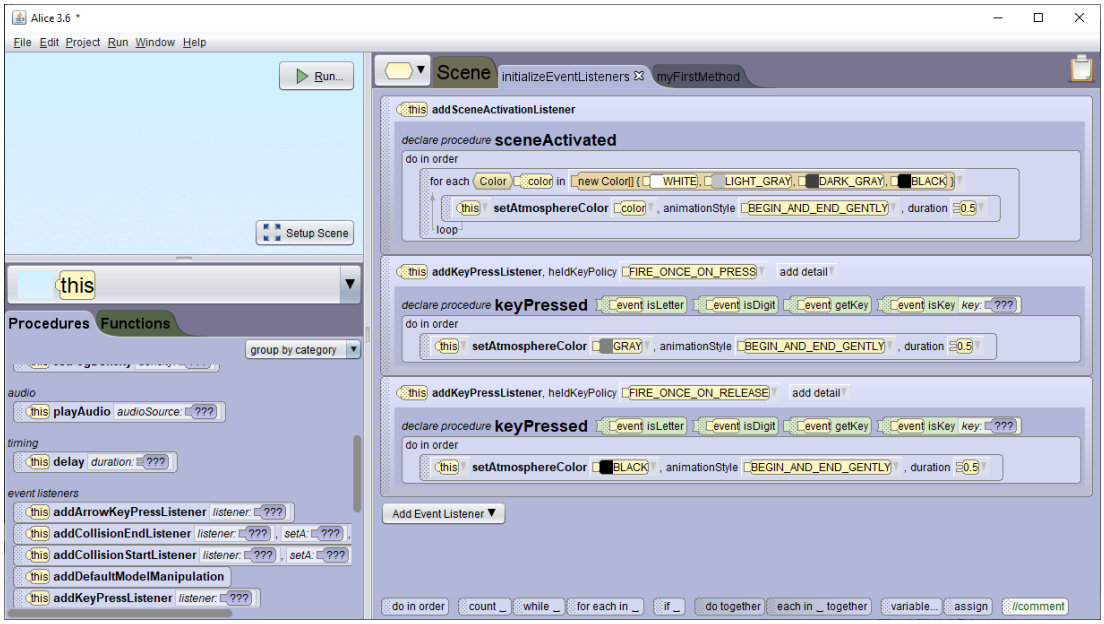


Figura 2.4: Captura de pantalla del software *Alice 3.6*. En la parte izquierda se aprecia una previsualización de la escena, un selector de objetos (*'this'* hace referencia a la escena) y un catálogo de procedimientos y funciones que pueden ser implementadas y, a la derecha, un panel destinado a programar mediante un lenguaje basado en bloques, la lógica de los objetos
Fuente: Elaboración propia

temático y computacional o la creatividad, mediante la creación de programas arbitrarios que tienen como resultado gráficos bidimensionales que representan aquellas historias que los niños quieran plasmar y animar.

Scratch tiene una forma particular de gestionar los errores, ya que suprime los errores sintácticos al forzar a que las piezas solo puedan encajarse en los espacios con su misma forma (como si fuera un puzle) y resuelve los errores semánticos de forma 'amigable', ajustando los valores fuera de rango a los límites del mismo y, en caso de realizar operaciones '*imposibles*' como una raíz cuadrada de un número negativo o una división entera entre 0, devuelve *cero* o *NaN* para hacer notar el error al usuario sin recurrir a mostrarle un mensaje que quizás no entienda, ya que el público objetivo de *Scratch* generalmente no tiene conocimientos previos de informática y/o programación [8]. La figura 2.5 muestra como resuelve *Scratch* la expresión $\frac{\sqrt{-5} \text{ mód } 0}{0} \cdot 42$, arrojando 0 como resultado e informándonos de que las variables a, b y c tienen un valor no representable numéricamente (NaN)

2.1.6. Snap!

Snap!, también conocido como *BYOB (Build Your Own Blocks)*, es la continuación del proyecto *Scratch*, extendiendo su lenguaje para que también se

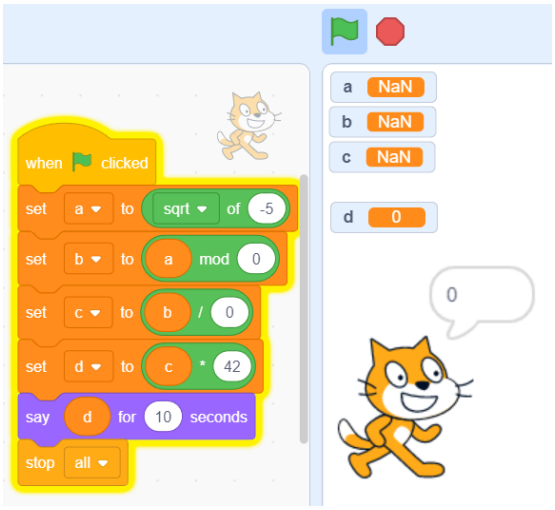


Figura 2.5: Captura de pantalla que muestra un programa en *Scratch* con errores aritméticos y de tipos ejecutándose, se aprecia que no se notifica ninguno de estos errores, pero la ejecución finaliza con el valor 0 asignado a la variable D.
Fuente: Elaboración propia

puedan crear programas complejos manteniendo el mismo estilo de programación y ejecución. Con ello, ampliaron el abanico de usuarios para el cual *Scratch* fue diseñado [9], convirtiendo las listas, sprites (imágenes) y procedimientos en elementos de 'primera clase', convirtiéndolo en un lenguaje con el cual, mediante prototipos, se logra una aproximación a la programación orientada a objetos. La funcionalidad es idéntica a la de *Scratch*, pero ofrece más *sprites*, a los cuales se pueden asociar diferentes comportamientos y hacer que interactúen entre sí mediante paso de mensajes. En la figura 2.6, se muestra un ejemplo de los elementos a disposición del usuario para que éste cree sus propios bloques y con ello estructurar mejor el código, el programa de la figura hace uso del bloque '*drawCircle*', el cual, al estar parametrizado hace posible dibujar círculos de diferentes radios sin tener código duplicado.

2.1.7. VR-ocks

VR-ocks es un prototipo de aplicación en realidad virtual destinada a la enseñanza de los principios más básicos de la programación[10]. El nombre proviende de *VR-Blocks*, como referencia a la abstracción usada por sus creadores para representar los elementos del lenguaje, bloques tridimensionales, los cuales debe emplear el usuario para crear programas que permitan resolver una serie de puzles o retos.

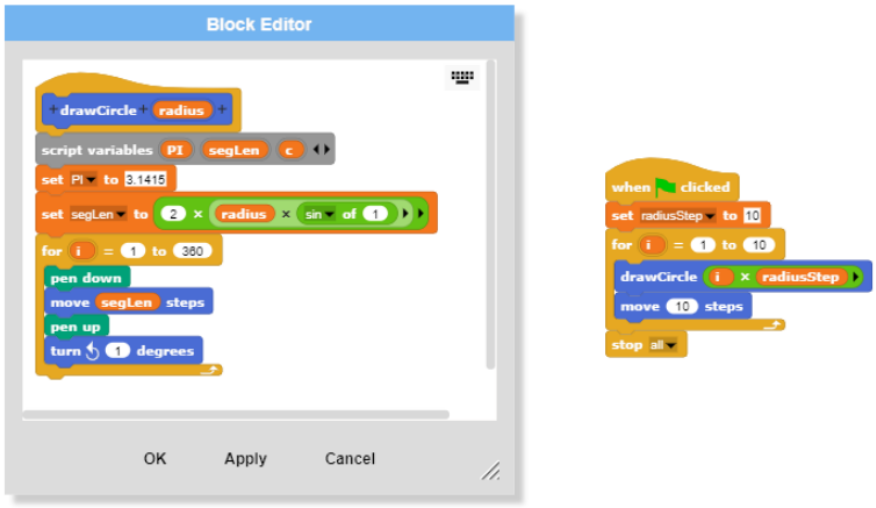


Figura 2.6: Captura de pantalla del entorno *Snap!* en el que se puede ver a la derecha un pequeño programa que hace uso de una función no predefinida en el lenguaje (*drawCircle(radius)*), cuya definición se encuentra a la izquierda dentro del '*Block Editor*'.

Fuente: Elaboración propia

Este prototipo aborda la creación de un entorno de programación en realidad virtual desde la gamificación, como puede apreciarse en la figura 2.7, desarrollándolo en *Unity*[1], y presentando al usuario una serie de actividades con una complejidad creciente. El entorno en el que se desarrollan las actividades, consiste en un espacio en el que hay un personaje humanoide que obedeciendo a las instrucciones del programa del usuario, debe cumplir un determinado objetivo.

[1] `https://unity.com/`

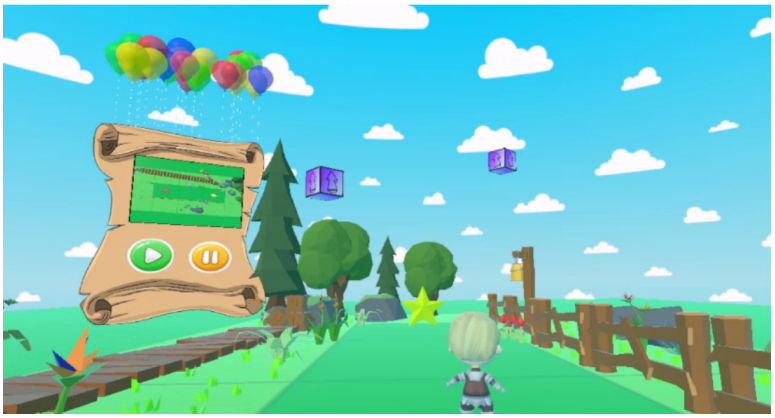


Figura 2.7: Captura de pantalla de *VR-ocks*, en ella se aprecia el personaje humanoide que el usuario debe controlar junto a un panel previsualizando el puzle y proporcionando de dos botones para ejecutar y pausar el programa. Los bloques que deben usarse (morados), se encuentran flotando en el aire.
Fuente: artículo [10], figura 1

2.2. Tecnologías relacionadas

En esta sección se hará un repaso por las tecnologías necesarias para la consecución de este trabajo, empezando por las tecnologías presentes en la gran mayoría de navegadores modernos y continuando con las librerías JavaScript que hacen posible la creación de escenas de realidad virtual en el navegador.

2.2.1. HTML5

HTML5 es la quinta revisión de la especificación del lenguaje de marcas HTML, que se empezó a crear en 2007 y aún se continúa trabajando en ella.

Su principal objetivo es dar soporte a todos aquellos elementos que se fueron añadiendo a las webs en la década de los 2000, como animaciones, sonidos o vídeos incrustados en la página y que, para poder reproducirse requerían de la instalación de plug-ins como Adobe Flash Player o Microsoft Silverlight entre otros, lo cual consigue proveyendo a los programadores de un conjunto de APIs para dar mejor soporte al contenido multimedia, como son WebGL, WebXR o WebAssembly entre muchos otros, eliminando con ello la necesidad de utilizar *plug-ins* de terceros y ampliando las capacidades de las páginas web.

También añadió nuevos elementos como 'section', 'article', 'header' o 'nav' para ser utilizados junto con las etiquetas 'div' y dotar de una mayor semántica al código.

2.2.2. JavaScript

JavaScript es un lenguaje de scripting creado en 1995 y llamado originalmente Mocha, el cual se ejecuta en los navegadores web para dotar a las páginas web de dinamismo y capacidad de reacción ante eventos tales como clics, movimientos del ratón o pulsaciones de teclado.

Está estandarizado bajo el nombre *ECMAScript*, cuya primera versión se publicó en junio de 1997[11], cabe destacar que, en el año 2006, la sexta revisión de este estándar, introdujo *AJAX*[2], una técnica que permite realizar de peticiones HTTP(S) de manera asíncrona (mediante promesas) y el manejo de las respuestas obtenidas sin necesidad de recargar la web original, lo que supuso que se pudiera actualizar su información sin necesidad de recargar todos los recursos de la misma. A partir de 2015, las versiones de *ECMAScript* dejaron de numerarse y se empezaron a nombrar también con el año en que se publicaba una nueva edición del estándar, siendo la última disponible en el momento de la escritura de este documento, *ECMAScript2022*[3] (ES13)

Inicialmente, JavaScript solo se empleaba en el lado del cliente (siendo ejecutado por los navegadores web), sirviendo de nexo de unión entre el servidor y la interfaz de usuario creada con HTML y estilizada con CSS, pero posteriormente, se empezó a usar también en el lado del servidor (back-end), siendo el ejemplo más notable *Node.js*.

2.2.3. WebGL

WebGL es un estándar web open-source y multiplataforma para la creación de gráficos en 3D en los navegadores exponiendo las APIs OpenGL en el navegador y permitiendo que los elementos gráficos creados con él se puedan beneficiar de la aceleración por hardware en los casos en que esté disponible (tarjetas gráficas).

Este estándar ha sido adoptado por los principales navegadores web del mercado y puede ser utilizado a con una semántica similar a OpenGL ES, ligeramente modificada para adaptarla a las características de JavaScript.

WebGL puede integrarse con HTML5 para crear todo tipo de gráficos en la web con un rendimiento similar al de una aplicación nativa, es por ello que han surgido proyectos que abstraen el bajo nivel de WebGL para facilitar a los programadores la creación de contenidos tanto 2D como 3D.

[2] Asynchronous JavaScript + XML

[3] `https://www.ecma-international.org/wp-content/uploads/ECMA-262_13th_edition_june_2022.pdf`

2.2.4. WebAssembly

WebAssembly, también conocido como WASM, es un formato de instrucciones binario diseñado para ser ejecutado en máquinas virtuales con una arquitectura basada en pila.

Es posible compilar código C++ entre otros, a este formato, permitiendo que el binario se pueda ejecutar en múltiples plataformas, de forma similar a como lo hace Java al compilar a su propio bytecode, 'compila una vez, ejecuta en cualquier parte'. Este formato hace uso de las características hardware comunes a un amplio rango de arquitecturas de computadores, logrando así un rendimiento cercano al de la ejecución de código nativo.

El hecho de ser ejecutado en máquinas virtuales ha facilitado su inclusión en los navegadores, en los cuales se pueden interconectar binarios WebAssembly con JavaScript, disfrutando así de la velocidad de ejecución del primero sin renunciar a las características de alto nivel que ofrece el segundo.

2.2.5. Three.js

Three.js es una librería open-source para el lenguaje JavaScript que abstrae WebGL facilitando la tarea de visualizar gráficos 2D y 3D en el navegador. Para usar la librería, además de conocimientos de JavaScript y HTML, es necesario tener nociones sobre gráficos generados por computador para poder aprovechar todo el potencial que nos ofrece. El componente principal es la 'escena', que actúa como un contenedor de elementos que posteriormente se renderizarán y mostrarán por pantalla. A dicha escena se deben incorportar los objetos que se quieran visualizar junto a sus correspondientes materiales y al menos una cámara que nos permita capturar la escena teniendo en cuenta sus parámetros (distancia focal, apertura, etc.). En la página *https://threejs.org/examples/* hay numerosos ejemplos de las posibilidades que ofrece esta librería y que pueden ser visualizados en el navegador web de cualquier dispositivo moderno.

2.2.6. WebXR

WebXR es una API que permite a las páginas web interactuar con el hardware disponible en los dispositivos para permitir la creación de experiencias tanto de realidad virtual (introducción de objetos virtuales en una visualización del mundo real) como de realidad virtual (generación de un entorno virtual completo explorable). Los requisitos hardware variarán según el tipo de experiencia que se quiera crear, ya que para la realidad aumentada puede ser suficiente con un tablet o smartphone reciente que disponga de cámara, pero para la realidad virtual se requiere de un hardware específico que recoja información del entorno del usuario

y le muestre los gráficos generados a la vez que ocluya el espacio real en el que se encuentra. Aunque pueda parecerlo, WebXR no genera ningún tipo de contenido gráfico, solamente se encarga de facilitar el intercambio de los datos necesarios entre el hardware y el software para que este último pueda crear dinámicamente fotogramas coherentes con el estado del hardware disponible (p.ej. si se ha pulsado algún botón o el usuario ha cambiado de posición)

2.2.7. A-Frame

A-Frame es un framework web para crear escenas de realidad virtual en el navegador (no requiriendo la instalación de ningún software adicional), que se apoya en HTML añadiendo etiquetas que permiten definir la estructura de la escena de forma sencilla como puede verse en el código 2.1, el cual produce el resultado que se puede observar en la Figura 2.8 al ejecutarse en un navegador.

Código 2.1: Código HTML de una escena de realidad virtual sencilla creada haciendo uso de A-Frame.

```
<!DOCTYPE html>
<html>
  <head>
    <script src="https://aframe.io/releases/1.3.0/aframe.min.js"></
        script>
  </head>
  <body>
    <a-scene>
      <a-box position="-1 0.5 -3" rotation="0 45 0" color="#4CC3D9"
          ></a-box>
      <a-sphere position="0 1.25 -5" radius="1.25" color="#EF2D5E"></
          a-sphere>
      <a-cylinder position="1 0.75 -3" radius="0.5" height="1.5"
          color="#FFC65D"></a-cylinder>
      <a-plane position="0 0 -4" rotation="-90 0 0" width="4" height=
          "4" color="#7BC8A4"></a-plane>
      <a-sky color="#ECECEC"></a-sky>
    </a-scene>
  </body>
</html>
```

Fuente: código HTML extraído de *https://aframe.io/examples/showcase/helloworld/*

Este framework facilita trabajar con *three.js*, que a su vez se apoya en *WebGL*, abstrayendo muchas operaciones como la rotación y traslación de entidades en forma de valores asociados a atributos HTML. *A-Frame* tiene una arquitectura *Entidad-Componente-Sistema*, la cual promueve un diseño más organizado al fomentar la encapsulación, reutilización de código y la modularización a través de la creación de componentes, que se pueden asociar a las entidades para dotarlas de comportamiento.

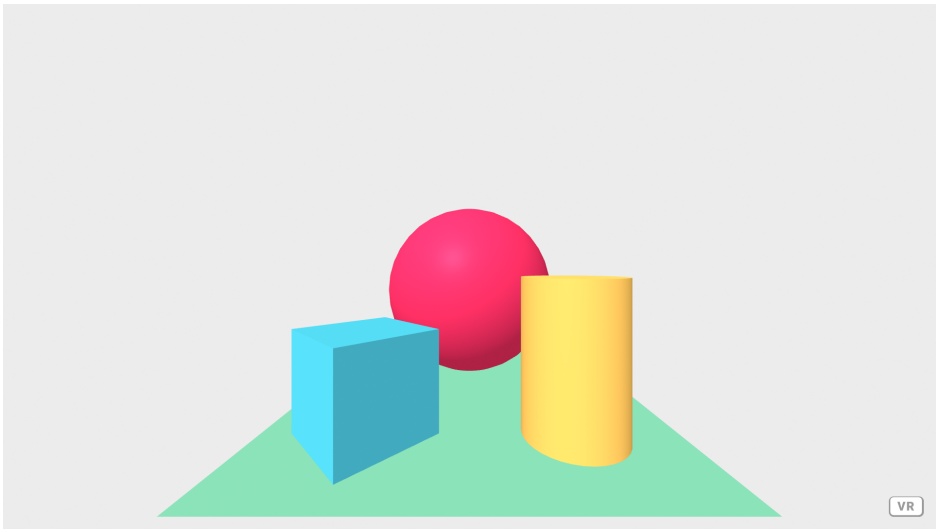


Figura 2.8: Página web resultante del código 2.1. A-Frame nos provee de un botón en la esquina inferior derecha para entrar en el modo de realidad virtual, que el usuario presionará cuando acceda desde un dispositivo compatible con ésta
Fuente: *https://aframe.io/examples/showcase/helloworld/*

Además de dotar a las entidades de comportamiento a nivel individual, se dispone también de 'sistemas', cuyo ámbito es global a toda la escena, además de permitir a los componentes comunicarse entre ellos por medio de eventos, los cuales pueden llevar datos asociados y dirigirse hacia entidades concretas o grupos de ellas, haciendo uso de las funciones *querySelector()* para seleccionar una única entidad o *querySelectorAll()*, para obtener una lista con todas las entidades que cumplan con el selector CSS proporcionado como parámetro.

2.2.8. HUGO

Para documentar el proceso de desarrollo, se ha creado un blog[4], para el cual se optó por utilizar *HUGO*[5] como framework, ya que permite transformar documentos *Markdown* en páginas web estáticas apoyándose en plantillas, que permiten configurar no solo el aspecto de la web sino también añadir elementos que puedan ser de utilidad, como índices de etiquetas para tener organizadas las páginas a través de ellas.

Otra de las ventajas por la que se decidió usar este framework frente a otros es que integra un servidor HTTP sencillo de iniciar y sin apenas configuración, junto a que es un proyecto *Open Source* y cuenta con una importante comunidad de usuarios que, además de contribuir a mejorar su documentación y crear guías

[4] `https://jdjuli.github.io/virto/`
[5] https://gohugo.io/

de uso, crean plantillas como la que se ha utilizado en este trabajo[6] y las publican de forma gratuita.

2.3. Componentes de A-Frame utilizados

En este proyecto se aprovecha la arquitectura *entidad-componente-sistema*[7] de *A-Frame* para enriquecer las escenas con componentes ajenos a *A-Frame*, tanto de propósito muy específico como los desarrollados para el proyecto como otros, más generales (y no por ello menos importantes) desarrollados por la Comunidad existente en torno a *A-Frame* y de los cuales se hablará en esta sección.

2.3.1. aframe-extras

Conjunto de componentes creados por el usuario de GitHub *n5ro*[8] para ampliar los disponibles en *A-Frame* y añadir así funcionalidades al framework. De esta librería se ha utilizado sobre todo el componente **sphere-collider**, que ha servido de base para crear otros componentes para la detección de colisiones más avanzados como el AABB (Axis Aligned Bounding Boxes) o el OBB (Oriented Bounding Boxes), cuya estructura es una copia de la de *sphere-collider*.

2.3.2. aframe-physics-system

Este proyecto añade componentes a *A-Frame* que permite simular físicas dentro de las escenas creadas a través de los motores de físicas *cannon.js*[9] y *ammo.js*[10], actualmente los únicos soportados y entre los cuales, el usuario puede elegir cuál usar.

En este proyecto se ha usado el motor *ammo.js*, que a nivel de escena, nos permite establecer el vector de atracción gravitatoria y otros valores comunes a todos los objetos como son la fricción con el espacio (simula la deceleración causada al atravesar la atmósfera) o la 'restitución' (factor de rebote o elasticidad de las colisiones).

A nivel de entidad, *ammo.js* nos provee de tres componentes, de los cuales los más importantes son *ammo-shape* para definir las características de la caja de colisión que envuelve al modelo tridimensional (para evitar usar como caja de

[6] https://github.com/nanxiaobei/hugo-paper
[7] `https://aframe.io/docs/1.3.0/introduction/entity-component-system.html`
[8] `https://github.com/n5ro`
[9] `http://schteppe.github.io/cannon.js/`
[10] `https://github.com/kripken/ammo.js/`

colisión la malla del objeto cuando éste tenga muchos polígonos y *ammo-body*, que nos permite establecer las propiedades físicas de esa entidad, tales como su masa, su 'resistencia a la rotación' o si debe emitir eventos cuando colisiona con otras entidades, que pueden ser de tres tipos, 'estáticos', 'kinemáticos' o 'dinámicos', siendo los primeros aquellos con los que únicamente se puede colisionar y no les afecta la gravedad, los segundos, simulados pero cuyos atributos físicos (posición, rotación) se pueden alterar programáticamente y los terceros, objetos completamente dinámicos y con los que solo se puede interactuar estableciendo restricciones físicas entre ellos, con entidades 'estáticas' o 'kinemáticas'.

El último componente que nos provee *aframe-physics-system* cuando se utiliza el motor de físicas *ammo.js* se llama *ammo-constraint* y establece una restricción entre la entidad a la que se le añade y la que se haya referenciado en el componente; esta restricción puede ser de muchos tipos, como bisagras, 'misma posición' o muelles que unan las entidades de forma elástica, entre otros.

2.3.3. Ammo.js

Ammo.js es una compilación realizada por el usuario de GitHub *kripken*[11], haciendo uso del compilador *emscripten*[12] para portar el motor de físicas *bullet*[13] (mantenido por la organización *Bullet Phisics SDK*), desarrollado en C++, a WebAssembly, de forma que pueda usarse a través de JavaScript en los navegadores web y en este caso concreto, en los navegadores de los visores de realidad virtual.

2.3.4. aframe-environment-component

Es un componente creado por la organización *Supermedium*[14], compuesta por los mantenedores del framework *A-Frame* y que permite añadir con muy poco esfuerzo un entorno visual a las escenas de A-Frame, compuestas por un suelo, una bóveda celeste y elementos decorativos.

Algunos entornos pueden ser configurados para simular el ciclo día-noche, niebla o proveerlos de una semilla inicial para generar siempre el mismo entorno o hacer que sea diferente cada vez que se visualice la escena, es por todo ello por lo que ha sido empleado en este proyecto para hacer la escena más amigable y resolver en gran medida el problema de orientación dentro del espacio virtual.

[11] `https://github.com/kripken`
[12] `https://emscripten.org/`
[13] `https://github.com/bulletphysics/bullet3`
[14] `https://github.com/supermedium`

2.3.5. aframe-teleport-controls

Creado por el usuario de GitHub *fernandojsg*[15], este componente añade la posibilidad de que el usuario se pueda desplazar por las escenas teletransportándose a la posición que desee, pulsando un botón de los controladores para definir el destino del teletransporte (se marca con un círculo de color en las superficies compatibles) y soltándolo para efectuarlo. En este proyecto, se usa una versión modificada de su componente que funciona exactamente igual, pero corrige una serie de errores para minimizar el número de advertencias emitidas por WebGL y con ello, evitar que oculte posibles errores de otros componentes.

2.3.6. aframe-super-hands-component

Es un conjunto de componentes desarrollados por el usuario de GitHub *wmurphyrd*[16] para permitir al usuario interactuar con las entidades de la escena mediante los controladores de los dispositivos de realidad virtual. Sin estos componentes, se tendrían que haber programado desde cero acciones tan naturales y cotidianas como agarrar un objeto para desplazarlo, rotarlo o escalarlo.

Si se utiliza junto con el componente *aframe-physics-system*, se pueden crear interacciones muy interesantes desde el punto de vista de la experiencia de usuario, ya que no solo se podrán manipular objetos sino que éstos también pueden tener atributos físicos como si pertenecieran al mundo real (masa, velocidad, aceleración, inercias, etc.).

El nexo de unión entre ambos conjuntos de componentes son los componentes que establecen restricciones entre entidades dentro del motor de físicas (*constraint* en caso de usar el motor *Cannon.js* y *ammo-constraint* si se usa *Ammo.js*), pero los componentes desarrollados por *wmurphyrd* solo soportan el motor *Cannon.js*, mientras que ambos están soportados en el *fork*[17] realizado por *diarmidmackenzie* en GitHub.

2.3.7. model-opacity

Este componente surge de la necesidad de modificar el factor de transparencia del material asociado a una entidad. Fue propuesto por *don-mccurdy* (*donmccurdy*[18] en GitHub) como respuesta a *una pregunta*[19] realizada en la web de preguntas y respuestas StackOverflow.

[15] `https://github.com/fernandojsg`
[16] `https://github.com/wmurphyrd`
[17] `https://github.com/diarmidmackenzie/aframe-super-hands-component`
[18] `https://github.com/donmccurdy`
[19] `https://stackoverflow.com/questions/43914818/alpha-animation-in-aframe-for-a-object-model`

No altera la transparencias de las entidades anidadas, pero en la mayoría de casos no supone un problema, ya que al ser *Open Source* siempre quedaría la opción de mejorarlo o simplemente añadirlo o eliminarlo de manera recursiva en todas las entidades anidadas.

3

Desarrollo del prototipo

En este capítulo se explica la metodología de desarrollo aplicada y las iteraciones realizadas hasta llegar a un producto mínimo viable

3.1. Método de trabajo

Durante el desarrollo se han llevado a cabo tres tareas simultáneamente, aprender a programar escenas en realidad virtual, diseñar el prototipo e implementarlo.
Debido a que se trata de una tecnología novedosa y puede surgir la necesidad de modificar el diseño del prototipo según vayan apareciendo problemas o haya que descartar ideas por su complejidad de uso, el modelo de desarrollo elegido ha sido el **iterativo incremental**, de forma que el trabajo se organiza en '*sprints*', los cuales se han dividido en cuatro etapas:

1. Planificación de la iteración y diseño de los cambios a través de reuniones periódicas con el co-tutor

2. Implementación del diseño obtenido en la fase anterior y refactorización del código

3. Pruebas. El prototipo no debe perder funcionalidad a causa de los nuevos cambios

4. Analizar los resultados haciendo pruebas del prototipo en busca de puntos de mejora. Al final de esta etapa, se archivan los prototipos obtenidos y se concretan las ideas que se llevarán al punto 1 de la siguiente iteración.

Los resultados de las iteracciones serán escenas de realidad virtual que recibirán el nombre de '*demostraciones*' o '*demos*' para abreviar, cuya usabilidad se irá viendo incrementada con el tiempo. Debido al carácter experimental de este desarrollo, el resultado de algunas etapas no será una única *demo* sino varias, con el objetivo de poder probar múltiples enfoques al mismo tiempo, en los casos en que esto ocurra, no se podrá establecer una relación directa entre el número de una demostración y la iteración en la que se desarrolló, ya que las demostraciones se numeran consecutivamente.
Además de las escenas de demostración, se irán creando otras para probar conceptos bajo el nombre de *test*, las cuales contendrán lo mínimo imprescindible para materializar ideas que podrán acabar siendo incorporadas o no a las demostraciones, en cualquier caso, no se garantiza funcionamiento y es posible que en las iteraciones posteriores sufran regresiones o dejen de funcionar.
Cuando se realicen una o más iteraciones cuyo resultado sea interesante analizar, se escribirá una entrada en el blog[1] comentando los avances realizados, así como las decisiones tomadas y algunos de los problemas encontrados y cómo se han resuelto o se pretenden resolver.

3.2. Iteraciones

En esta sección se agrupan las iteraciones realizadas para ofrecer una visión general de las fases que ha ido atravesando el desarrollo del prototipo.

3.2.1. Iteración 1

El objetivo de esta primera iteración fue comenzar a crear escenas tridimensionales sencillas con A-Frame, desarrollando componentes que pudieran ser de utilidad más adelante e integrar otros ya creados por diferentes autores con el objetivo de poder dedicar más tiempo a la experimentación.

Se eligió el IDE *Visual Studio Code* como editor de código e inicialmente se usó el servidor web que integra para hacer las primeras *demos* además de *Git* como sistema de control de versiones, subiendo los cambios a un repositorio en GitHub[2].
Más adelante, con la ayuda de un visor *Meta Quest*, se pudieron empezar a visualizar las escenas en realidad virtual y con ello, surgió la problemática de que

[1] `https://jdjuli.github.io/virto/`
[2] https://github.com/jdjuli/virto

por seguridad, para poder acceder modo de realidad virtual en el navegador web del visor, la página tiene que ser accedida a través de https y el servidor de *Visual Studio Code* no lo soporta en su configuración por defecto, tras varios intentos de configurarlo para que hiciera uso de HTTPS, la solución encontrada fue programar las escenas en el IDE y después subir el código a proyectos HTML almacenados y servidos por *glitch*[3], el cual, como se puede ver en la figura 3.1, provee a los programadores de un editor de código, además de una URL única y accesible a través de HTTPS, con lo que utilizando esa dirección, fue posible comenzar a probar y depurar las escenas desarrolladas en el visor *Meta Quest*.
Depurar los errores con el visor de realidad virtual supuso un problema de co-

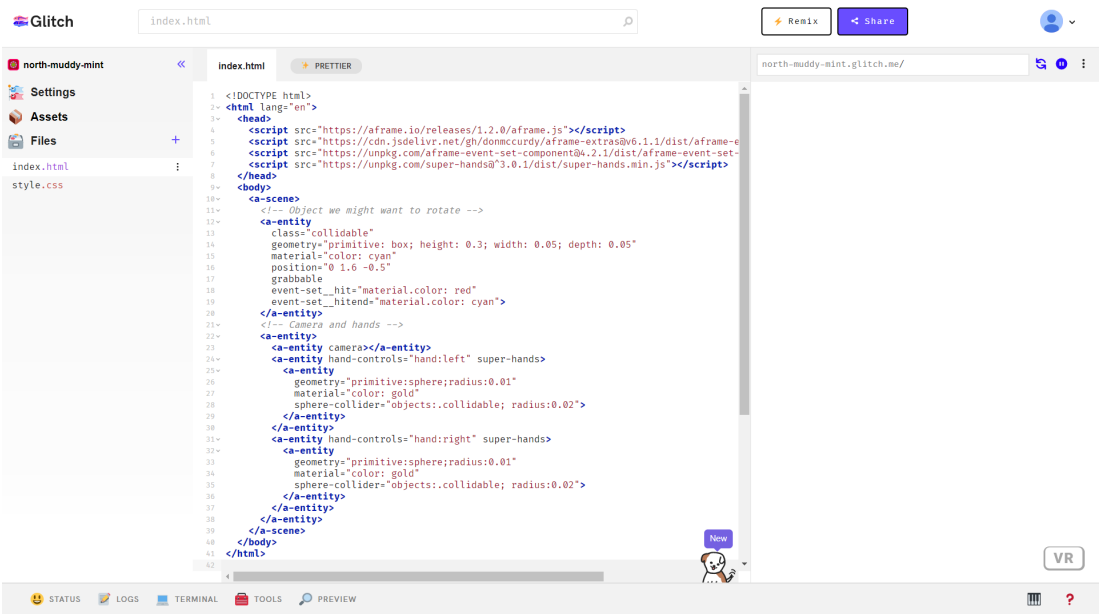


Figura 3.1: Captura de pantalla del IDE proporcionado por *glitch.com*[4]. A la izquierda se pueden administrar los ficheros que forman parte del proyecto, en la parte central, el editor de código y, opcionalmente en la parte derecha, se puede desplegar una previsualización de la página que resulta de nuestro código
Fuente: Elaboración propia

nectividad, ya que pese a que el cable de carga y depuración era suficientemente largo para conectar el visor al ordenador, no ofrecía la libertad de movimiento requerida en algunas escenas y, después de investigar alternativas a la depuración por cable, se optó por habilitar la depuración inalámbrica en el dispositivo[5] y, después de conectar a la misma red el visor y el ordenador con el que se llevaría a cabo la depuración, se estableció una conexión entre ambos de forma que fue posible llevar a cabo la depuración prescindiendo del cable.

Los primeros componentes desarrollados fueron relativamente sencillos y rea-

[3]https://glitch.com/
[5]`https://developer.oculus.com/documentation/native/android/ts-adb/`

lizaban tareas como esperar a que que se desencadenara un evento de clic sobre la entidad asociada para hacerla 'saltar' aumentando su posición vertical durante unos pocos segundos o, emitir un evento diferente que recogería otro componente, haciendo saltar a la entidad que tiene asociada (diferente de la que recibió el clic (demos 3 y 4).

Las demos posteriores se dedicaron a desarrollar componentes que cooperaran con los motores de físicas haciendo uso de los componentes de *aframe-physics-system*. En primer lugar, se reimplementó la escena en la cual las entidades 'saltan' cuando se hace clic sobre ellas (demo 3), aplicando en este caso un impulso vertical en lugar de cambiando las coordenadas (demo 5) y desarrollando después en la demo 6 una simulación de cinco esferas de colores rebotando y cambiando de color cuando colisionan entre ellas, en la figura 3.2 puede apreciarse la simulación en funcionamiento en un momento en que se han producido dos colisiones que involucran a dos esferas cada una, la escena no es interactiva más allá de poder cambiar el punto de vista, pero sirvió para descubrir las posibilidades que ofrece reaccionar a eventos fruto de la simulación de físicas y no solo de aquellos provocados por la interacción humana.

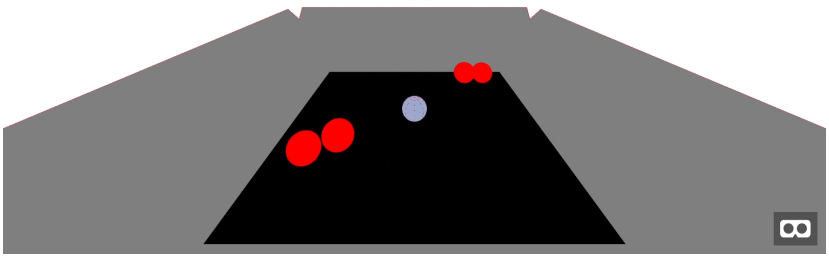


Figura 3.2: Captura de pantalla de la demo 6, la cual consiste en un recipiente con las paredes laterales inclinadas 45^{o} sobre el cual se dejan caer cinco esferas que cambian temporalmente su color a rojo cuando colisionen con otras como ocurre en la imagen.

Fuente: Elaboración propia

Más adelante, se empezaron a utilizar los componentes de *aframe-super-hands-component* para permitir a los usuarios participar en la escenas de manera directa, en primer lugar, permitiéndoles mover cubos flotantes y posteriormente, dotando a los cubos de masa y aceleración gravitatoria, a partir de lo cual se creó una escena a modo de primer acercamiento a lo que podría ser programar en realidad virtual, explorando la idea de construir programas apilando 'losas' virtuales de colores, los cuales representarían una instrucción determinada y, en concreto, para la demo que se desarrolló, se emplearon los colores cyan y rojo para representar

Algoritmo 1 *Ejecución demo 11* **trigger event**=*run* **output**=*none*

Require: Having received a '*clic*' event throught user interaction.

if not *running* **then**
  running ← **true**
  entities ← Entities with component '*instruction*'
  size ← *length*(*entities*)
  Sort *entities* by increasing height {attribute '*.object3D.position.y*'}
  for $i \leftarrow 0$ **to** $size - 1$ **do**
    action ← *entities*[*i*]
    Set timeout of $(250 \cdot i)$ milliseconds :
      Emit *'run'* event **to** *action*
  end for
  Set timeout of $(250 \cdot size)$ milliseconds:
    running ← **false**
  so ← CALL me(maybe)
end if

dos instrucciones que hicieran descender y ascender un móvil tridimensional al que se denominó 'drone' por su parecido a los dispositivos homónimos del mundo real. Dicho *drone* estaría representado por una esfera de color verde la cual, si se hace clic sobre ella, desencadena la ejecución del programa creado, de manera que las losas apiladas se ordenan según su altura de manera ascendente y se ejecuta de forma secuencial la instrucción correspondiente a cada una de ellas.

El algoritmo 1, se implementó en el componente '*programa*' que se utilizó en la demo 11, con el objetivo de controlar la ejecución de las instrucciones según la altura a la que se hubieran apilado, de abajo a arriba. Dicho algoritmo no consideraba el caso en que varias instrucciones estuvieran a la misma altura de la base y, en dichos casos el orden de ejecución era impredecible, ya que el orden depende de el valor de altura proporcionado por el motor físico. Por otro lado, el movimiento del *drone* (representado por la esfera verde) se realizaba alterando su posición y no empleando los impulsos proporcionados por el motor de físicas con el fin de descomponer el desarrollo en fases incrementales para poder facilitar tanto la programación como la búsqueda y resolución de errores posterior.

Hacia el final de esta etapa, surgió la oportunidad de participar en el *Concurso Universitario de Software Libre* con este proyecto y el hecho de que se encontrara en fase inicial, no supuso ningún problema a la hora de presentarlo a concurso tal como estaba, habiendo únicamente que crear un blog[6] en el cual se han ido escribiendo entradas que explica e ilustra el progreso de este proyecto, así como las dificultades encontradas y resolverlas. Este blog se continuó después de terminar el concurso como forma de documentar el proceso de desarrollo y dar a conocer el proyecto.

[6] `https://jdjuli.github.io/virto/`

De manera previa a la participación en el concurso, el proyecto contaba con una página web sencilla, creada con *Bootstrap* para servir de índice y agrupar las escenas que se iban creando. La idea inicial fue adaptar esa página para añadir una sección de blog de cara al concurso, pero después de algunas pruebas se comprobó que resultó más económico en términos de tiempo, emplear un generador de páginas web estáticas como HUGO, que permiten transformar archivos *MarkDown* en páginas web mediante el uso de plantillas, con la ventaja de que la web generada puede ser hospedada en cualquier servidor de páginas web estáticas, como el provisto por GitHub a través de *GitHub Pages*[7].

3.2.2. Iteración 2

En esta segunda iteración, se comenzó explorando la idea de manipular figuras geométricas usando los controladores de realidad virtual, para ello, nuevamente se coordinaron los componentes de *aframe-physics-system* con los de *aframe-super-hands-component* a través de nuevos componentes creados específicamente para permitir desplazar libremente aquellas entidades que tuvieran asignada una clase CSS concreta (por medio del atributo *class* de HTML).
La demo 12 se desarrolló como ejemplo de uso del componente que permite desplazar entidades y en ella, se le presenta al usuario una mesa gris con tres formas geométricas en su lado derecho (un cubo, un cilindro y un cono), que puede agarrar cerrando la mano en torno al controlador de realidad virtual y desplazar arbitratiamente por la escena, abriendo la mano para dejarlas caer en el lugar deseado. Con lo aprendido en esta escena, se desarrolló otro componente que permitía duplicar las entidades en caso de que se encontraran a una distancia de su posición inicial superior a un límite configurable, de forma que el usuario pueda disponer de copias de aquellas entidades sobre las que tenga un especial interés (en vistas a que estas entidades más adelante se conviertan en instrucciones). En la demo 13, visualmente idéntica a la demo 12, se puede explorar este mecanismo de clonación de entidades y obtener con ello un resultado similar al que se puede ver en la figura 3.3, la cual sería imposible replicar en la demo 12 ya que carece de método alguno de clonación de entidades.

Partiendo de la funcionalidad mostrada en la demo 13 y recuperando la idea de asemejar las instrucciones a losas de colores, se desarrolló la demo 14, en la cual las instrucciones están representadas por bloques de color azul y rojo (mover hacia abajo y hacia arriba respectivamente), que el usuario puede agarrar haciendo uso del rayo de luz que sale de los controladores para construir el programa dejando caer los bloques sobre la 'alfombra' de color azul, añadiendo la instrucción al principio del programa (en la parte inferior). Se añadió también la posibilidad de extraer instrucciones del programa agarrándolas nuevamente con el haz de luz y arrastrándolas fuera de la torre.

[7] https://pages.github.com/

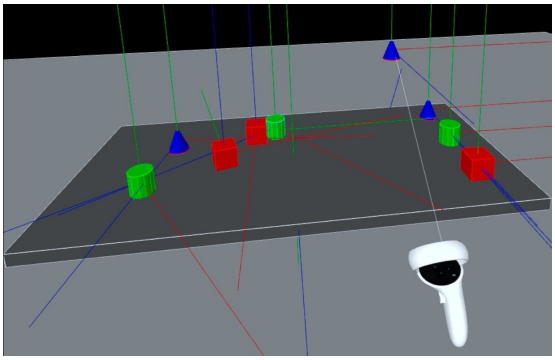


Figura 3.3: Captura de pantalla de la demo 13, a la derecha se dispone de tres entidades con forma de cono, cilindro y cubo, si se apunta a una de ellas y se mantiene pulsado el botón lateral del controlador, se puede extraer una copia de dicha entidad y situarla sobre la mesa.
Fuente: Elaboración propia

Esta demo cambia ligeramente la forma de ejecutar el programa propuesta en la demo 11, ya que en ella la ejecución del programa se inicia con la recepción de un evento configurable, lo que permite crear nuevas formas de desencadenar la ejecución, lo cual se puede apreciar en la demo 14 en forma de cubo verde, que cuando recibe un evento de *clic*, emite el evento que desencadena la ejecución.

Comenzando en esta demo, los componentes que abstraen conceptos de programación se han ido diseñando de forma que su posición dentro del código HTML complemente su significado y facilite la comunicación entre ellos a la vez que permite definir una sintaxis para el lenguaje que se está desarrollando. Este modelo parecía muy prometedor, pero el hecho de que *A-Frame* no responda adecuadamente al re-emparentado de nodos del DOM, hizo que hubiera que replantear esta operación de manera que se añadiera al padre destino una copia del nodo que se desea mover, destruyendo posteriormente el nodo original. Esta solución supone la reinicialización de los componentes asociados a la entidad re-emparentada, además de provocar fallos en todos aquellos componentes que hicieran uso de una referencia al nodo eliminado.

Una vez implementada esta forma de re-emparentado, no tardó en aparecer el primer efecto colateral, ya que si usando los componentes *aframe-super-hands-component* se desplaza una entidad y en algún punto del recorrido esta se re-emparenta, dicha entidad se desprende de la mano que la agarraba pese a que ésta continue cerrada, haciendo que haya que agarrarla de nuevo para poder seguir desplazándola.

Tras analizar en detalle el código de los componentes de *aframe-super-hands-component*, se encontró una forma de resolver este problema, que consistía en clonar junto al nodo correspondiente a la entidad agarrada, el estado de sus componentes asociados y posteriormente, actualizar las referencias al nodo elimi-

nado en los componentes encargados de llevar a cabo el desplazamiento, pero se descartó esta solución debido a que depende mucho de la implementación de componentes desarrollados por otras personas, lo cual podría complicar más adelante el mantenimiento de los componentes desarrollados para este trabajo.

La demo 14 también supuso un nuevo reto respecto al motor de físicas, ya que como se puede comprobar, al agarrar los cubos rojo y azul por primera vez, se desplaza la entidad pero no la caja de colisión, que sí se desplaza junto con la entidad cuando vuelve a ser agarrada por segunda vez. Este problema se resolvería en las siguientes demos cambiando el orden en que se inicializan los componentes y más adelante, haciendo que las instrucciones dejaran de formar parte del motor de físicas.

Después de comprobar que los cubos y las losas representan razonablemente bien la estructura de un programa secuencial, se procedió a mejorar la interacción entre el programa y el usuario, concretamente la manera en que los programas son editados, para ello, en primer lugar, se creó un componente específico que permitiera, a través de una compacta interfaz de tres botones, crear instrucciones que pudieran añadirse al programa, para después trabajar en la manera en que las instrucciones se insertarían y eliminarían del programa.

En esta línea, la demo 15 propone utilizar restricciones de proximidad entre las instrucciones que forman parte del programa, generando un 'muelle' que se estira cuando se arrastra alguno de los bloques que lo componen. Esta idea resulta visualmente atractiva y es eficaz cuando se añaden bloques al inicio o al final del programa, pero el problema que llevó a descartar esta idea es lo sencillo que resulta crear bucles de restricciones dentro del programa, los cuales provocan que los bloques se ordenen de manera impredecible y, en el peor de los casos, bloquean la pestaña del navegador.

Vistos los problemas derivados de usar físicas en las instrucciones que forman parte del programa, se abandonó esta idea y se optó por reducir el tamaño de las instrucciones cuando se agarran para facilitar hacerlas colisionar con la posición del programa donde se quieren insertar y, hacer que una vez insertadas, se les asigne una posición vertical fija y solo se puedan desplazar en horizontal, marcando claramente su orden de ejecución y resolviendo así el problema de los bucles de restricciones.

El siguiente paso que se dio fue mejorar la información de que dispone el usuario a la hora de modificar el programa, ya que la forma de extraer las instrucciones arrastrándolas fuera de él, deja claro que instrucción se va a eliminar, pero en cambio, insertar instrucciones haciéndolas colisionar con la instrucción anterior (o la base en caso de no haber ninguna), no resulta tan evidente. Para resolver este inconveniente, se propuso añadir una etapa de previsualización de las instrucciones en el lugar que ocuparán en el programa antes de añadirlas, de forma que el usuario pueda decidir con más precisión dónde las inserta. En la fi-

gura 3.4 puede verse la previsualización de la instrucción cuando esta se arrastra por encima de la primera instrucción del programa.

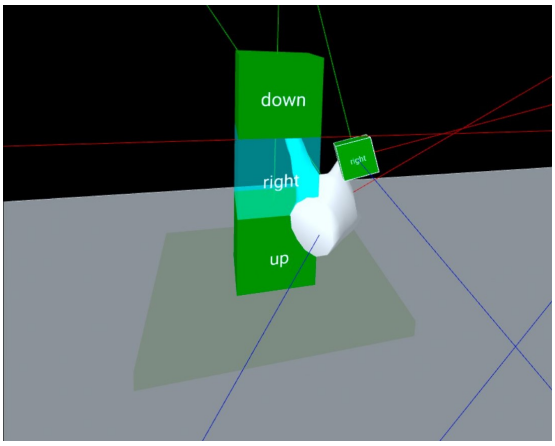


Figura 3.4: Captura de pantalla que muestra el resultado de previsualizar una instrucción en el lugar que ocupará en el programa. Dicha previsualización es translúcida para transmitir al usuario la idea de que no es definitiva, sino que se trata de una ayuda visual.

Fuente: Elaboración propia

Como resultado de esta segunda iteración, se desarrolló la demo 18, que cuenta con un aspecto visual más atractivo al haber añadido texturas a las entidades y un entorno a la escena, la cual implementa el selector para crear entidades, la previsualización de instrucciones y un nuevo botón, al lado del de ejecutar, para reestablecer la posición del drone, el cual se representa con un modelo tridimensional texturizado, en lugar de la esfera verde empleada en otras demos.

3.2.3. Iteración 3

Durante el análisis del resultado de la iteración anterior, se detectaron problemas de escalabilidad en el modelo de programación implementado, como la dificultad para crear programas arbitrariamente largos, que podrían tener una altura superior a la alcanzable por el usuario, dificultando que éste pudiera editarlos. También se planteó cómo se podrían integrar en ese modelo de apilamiento las estructuras de control condicional y repetitiva, y aunque a priori era factible, volvía a surgir el problema de la altura del programa.

Una mejora que sí se podía introducir sin problemas era la parametrización de las instrucciones, ya que es relativamente fácil diseñar un modelo 3D con huecos, que representaría una instrucción a la cual es posible asociar parámetros introduciéndolos en dichos huecos. Otra posibilidad que se desprendió de esta última idea era la posibilidad de añadir instrucciones aritméticas, de forma que además de mover el drone, el programa pudiera realizar operaciones aritméticas y/o lógicas

cuyo resultado pudiera usar como parámetro en instrucciones y estructuras de control.

Valorando las oportunidades y problemas de escalabilidad que tenía el modelo de programa que se venía utilizando, se optó por rediseñar la manera en que se visualizan y modifican los programas para que las instrucciones se apilaran horizontalmente y con ello, quedara resuelto el problema de los programas de altura arbitraria, aunque apareciera otro más leve relacionado con la dificultad de alcanzar una entidad lejana dentro de un espacio real reducido, que se resolvió permitiendo al usuario teletransportarse por la escena. La demo 19 es la primera en implementar esta nueva idea de programas horizontales, y lo hace, como se aprecia en la figura 3.5, mostrando un gran bloque gris en el lado izquierdo de la mesa, que sirve de base para empezar a colocar instrucciones en su lado derecho. Dichas instrucciones se han reducido a dos, 'mover' y 'rotar', a la vez que han dejado de ser cubos para tener un aspecto más alargado y dos orificios, uno cuadrado pensado para insertar parámetros cualitativos como la dirección del movimiento o el eje de giro y, otro cilíndrico para insertar un valor que determinará 'cómo de fuerte' se realizará la acción, ya que en esta escena, el *drone* se desplaza mediante los impulsos angular y lineal que provee el motor de físicas.

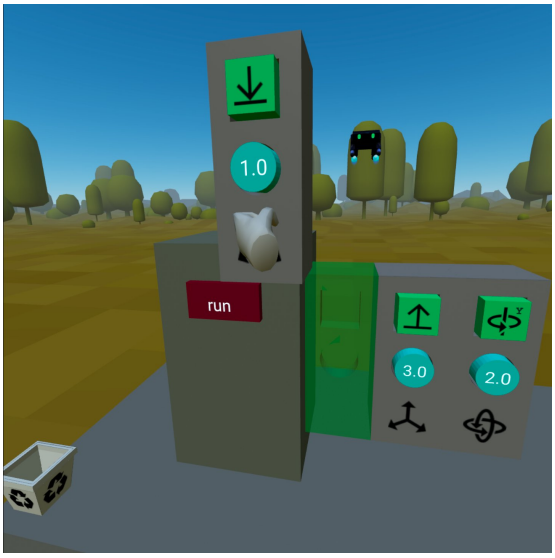


Figura 3.5: Captura de pantalla de la demo 19, se puede apreciar cómo la estructura del programa es horizontal, así como se conserva la idea de la previsualización y, las instrucciones pueden ser parametrizadas a través de los dos huecos que tiene el modelo 3D empleado.

Fuente: Elaboración propia

De manera paralela al desarrollo de escenas en las cuales los programas se construyan horizontalmente, se exploró una nueva área, la detección de errores y su notificación al usuario. Para ello, se partió de la base de que los errores debían mostrarse de manera discreta para no causar alarma en el usuario, de

manera que se optó por añadir una etapa de validación antes de ejecutar los programas, de manera que si la validación encontraba errores, se reprodujera un sonido grave y se abortara la ejecución, mientras que si por el contrario, la validación finalizaba exitosamente, el sonido que se reprodujera fuera agudo y comenzara la ejecución. Implementar este sistema de detección de errores fue sencillo en los programas iterativos y tuvo un efecto lateral positivo inesperado: el usuario recibía feedback no sobre si el programa estaba correcto o no, sino también sobre si había pulsado correctamente el botón de ejecutar, pero con todo, no se continuó desarrollando esta funcionalidad para trabajar en añadir nuevas características al entorno, como la inclusión de estructuras de control. Hacia el final de la iteración anterior, se trabajó en la creación de un nuevo sistema de creación de entidades que sustituyera a la anterior interfaz de tres botones, de manera que el usuario pudiera crearlas sin tener que desplazarse a un punto concreto de la escena. El resultado fue un panel ligado a la mano izquierda, que se muestra y oculta apretando el *joystick* analógico y permite al usuario desplazarse a través de las diferentes entidades que puede crear al tocar los botones con los símbolos $<$, para ver los elementos más a la izquierda y $>$ para hacer lo mismo con los que se encuentran más a la derecha. Para crear una entidad, el usuario solo tiene que pulsar el icono correspondiente al tipo de entidad que quiere crear.

Más adelante, en esta iteración, se mejora la idea del panel para crear entidades, suprimiendo los botones, los cuales se sustituyeron por el movimiento del *joystick* analógico, lo que permitió añadir 'categorías', que consisten en colecciones de tipos de entidades con un mismo propósito (como pueden ser las instrucciones, sus parámetros o las variables), entre las cuales se puede navegar moviendo el *joystick* verticalmente y, en el supuesto de que una categoría contenga más elementos de los que pueden visualizarse en el panel, se podrá navegar por ellos moviendo el *joystick* horizontalmente. En la figura 3.6 se muesta el menú (panel) antes y después de haber realizado los cambios que se describen.

(a) (b)

Figura 3.6: Diferencia entre los menús para crear entidades empleados en la demo 18 (a) y en la demo 19 (b). Como puede verse, la principal diferencia radica en la supresión de los botones con flechas y la adición de un texto que indica qué tipo de entidades se pueden crear en cada momento (parámetros en este caso).

Inicialmente, los parámetros cuantitativos de las instrucciones se representa-

ron mediante cilindros horizontales de color cyan que almacenaban en su interior un valor constante y mostraban dicho valor en su parte frontal, pudiendo dicho valor ser sustituido por una letra, asemejándolos a las constantes de los lenguajes de programación tradicionales.

Los parametros cualitativos fueron más sencillos de implementar, ya que internamente solo almacenaban el tipo de parámetro que representaban y lo visibilizaban aplicando una textura concreta a su geometría cúbica.

Las variables propiamente dichas, se implementaron un poco más adelante, desarrollando para ello dos componentes que materializaban dos conceptos comunes al hablar de programación, creando por un lado, el componente 'referencia', que sirve de interfaz a las instrucciones para leer y modificar el valor de la variable a la que pertenece esa referencia y por otro lado, el componente 'variable', que almacena internamente su valor y se encarga de generar referencias a sí misma que después podrán ser introducidas en las instrucciones.

Adicionalmente, se creó un componente llamado 'selector', diseñado para situarse encima de las variables y que podía desplazarse lateralmente por el ancho de la variable para modificar su valor. La falta de precisión observada al utilizar este selector motivó la parametrización del componente 'variable' para configurar el rango de valores del que se quiera disponer, lo cual no soluciona completamente el problema pero mejora la experiencia sin recurrir a introducir el valor usando un teclado virtual o alguna herramienta equivalente.

Con vistas a que el entorno de programación fuera escalable, se abstrayeron los programas como entidades independientes entre sí y se creó el componente 'ide', el cual mantiene una lista actualizada de los programas que hay dentro de su entidad, permitiendo que se pudiera introducir más de una 'mesa' de programa en la escena y se pudiera ejecutar cualquiera de los programas presentes en ellas de forma independiente sobre el mismo *drone* o sobre drones diferentes si así se especifica en el HTML de la escena.

Otro propósito de poder tener múltiples programas en una misma escena es permitir más adelante invocar dichos programas desde otros, de forma que programas complejos puedan descomponerse en subprogramas más fáciles de implementar y probar, incentivando así a los usuarios a programar de esta manera.

Tras considerar detenidamente implementar una operación para invocar otros programas, se pospuso su desarrollo ya que plantea problemas relativos al ámbito de las variables y la parametrización de los programas que se deberían que resolver antes de implementar esta funcionalidad, lo que no frenó la creación de las demo 20 con un programa en la escena que permite hacer uso de variables para parametrizar las instrucciones y la demo 21, idéntica a la anterior, pero ofreciendo además de un programa de ejemplo, otro completamente vacío como puede verse en la figura 3.7, preparado para que el usuario desarrolle en él desde

cero cualquier programa. En ninguna de las dos se puede invocar un programa desde otro, pero cada uno de ellos tiene su propio ámbito de variables, iniciando el camino hacia la jerarquía de ámbitos de variables presente en un importante número de lenguajes de programación existentes.

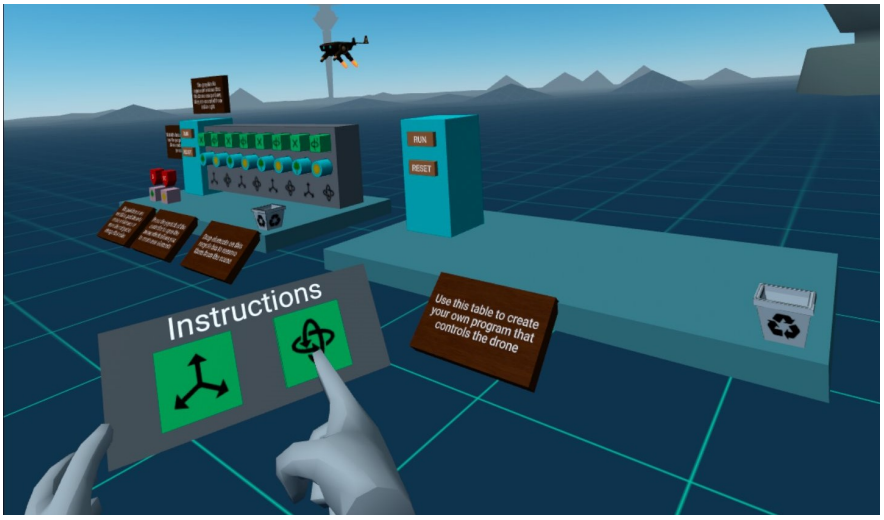


Figura 3.7: Captura de pantalla de la demo 21, que a diferencia de la demo 20, incorpora un segundo programa asociado al mismo *drone*, completamente vacío para permitir al usuario crear un programa desde el principio mientras conserva el programa como ejemplo.
Fuente: Elaboración propia

3.2.4. Iteración 4

Partiendo de las escenas desarrolladas en la iteración 3 y comprobando que el modelo de programación diseñado funciona mejor que el empleado en la iteración 2, se decide continuar con él, introduciendo estructuras de control y creando un nuevo tipo de variable que permita comandarlas.

La geometría y textura de las estructuras de control se diseñó de manera que una vez introducidas en el programa, su aspecto fuera similar a un diagrama de flujo, además de facilitar a los usuarios con poca o ninguna experiencia programando, la comprensión del funcionamiento de estas estructuras mediante el uso de los colores verde y rojo para representar qué instrucciones se ejecutarían en función del valor de la variable asociada a la estructura de control.

Para implementar estas estructuras de control, ha sido necesario repensar la sintaxis del lenguaje, de forma que un programa esté compuesto por un ámbito de variables y una sección de código. Las secciones de código podrán encontrarse también dentro de las estructuras de control, compartiendo el ámbito con el

programa, de forma que el condicional contendrá dos bloques de código y decidirá cual de ellos ejecutar en función de la variable que tenga asociada y después pasará el control a la instrucción inmediatamente posterior al condicional, mientras que en los bucles, en caso de que la variable asociada a ellos tenga el valor 'Verdadero', se alternará la ejecución de su único bloque de código interno con la consulta de la variable asociada, hasta que ésta tenga el valor 'Falso'.

La inclusión de la estructura de control repetitiva trae consigo la posibilidad de crear programas que no terminen nunca a causa de los llamados 'bucles infinitos', los cuales se ha decidido permitir solo en el caso de que el bloque de código del bucle contenga al menos una instrucción, de manera que actúe como 'retardo' y no bloquee la escena. En caso de no cumplirse esta condición, el programa se ejecutará hasta llegar al bucle y terminará su ejecución sin mostrar ningún error. De igual manera, en caso de que alguna instrucción carezca de parámetro cualitativo (representados por cubos verdes) o cuantitativo (referencias a variables, representadas mediante cilindros de color cyan o rosa), la ejecución avanzará hasta dicha instrucción y se detendrá, lo cual puede servir de ayuda al usuario para identificar el error.

Teniendo en cuenta que esta forma de localizar los errores puede resultar confusa, se ha planteado de manera complementaria, envolver la instrucción en la que se detecta el error en una caja translúcida de algún color llamativo como pueden ser el rojo o el amarillo, habitualmente relacionados en el mundo de la informática con errores y advertencias respectivamente. Esto vendría a completar la idea explorada en la iteración 3 de reproducir un sonido al ejecutar el programa, siendo este grave si contiene errores y agudo si carece de ellos.

Partiendo de que las estructuras de control toman una decisión u otra en función de un valor de verdad, se modificó tanto el aspecto como el comportamiento de las variables para incluir la posibilidad de que éstas almacenen dicho valor de verdad, siendo las variables con el fondo rosa, numéricas y aquellas con el fondo azul, lógicas (también llamadas 'booleanas'). La modificación de las variables trajo consigo la necesidad de adecuar el componente destinado a seleccionar sus valores a este nuevo tipo, de forma que mientras se usa para seleccionar valores numéricos, sea de color azul y puede tomar valores intermedios, pero si se usa para seleccionar valores de verdad, su color será rojo o verde según el valor elegido y no podrá tomar valores intermedios, el selector se moverá automáticamente al extremo del intervalo más cercano al lugar donde se haya soltado.

Internamente, el componente 'variable' tiene un nuevo argumento que representa el tipo de la variable, el cual emplea para modificar la forma en que se valida el valor (en caso de ser numérica, solo aceptará números y no valores de verdad), además de para crear un selector sobre sí misma del tipo apropiado y crear las referencias de uno u otro tipo.

En esta iteración aparecieron nuevos problemas de usabilidad relacionados

con las estructuras de control, los cuales impiden extraer las instrucciones de su interior haciendo que, cuando se agarra una de ellas, se desplace la estructura de control completa, dificultando la programación. Inicialmente, se pensó que este problema pudiera estar ocasionado por algún fallo de programación en los componentes encargados de detectar la colisión de la mano con las entidades y permitir que estas sean agarradas, por lo que se hizo una revisión de los métodos de detección más comunes, implementándolos y probándolos en la escena, pero ninguno de ellos resolvía completamente el problema. Descubrir que este problema es transitorio y no se puede reproducir con facilidad llevó a pensar que podría estar causado por algún problema en alguno de los componentes utilizados externos a este trabajo, por lo que se tratará de solucionar más adelante revisando las interacciones entre componentes y colaborando con los respectivos autores en caso de ser necesario. Mientras tanto, para mitigar el problema, se ha propuesto interactuar con otras entidades antes de repetir la acción que produjo el fallo (con la esperanza de que se resuelva de esa manera) o manipular las estructuras de control fuera del programa para reducir su impacto.

Se comprobó también que el *drone* en ocasiones no ejecutaba todas las instrucciones del programa, saltándose algunas de manera aleatoria. Se depuró en detalle este fallo y se llegó a la conclusión de que es un fallo 'aceptable', al no comprometer la ejecución de la escena y ocurrir muy ocasionalmente, por lo que hasta que se resuelva, se recomienda ejecutar de nuevo el programa ya que según lo observado, la probabilidad de que este error se manifieste dos veces seguidas es ínfima.

4

Descripción para usuarios

Este capítulo tiene por objetivo mostrar cómo puede ser utilizado el prototipo atendiendo al propósito del usuario.

4.1. Usar el prototipo en realidad virtual

Como el prototipo está diseñado para ser utilizado con un visor de realidad virtual, a continuación se describirán los pasos a llevar a cabo para la composición de un programa sencillo , pero que a su vez haga uso de todas las funcionalidades del prototipo.
El prototipo también puede utilizarse desde un ordenador, no pudiéndose crear nuevas entidades en dicho caso, lo cual limita la capacidad de edición y creación de programas, pero cualquiera de las demás acciones que serán descritas a continuación pueden realizarse cambiando los gestos por movimientos de ratón y, los desplazamientos por pulsaciones de las teclas *wasd* o las flechas ←↑↓→.

Lo primero que se necesita para empezar a usar el prototipo es abrir un navegador web y acceder a una escena de realidad virtual creada con él y la cual se sirva a través de *HTTPS* para poder entrar en el modo de realidad virtual. Se asumirá que se cuenta con una escena ya creada con un programa en blanco (en la subsección 4.2 se explica cómo crearlas) y, una vez se haya accedido a ella, el usuario estará frente a una imagen similar a la que se puede observar en la figura 4.1, en la cual se puede apreciar en la esquina inferior derecha un botón con el texto '*VR*' escrito en él y que, al pulsarlo activará el modo de realidad virtual.

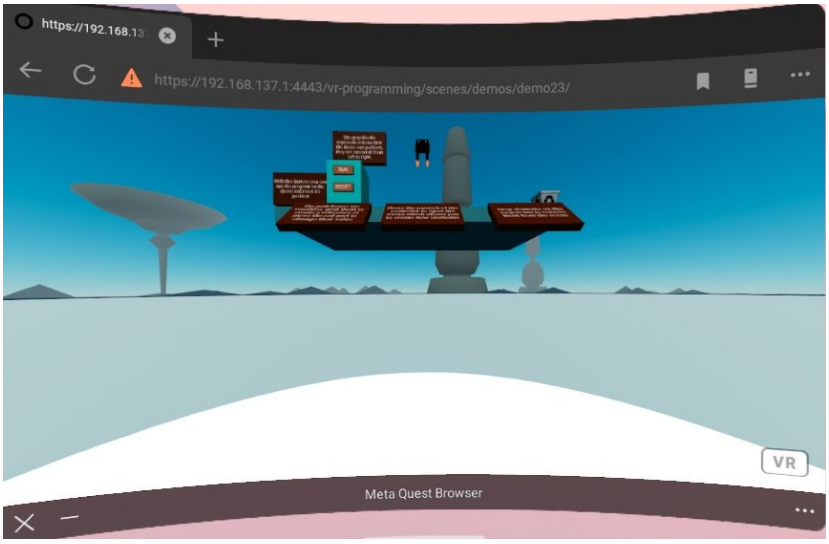


Figura 4.1: Navegador *Meta Quest Browser* visualizando una escena A-Frame antes de activar el modo de realidad virtual.
Fuente: *elaboración propia*

Una vez activado el modo de realidad virtual, lo primero que se apreciará es que los controladores se han convertido en unas manos que los sostienen y de ellos, como se puede ver en la figura 4.2, salen unas líneas con unos carteles en sus extremos que indican la utilidad de cada uno de los botones de los controladores, de los cuales cabe destacar el gatillo frontal, que sirve para activar el teletransporte; el gatillo lateral, que sirve para agarrar las entidades y, el *joystick*, el cual al pulsarlo muestra y oculta el menú para crear entidades, por el cual, una vez esté visible, será posible navegar moviendo el *joystick* correspondiente al controlador al que está asociado. Se puede mostrar u ocultar la ayuda presionando simultáneamente los botones B e Y de los controladores.

A continuación será necesario conocer los elementos de la escena, los cuales se pueden ver en la figura 4.3, y en cuya parte central está el drone, que se podrá operar a través de los programas creados en el entorno de desarrollo, en el cual, en este caso, hay un único programa (podría haber más que controlaran el mismo u otros drones) representado por una tabla horizontal de color gris, sobre el cual se encuentran, a la izquierda una caja de color cyan con dos botones en ella, los cuales servirán para ejecutar el programa ('*run*') y reestablecer la posición del *drone* ('*reset*'), y a la derecha, una pequeña papelera de reciclaje que sirve para destruir aquellas entidades que no sea necesario mantener en la escena.

Respecto a la tabla o mesa que representa el programa, en la parte más próxima al usuario, se mostrarán las variables que el usuario vaya creando y, en la parte más alejada la secuencia de instrucciones que componen el programa, la cual comienza en el bloque de color cyan y crece hacia la derecha.

Salvo que la escena cuente con un programa ya creado, no habrá ningún

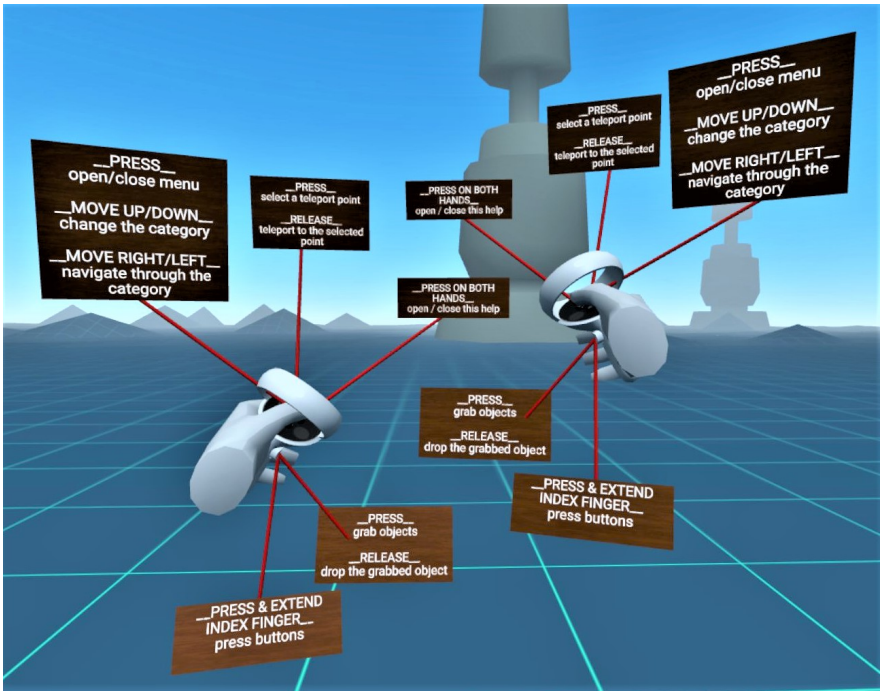


Figura 4.2: Mensajes de ayuda mostrados al usuario al entrar en el modo de realidad virtual

Fuente: *elaboración propia*

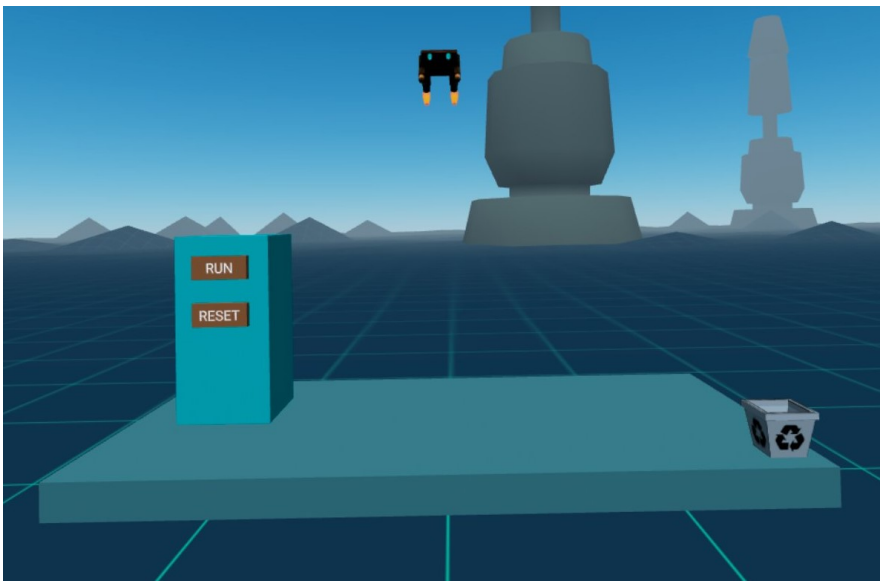


Figura 4.3: Programa vacío, a la derecha puede verse la papelera que permite destruir entidades, a la izquierda los controles del programa y en la parte superior central, el *drone* controlado por el programa.

Fuente: *elaboración propia*

elemento del lenguaje y el usuario tendrá que crearlos haciendo uso del 'menú

de creación de entidades', que es un catálogo de los componentes del lenguaje que pueden ser introducidos en el programa materializado en forma de tableta digital. Para abrir este menú, basta con pulsar cualquiera de los *joystick* de los controladores derecho o izquierdo y, una vez abierto, el usuario comprobará que está dividido en tres categorías ('instrucciones','parámetros' y 'variables'), entre las cuales puede cambiar moviendo el *joystick* hacia delante y hacia atrás, además de poder moverse a través de los elementos de una categoría concreta moviéndolo hacia la derecha e izquierda.

Una vez localizado el elemento que se desea crear en la pantalla, es necesario tocarlo con el dedo índice de la mano contraria para hacer que aparezca delante del usuario, para ello, se debe hacer el gesto de 'señalar', relajando la mano y estirando únicamente el dedo índice. Si el usuario desea crear más de una entidad del mismo u otro tipo, solo tiene que repetir la operación con los elementos deseados.

Cada elemento del lenguaje tiene su propia forma y comportamiento a la hora de asociarse con otros, previniendo de esta manera los errores sintácticos y, algunos errores semánticos, ya que este comportamiento impedirá por ejemplo, que a una instrucción 'mover' se le asocie una referencia a una variable booleana. Todos los elementos del lenguaje se pueden desplazar por la escena agarrándolos al mantener pulsado el gatillo lateral con el dedo anular. Para facilitar la interacción entre ellos y ayudar al usuario, cuando un elemento del lenguaje es arrastrado sobre (a través de) otro con el cual guarda relación y es compatible, se mostrará en el elemento una previsualización de color verde semitransparente que indica el resultado de componer ambos elementos y la cual no aparecerá en caso de que sean incompatibles pese a que sus formas encajen.

Actualmente el lenguaje implementado para controlar el *drone* cuenta con los siguientes elementos:

- **variables**: Están representadas por un cubo pequeño con un objeto puntiagudo encima que puede desplazarse lateralmente para ajustar su valor. En la parte frontal, cada variable tiene un icono que actúa como 'nombre' de dicha variable y sirve para establecer una correspondencia entre las variables y sus referencias. El color de fondo del cubo será rosa en caso de que la variable sea numérica (el selector será de color azul) o azul en caso de que sea booleana (en este caso, el selector será de color rojo o verde según el valor sea falso o verdadero respectivamente).
- **referencias**: Visualmente son unos cilindros horizontales cuyo color e icono coinciden con los de la variable a la que hacen referencia. Su cometido es aportar información a las instrucciones y estructuras de control que cambiará cuando así lo haga el valor de la variable asociada. Para crearlas es necesario agarrar una variable y tirar de ella hasta que vuelva a su posición inicial, creando con ello una referencia a si misma que será la que se intro-

duzca en el hueco cilíndrico de las instrucciones, las cuales solo la aceptarán si es del tipo adecuado, mostrando una previsualización de color verde en caso de serlo.

- **parámetros**: Aportan información cualitativa a las instrucciones, visualmente se representan con un cubo verde con un icono negro en su parte frontal. Pueden ser creadas desde el menú de creación de entidades y éste solo se acoplará a las instrucciones si lo admiten (tienen un hueco cuadrado) y es adecuado (antes de soltarlo en una instrucción, se previsualiza).

- **instrucciones**: Permiten interactuar con el drone, actualmente hay disponibles dos, que permiten desplazar y rotar el drone. Están representadas por un objeto de color gris dividido verticalmente en tres partes diferenciadas, en la inferior, presentan un icono de color negro que indica de qué instrucción se trata, en la central, tienen un hueco circular en el cual se debe introducir una referencia a una variable numérica y, en la superior, hay un hueco cuadrado que sirve para albergar un parámetro, el cual debe ser consistente con la instrucción de la que se trate (por ejemplo, no es posible usar parámetros de rotación en torno a ejes en una instrucción 'mover', pero sí en una 'rotar').

  Para insertar una instrucción, es necesario agarrarla y moverla sobre la instrucción anterior al lugar deseado o, sobre el bloque cyan a la izquierda del programa si se desea insertar al inicio o no hay más instrucciones.

- **estructuras de control**: Son los elementos más grandes visualmente, en su parte izquierda disponen de un hueco circular para introducir una referencia a una variable booleana, la cual, al comprobar su valor durante la ejecución, hará que la ejecución del programa continúe a la derecha del fragmento de color rojo o verde según la variable referenciada almacene el valor falso o verdadero respectivamente.
  Las estructuras de control disponibles son la condición y el bucle, de colores cyan y naranja respectivamente, sobre los cuales hay unas flechas que indican la dirección del flujo de ejecución y que sirven de aclaración respecto a la forma de funcionar de cada una de ellas. En ambos casos, para añadir instrucciones en la rama verdadera, falsa o después de la estructura, es necesario agarrarlas y moverlas cerca del lugar deseado hasta que se previsualize en dicho lugar, momento en el que al soltarla, se introducirá en el lugar de la previsualización.

La forma que tiene el programa de ejecutarse es similar a un depurador, lo que significa que una vez se pulse el botón 'run' del bloque cyan situado en la parte izquierda de la mesa, las instrucciones se van ejecutando una detrás de otra, obteniendo sus valores en el momento de ejecutarse, lo que permite modificar las variables durante la ejecución y ver cómo el programa se continúa ejecutando de acuerdo a los nuevos valores. También es posible modificar las secuencias de

instrucciones mientras el programa se ejecuta, aunque solo aquellas que no hayan empezado a ejecutarse con anterioridad, esto significa que se puede modificar el interior de un bucle si la ejecución aún no ha llegado a él, pero no mientras sus instrucciones estén siendo ejecutadas.

4.2. Crear escenas con programas predefinidos

VIRTO, además de permitir la creación y ejecución de programas a través de la realidad virtual, también permite que, aquellos usuarios que conozcan el lenguaje HTML, puedan utilizar los componentes desarrollados para personalizar las escenas de realidad virtual, cambiando la disposición de los elementos u creando otros nuevos (añadiendo más programas y/o *drones* por ejemplo). Es por ello que esta subsección se dedica a explicar cómo se crean programas para trabajar después con ellos en realidad virtual.

Para crear una escena, se recomienda copiar el código HTML de otra escena ya existente para reducir el tiempo invertido en introducir las URL de los componentes y recursos que se vayan a emplear. En el código 4.1 se puede ver la estructura básica del fichero HTML antes de comenzar a añadir elementos a la escena y en él se puede apreciar que los recursos empleados por la escena (a parte de los scripts), están ubicados dentro de la etiqueta *'<a-assets>'* y se referencian a través de la etiqueta *'<a-asset-item>'*, aunque se recomienda usar otras etiquetas también soportadas como *'<img>'* o *'<audio>'* en función del tipo de contenido que se desee cargar, para facilitar la lectura del código y optimizar su carga cuando sea posible. En lo que concierne al prototipo, las imágenes pueden ser sustituidas por otras (por ejemplo, para usar otros iconos de variable), mientras que los modelos 3D no deben ser reemplazados, ya que algunos componentes están muy ligados a ellos y modificarlos podría ocasionar problemas visuales y/o de funcionamiento.

A continuación será necesario incluir en la escena una cámara y los controles para permitir al usuario interactuar con las entidades que se añadan posteriormente, no es estrictamente necesario a la hora de crear una escena arbitraria porque *A-Frame* los añade automáticamente si no lo ha hecho el programador (como ocurre en el código 4.1). En este caso, se recomienda usar el componente *multidevice* desarrollado para este trabajo, el cual no solo añade la cámara sino que detecta el dispositivo que se está utilizando y adecua los controles a él, configurando los componentes necesarios para poder interactuar con la escena independientemente de que se acceda desde PC o desde un visor de realidad virtual.

Código 4.1: HTML básico para crear una escena de realidad virtual con *A-Frame*, en el lugar indicado se deben ubicar las etiquetas que definirán su contenido.

```
<!DOCTYPE html>
<html>
  <head>
    <script src="https://aframe.io/releases/1.3.0/aframe.min.js"></
        script>
    <!-- resto de scripts -->
  </head>
  <body>
    <a-scene>
      <a-assets>
        <!-- texturas y modelos 3D utilizados -->
        <a-asset-item id="id_del_modelo" src="
            URL_del_fichero_obj_o_gltf"></a-asset-item>
        <img id="id_de_la_textura" src="URL_de_la_textura"></img>
      </a-assets>

      <!--  En este espacio se define la escena -->

    </a-scene>
  </body>
</html>
```

Fuente: Elaboración propia

Una vez configurada la cámara, se recomienda añadir a la escena al menos un suelo para que el usuario no se encuentre en el vacío, para lo cual puede emplearse el componente *aframe-environment-component*, que proporciona una forma rápida y sencilla de dotar a la escena de un 'entorno' visualmente agradable en el cual comenzar a añadir elementos, que en HTML estarán representados mediante la etiqueta *'<a-entity>'*.

Las dos entidades imprescindibles para poder componer el entorno de desarrollo son, el *drone* y el IDE, en el cual se introducirán los programas que controlan el *drone*. Para insertar estos elementos en la escena, hay que crear dos elementos *'<a-entity>'* y asociarles respectivamente los componentes **drone** y **ide**, lo cual se consigue añadiendo al elemento HTML un atributo con su nombre.

Para añadir un programa, basta con introducir dentro del elemento con el atributo **ide**, otro elemento con el atributo **program**, al cual se puede dar un nombre a través de su valor. En el código 4.2 se puede apreciar el uso de los tres componentes mencionados anteriormente para crear una escena con un *drone* y un programa llamado 'main', en el cual se introducirán las variables e instrucciones que lo formarán.

Código 4.2: Código de una escena *A-Frame* creada usando los componentes de ViRto que muestra un programa llamado 'example' vacío. Si se añaden los scripts y recursos necesarios, al ejecutar el código en un navegador se podrá observar cómo el componente 'program' crea en su interior dos entidades con los componentes 'scope' y 'code'.

```
<a-scene>
  <a-assets> ... </a-assets>
  <a-entity drone position="0 2 -2></a-entity>
  <a-entity ide>
    <a-entity program="name:example"> ... </a-entity>
  </a-entity>
  <a-entity multidevice></a-entity>
</a-scene>
```

Fuente: Elaboración propia

A la hora de construir el programa en HTML, este prototipo separa las variables de las instrucciones utilizando dos elementos diferentes, que necesitan tener los componentes **scope** y **code** asociados a ellos respectivamente.

Dentro del 'scope' se introducirá un elemento por cada variable que se quiera declarar, el cual tendrá el componente **variable**, además de un identificador a través del atributo **id**, que servirá para referir a dicha variable en el código y no será visible en la escena. Por otra parte, dentro del elemento con el componente 'code' se añadirán, por orden, las instrucciones del programa, creando elementos y asociándoles los componentes **instruction** si se trata de una instrucción para controlar el *drone*, **instruction-conditional** si se desea introducir una decisión en el programa o, **instruction-loop** si se desea introducir un bucle.

Las estructuras de control (decisiones y bucles) tienen una sintaxis especial, ya que la decisión requiere que en su interior haya dos elementos con el componente **code** y las clases (atributo *class*) *branchTrue* y *branchFalse* para diferenciar qué bloque de código se ejecutará en función del valor de la variable booleana indicada a través del atributo **instruction-conditional**. El bucle, en cambio, es más sencillo de escribir y solo requiere en su interior un elemento con el componente **code** asociado y que se haya indicado el nombre de una variable booleana a través de su atributo **instruction-loop**.

Con los elementos anteriores se puede construir sintácticamente un programa, pero falta la parte semántica, es decir, su significado y la relación entre las piezas que lo forman, para lo cual es necesario manipular los atributos de los componentes, a los cuales se accede a través del valor de los atributos que tienen sus mismos nombres como se hace en el código 4.3, en el cual se asocian las variables booleanas 'flor' y 'perro' a las instrucciones condicional y bucle respectivamente, además de las variables 'luna' y 'gato' a las instrucciones de 'mover' y 'rotar'. El programa finaliza cuando la variable 'perro' almacene el valor 'falso'.

En la figura 4.4 se aprecia visualmente cómo ViRto reconstruye visualmente el código referido anteriormente.

Código 4.3: Fragmento de HTML que muestra un programa ViRto que mueve el drone hacia arriba haciendo zig-zag hacia la derecha o hacia la izquierda según el valor de la variable 'flor'. En este ejemplo se ha omitido la importación de los componentes ViRto y sus correspondientes recursos, siendo ambos necesarios para poder utilizar este código.

```
<a-scene>
  <a-assets> ... </a-assets>
  <a-entity drone position="-1  3 -2"></a-entity>
  <a-entity ide position="0 1 0">
    <a-entity program="name:main">
      <a-entity scope>
        <a-entity id="luna" variable="type:integer;value:2;min:0;
            max:5;icon:#icon_moon"></a-entity>
        <a-entity id="gato" variable="type:integer;value:45;min:0;
            max:90;icon:#icon_cat"></a-entity>
        <a-entity id="perro" variable="type:boolean;value:true;icon:#
            icon_dog"></a-entity>
        <a-entity id="flor" variable="type:boolean;value:false;icon:#
            icon_flower"></a-entity>
      </a-entity><!-- end scope -->
      <a-entity code>
        <a-entity instruction-loop="reference:#perro">
          <a-entity code>
            <a-entity instruction-conditional="reference:#flor">
                <a-entity class="branchTrue" code>
                  <a-entity instruction="function:move;
                      parameter:right;reference:#luna"></a-entity>

                </a-entity> <!-- end branch true -->
                <a-entity class="branchFalse" code>
                  <a-entity instruction="function:move;parameter:left
                      ;reference:#luna"></a-entity>
                </a-entity> <!-- end branch false -->
            </a-entity> <!-- end conditional -->
            <a-entity instruction="function:rotate;parameter:zaxis;
                reference:#gato"></a-entity>
          </a-entity> <!-- end loop code -->
        </a-entity> <!-- end loop -->
      </a-entity> <!-- end program code -->
    </a-entity> <!-- end program -->
  </a-entity> <!-- end ide -->
  <a-entity multidevice position="0 0 2"></a-entity>
</a-scene>
```

Fuente: Elaboración propia

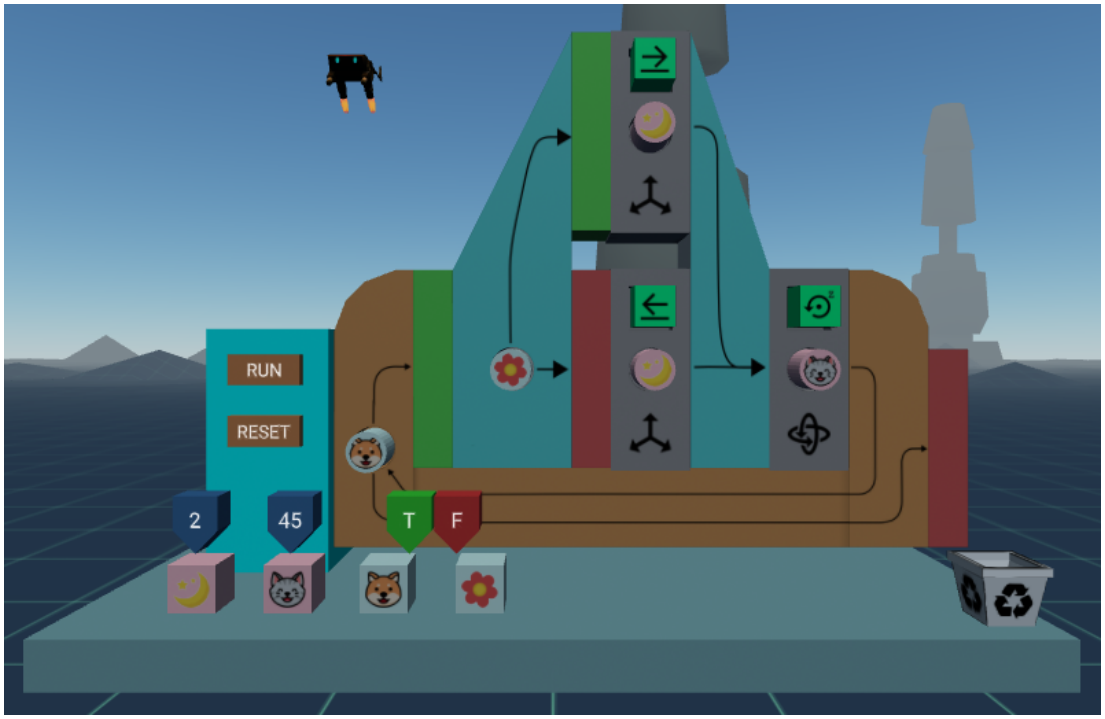


Figura 4.4: Captura de pantalla del entorno de desarrollo obtenido a partir del código 4.3.

Fuente: *elaboración propia*

5

Descripción informática

En este capitulo se comenzará explicando la relación de componentes de ViRto y el funcionamiento del algoritmo del intérprete-depurador que ejecuta los programas, para terminar describiendo la función de cada componente dentro del prototipo.

5.1. Relación entre componentes

Como ViRto se ha construido sobre *A-Frame*, hereda su arquitectura *entidad-componente-sistema*, correspondiendo en este caso, las entidades con la parte física de los elementos del lenguaje y, los componentes, el código que establece cómo se deben comportar y relacionar dichos elementos. Este proyecto no hace uso de sistemas *A-Frame* al dotar a los componentes de comportamiento y capacidad de interacción entre ellos.

Los componentes están pensados para ser usados fácilmente a la vez que permitir su serialización, por ello, no es imprescindible crear todas las entidades de la escena y asociarles los correspondientes componentes, ya que éstos al inicializarse, comprobaran las entidades que tienen anidadas y, en caso de no encontrar una que sea imprescindible para su funcionamiento, la creará automáticamente, así, en el código 5.1 se pueden ver las entidades mínimas que hay que utilizar (y sus componentes asociados) para poder empezar a crear programas.

Los componentes por tanto, tienen una fuerte relación entre ellos que permite,

Código 5.1: Estructura básica de entidades y componentes para poder programar con VIRTO.

```
<a-scene>
  <a-entity drone></a-entity>
  <a-entity ide>
    <a-entity program="name:main"></a-entity>
  </a-entity>
  <a-entity multidevice></a-entity>
</a-scene>
```

Fuente: Elaboración propia

como se ha visto en el código 5.1, desplegar todo el entorno de desarrollo a partir de cuatro componentes, además de facilitar el guardado y la carga de programas, debido a que, como al inicializarse consultan las entidades de su vecindad y los componentes que tienen asociados, pueden reconstruir su estado en memoria a partir del DOM.

En la figura 5.1 se puede ver un diagrama de clases representando las relaciones existentes entre los componentes de VIRTO, puede apreciarse en él, como hay dos grupos principales de relaciones de composición y agregación, uno en torno a '*instruction-base*' y el segundo, pendiendo de '*multidevice*', donde las composiciones indican los componentes imprescindibles para el funcionamiento del entorno, y que por tanto, se crearán automáticamente si no se encuentran en el DOM y, las agregaciones, los componentes que, aunque necesarios para crear un programa válido, no comprometen el funcionamiento de los componentes que los agregan.

Debido al desarrollo incremental llevado a cabo, en esta etapa aún no se han podido determinar con claridad los atributos y operaciones comunes a todas las instrucciones, al faltar todavía algunas por diseñar e incluir, por lo que el componente '*instruction-base*', etiquetado como << *abstract* >> no se ha implementado todavía, pero se espera que esté disponible más adelante cuando sus características se hayan definido con claridad. Relativo a este componente, cabe destacar un detalle presente en el diagrama y en la implementación, y es la posibilidad de anidar componentes '*code*' mediante las instrucciones condicional y bucle, con lo que se crea una sintaxis recursiva que aporta un gran poder de expresión al lenguaje.

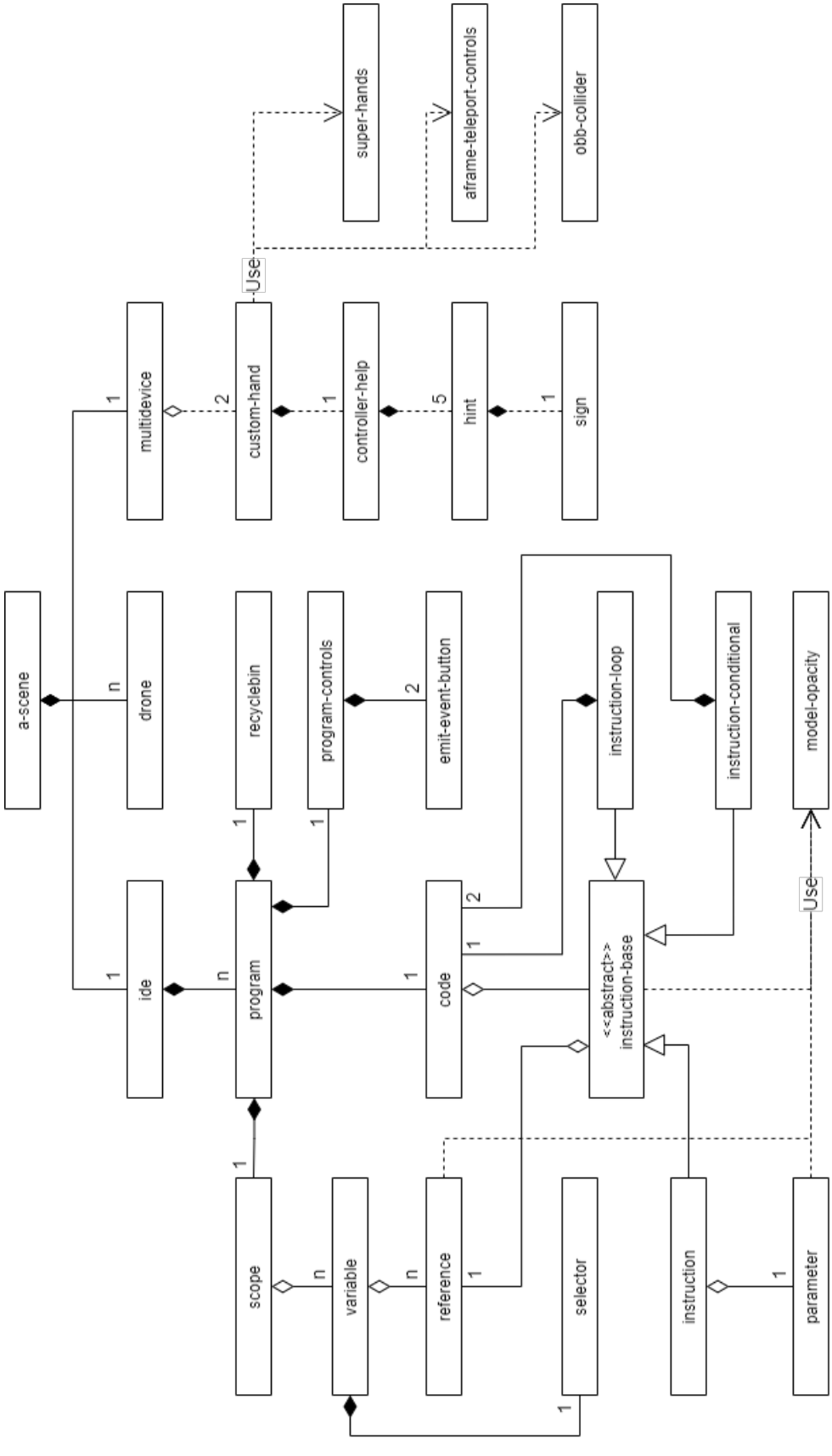


Figura 5.1: Diagrama de clases que representa las relaciones entre los componentes desarrollados. Las composiciones y agregaciones se materializan a través de la jerarquía del DOM y representan los componentes imprescindibles y opcionales respectivamente para el correcto funcionamiento de la escena.

Fuente: *elaboración propia*

5.2. Implementación del intérprete-depurador

El propósito de crear un intérprete-depurador en lugar de un intérprete es dar a los usuarios la posibilidad de manipular las variables en tiempo de ejecución y que puedan apreciar los cambios en tiempo real, además de servir de preámbulo para la inclusión de operadores aritméticos. Durante la iteración 3 del desarrollo, debido a que únicamente se podían construir programas secuenciales, fue suficiente con implementar el algoritmo 1, que, pese a utilizar la función *`setTimeout()`* para organizar la ejecución de las instrucciones, las imponía un tiempo fijo para ejecutarse independientemente de su tipo o sus argumentos.

Tras implementar las instrucciones condicionales y bucles, se refactorizó la sintaxis del lenguaje para incluir los ámbitos de los programas y, también, crear el componente '*code*', el cual representa fragmentos de programas secuenciales.

Este componente '*code*' se ha utilizado para modelar las estructuras de control y la sección de código del programa, lo que permite tratar el interior de las estructuras de control como programas en sí mismos, permitiendo el anidamiento de estas estructuras, logrando un lenguaje más expresivo que obligó a rediseñar la forma en que se ejecutan los programas. Fruto de este rediseño, fue la consideración del componente '*code*' como un elemento ejecutable más (como ya lo eran los programas y las instrucciones y estructuras de control), de forma que los elementos ejecutables delegan su ejecución de manera asíncrona asíncrona en sus descendientes en el DOM.

Al ejecutar los programas de esta manera, se elimina la necesidad de prefijar o calcular los tiempos de ejecución para introducir *timeouts*, a la vez que se facilita la inclusión de nuevas instrucciones con tiempos de ejecución diversos, únicamente cumpliendo con el requisito de implementar su ejecución de manera similar a la mostrada en el código 5.2.

El código antes mencionado, hace uso de las palabras reservadas **async** y **await**, introducidas en *ECMAScript 8* (2017) con el objetivo de facilitar la programación con promesas, marcando con *async*[1] una función que retorna una promesa y con *await*[2], el código que se debe ejecutar cuando se resuelva la promesa (anteriormente había que usar el método *.then()* de la promesa).

En la figura 5.2 puede verse un diagrama de flujo que refleja de forma simplificada cómo se ejecutan de forma recursiva los componentes '*code*', representados en la figura a través del texto 'Bloque de código'. Si se analiza con detalle el diagrama y se tiene en cuenta que la ejecución de las instrucciones no es instantánea, sino que lleva unos milisegundos, se puede inferir cómo los valores de las referencias se consultan '*Just In Time*', dotando al intérprete de características propias

[1] `https://developer.mozilla.org/docs/Web/JavaScript/Reference/Statements/async_function`

[2] `https://developer.mozilla.org/docs/Web/JavaScript/Reference/Operators/await`

Código 5.2: Código de la función exec en el componente '*code*'. Esta función es la encargada de ejecutar de forma asíncrona y ordenada cada una de las instrucciones que contiene en su interior la entidad que tiene este componente.

```
exec: async function(){
  return new Promise( async (resolve,reject)=>{
    try{
      for(instruction of this.instructions){
        await instruction.exec();
      }
    }catch(e){
      reject(e);
    }
      resolve();
  });
}
```

Fuente: Elaboración propia

de un depurador.

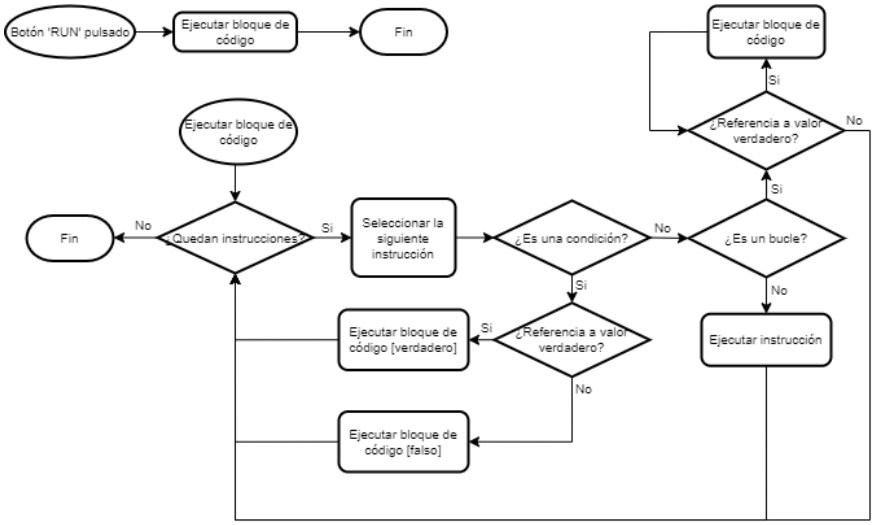


Figura 5.2: Diagrama de flujo del intérprete-depurador, nótese la relevancia del 'bloque de código' (componente '*code*'), que permite definir las estructuras de control como una decisión seguida de la ejecución de uno de estos 'bloques de código'.

Fuente: *elaboración propia*

5.3. Descripción de componentes

En esta sección se describen con mayor detalle cada uno de los componentes desarrollados, así como se muestra el aspecto visual que imponen a las entidades a las que pertenecen.

- **aabb-collider**: Es un detector de colisiones entre una esfera de radio configurable y un cubo con sus ejes alineados con los del mundo ('*aabb*' proviene del inglés '*Axis Aligned Bounding Box*', que se puede traducir como 'cajas de colisión alineadas con los ejes'). Este enfoque simplifica los cálculos a costa de no dar buenos resultados si los objetos no son cúbicos o están rotados. Este componente es una modificación de otro llamado 'sphere-collider' desarrollado por el usuario de GitHub *n5ro*[3], siendo sus parámetros idénticos para facilitar la experimentación con otros tipos de colisionadores.

- **assets-manager**: Este sistema[4] para A-Frame fue diseñado para tratar de ligar los componentes a los recursos que necesitan para funcionar, de manera que cuando dichos componentes se inicialicen, soliciten a *assets-manager* la carga de sus recursos asociados sin tener que repetir en todos los componentes el código que lo hace posible, reduciendo el código duplicado y facilitando la legibilidad del mismo.

  Pese a la gran ayuda que brinda este sistema, se dejó de usar porque se comprobó que importar los recursos durante la inicialización de los componentes incrementa sensiblemente el tiempo de carga de la escena al no realizarse totalmente en paralelo, como ocurre si todos los recursos se encuentran desde el inicio en el elemento '<*a-assets*>'.

- **box-collider**: Es un colisionador creado desde cero para explorar el funcionamiento de este tipo de componentes. Tiene una estructura diferente al resto de colisionadores implementados, que están basados en '*sphere-collider*' y por tanto, no los puede reemplazar directamente. La gran ventaja que presenta este componente es que detecta correctamente las colisiones entre una esfera y una caja incluso cuando esta última esté rotada en torno a uno o más de sus ejes, pero por contra, es ligeramente menos eficiente que otros colisionadores.

- **code**: Componente sin aspecto gráfico que agrupa y organiza las instrucciones existentes en su interior, entendiendo por instrucciones aquellos elementos hijos que tengan asociado un componente cuyo nombre empiece por '*instruction*'. De forma dinámica, si se añaden o eliminan instrucciones, se actualizan las posiciones de todas las demás para que no se solapen ni haya huecos entre ellas.

[3]https://github.com/n5ro/aframe-extras/blob/master/src/misc/sphere-collider.js
[4]https://aframe.io/docs/1.3.0/core/systems.html

Este componente representa un 'bloque de código' y es posible ejecutarlo a través de su función '*exec()*', la cual invocará a su vez la función '*exec()*' de cada una de las instrucciones que contiene, realizando así una ejecución secuencial.

- **controller-help**: Su propósito es el de mostrar una serie de carteles orientados hacia el usuario que informan de las acciones que puede llevar a cabo si presiona botones concretos de los controladores, los cuales se indican por medio de gruesas líneas rojas.

- **custom-hand**: Es una agregación de varios componentes, entre ellos '*hand-controls*' y '*super-hands*', que encapsula su configuración y añade otras funcionalidades como mostrar/ocultar el menú de creación de entidades al pulsar el *joystick* o permitir tocar cosas con la punta del dedo índice cuando se hace el gesto de señalar. Este componente es el encargado de mostrar y ocultar el menú de creación de entidades provisto por '*hand-menu*'

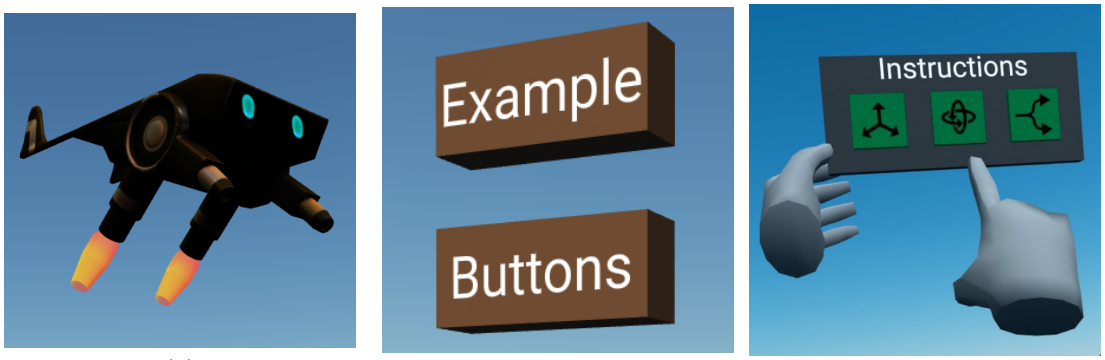
- **drone**: Este componente es el responsable de hacer moverse el modelo 3D del *drone* y puede ser modificado añadiendo funciones que representen nuevas instrucciones que se deseen introducir en el entorno de programación (por ejemplo, encender una luz o disparar un proyectil), ya que cada instrucción podrá estar directamente relacionada con una función de este componente, a la cual se le pasan los valores de sus parámetros como argumentos.

  En la figura 5.3-a se puede ver el modelo 3D utilizado en `ViRTo`, el cual ha sido diseñado por *Renafox* y publicado en *Sketchfab*[5] bajo la licencia CC BY-NC 4.0.

- **emit-event-button**: Su propósito es convertir cualquier entidad en un botón, el cual, al pulsarse con el dedo índice o hacer clic sobre él, emite un evento a otra entidad que se indique, para que dicha entidad pueda reaccionar en consecuencia. Es un componente sencillo pero que permite interactuar con el usuario de manera sencilla a través de la programación orientada a eventos. Su aspecto es rectangular, mientras que su texto se puede configurar como muestra la figura 5.3-b

- **hand-menu**: Componente que presenta un menú visual al usuario a través del cual puede añadir entidades a la escena. Está dividido en 'categorías' y 'elementos'; se puede navegar entre las categorías y los elementos dentro de ellas a través de las funciones '*increaseCategory()*', '*decreaseCategory()*', '*increaseFirstItem()*' y '*decreaseFirstItem()*, mientras que para crear una entidad del tipo representado por un elemento, basta con tocar dicho elemento, como se aprecia en la figura 5.3-c. Las funciones de navegación están asociadas a los movimientos vertical y horizontal del *joystick* dentro del componente '*custom-hand*', pero las funciones se han creado para poder implementar otros tipos de navegación.

[5] `https://sketchfab.com/3d-models/drone-ce248709dea64ec1844e8dd9b614f7c0`



(a) (b) (c)

Figura 5.3: Representación gráfica de los componentes '*drone*' (a), 'emit-event-button' (b) y 'hand-menu' (c)

- **hint**: Permite crear un cartel informativo que esté siempre orientado hacia el usuario para facilitar su lectura y además, una línea gruesa que une el cartel con otro punto de la escena y que sirve como ayuda visual para asociar el contenido del cartel con otra entidad, facilitando así añadir información a las escenas de manera sencilla y amigable para el usuario. En la figura 5.4-a se muestra el aspecto visual de este componente.

- **ide**: Componente sin representación gráfica que agrupa en su interior programas, asegurando la unicidad de sus nombres, a la vez que provee de una función para crear un programa nuevo y asignarle un nombre, retornando su elemento HTML o '*null*' en caso de que el nombre indicado ya esté en uso.

- **instruction-conditional**: Este componente representa una instrucción condicional, presente en otros lenguajes mediante las palabras reservadas '*if*' y '*else*'. Puede estar emparentada con elementos con los componentes '*program*' si aún no se ha introducido en el código o '*code*' en caso contrario.

  Internamente, la condición tiene tres elementos, una entidad con el componente '*reference*' asociado, que indica la variable cuyo valor se tendrá en cuenta para ejecutar una rama u otra y otros dos elementos con el componente '*code*' y cada uno de ellos con las clases 'branchTrue' y 'branchFalse' en función de si dicho bloque de código representa las instrucciones a ejecutar en caso de que la variable referenciada tenga el valor verdadero o falso. Puede apreciarse la disposición de los elementos que conforman la instrucción condicional en la figura 5.4-b

- **instruction-loop**: Representa una instrucción repetitiva de tipo 'mientras', presente en muchos lenguajes de programación bajo la palabra reservada '*while*', la cual, evalúa primero el valor de una variable o expresión booleana y solo en caso de ser verdadera, ejecuta el bloque de código de su interior

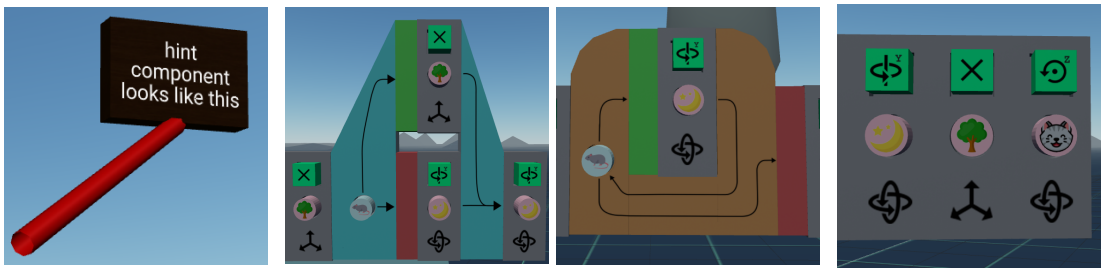
antes de volver a evaluar dicha variable o expresión hasta que sea falsa y termine su ejecución pasando a la instrucción posterior.

La entidad que tiene este componente asociado, tiene dos elementos hijo, el primero es una referencia a una variable (componente '*reference*'), la cual se crea automáticamente si se indica a través del parámetro 'reference' de este componente y, el segundo, es un bloque de código con las instrucciones que se ejecutarán de forma repetitiva (componente '*code*'). en la figura 5.4-c se puede ver el aspecto que presenta un bucle con todos los elementos previamente descritos.

Debido a que el lenguaje no tiene instrucciones para asignar y/o modificar los valores de las variables, los bucles cuya variable inicialmente almacene el valor 'verdadero', ejecutarán indefinidamente su bloque de código hasta que el usuario modifique el valor de dicha variable a 'falso' usando el selector de valores que esta tiene en su parte superior.

- **instruction**: Este componente representa una instrucción arbitraria que recibe un argumento cualitativo (a través del parámetro 'parameter' del componente o teniendo un hijo con el componente homónimo asociado) y una referencia a una variable numérica ('`type:integer`'). El cometido de la instrucción se establece a través de su parámetro 'function' y cuando se ejecute la instrucción invocando a su función '*exec()*', invocará a la función del drone con el mismo nombre que se haya indicado a través del parámetro 'function', el cual también vale para determinar el icono que se muestra en la parte inferior del modelo 3D, como puede apreciarse en la figura 5.4-d.

  El propósito de este componente, es abstraer la llamada a las funciones del *drone*, proporcionando una interfaz uniforme a través de la cual el usuario puede invocar dichas funciones y establecer sus argumentos de manera interactiva, además de desacoplar el *drone* de las instrucciones que lo gobiernan y poder modificar e incluso reemplazar cualquiera de las partes sin tener que hacer cambios profundos en el código de cualquiera de los componentes.



(a) (b) (c) (d)

Figura 5.4: Representación gráfica de los componentes 'hint' (a), 'instruction-conditional' (b), 'instruction-loop' (c) e 'instruction' (d)

- **multidevice**: Configura la entidad a la que se asocia con los componentes necesarios para poder interactuar con la escena tanto si el usuario accede

desde un PC como si lo hace desde un visor de realidad virtual. Para ello, comprueba el tipo de dispositivo que ha cargado la escena a través de la función '*AFRAME.utils.device.isMobileVR()*' y, en función de su valor de retorno establece una altura fija para la cámara y configura los controles de ratón y teclado o, por el contrario, deja que sea la API de '*WebXR*' la que establezca la altura de la cámara y crea dos entidades con el componente 'custom-hands' para poder interactuar con la escena por medio de los controladores del visor de realidad virtual.

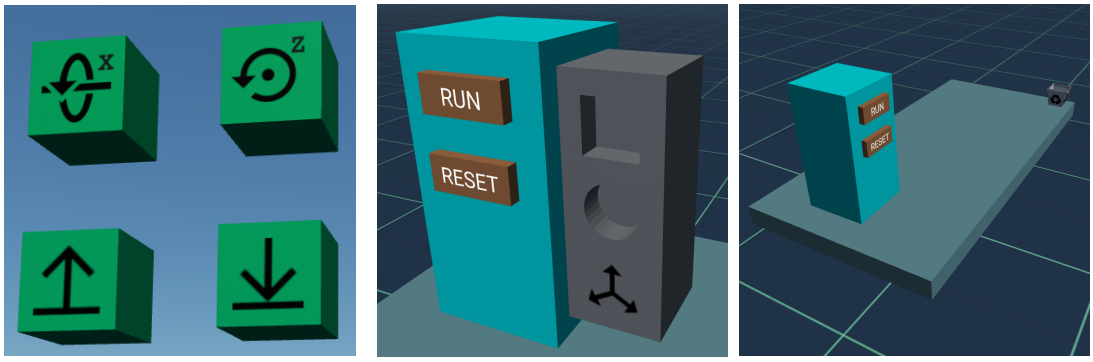
- **obb-collider**: Es un detector de colisiones entre una esfera y un cubo con orientación arbitraria, es idéntico a 'box-collider', con la diferencia de ser intercambiable por 'sphere-collider' y 'aabb-collider' ya que comparten la misma estructura y parámetros.

- **parameter**: Este componente permite representar información cualitativa que pueda ser pasada a una instrucción con el objetivo de parametrizar su ejecución, teniendo dos parámetros llamados 'type' y 'function', indicando el primero su valor (que debe poder ser interpretado por el *drone*) y a partir del cual se buscará su textura en los recursos (pueden verse varias de estas texturas aplicadas en la figura 5.5-a) y, el segundo, la función compatible con este parámetro, de forma que se puedan evitar errores sintácticos al impedir que un parámetro se pueda asociar a una función en la que no tiene sentido (por ejemplo, impidiendo mover el *drone* 'alrededor del eje X', cuando este parámetro tiene más sentido si se asigna a una instrucción de rotación).

- **program-controls**: Da forma al bloque cyan ubicado en la parte izquierda del programa y configura la entidad para previsualizar y añadir instrucciones al principio del programa, además de añadir los botones 'run' y 'reset' como se aprecia en la figura 5.5-b, para iniciar la ejecución del programa y restaurar la posición del *drone* respectivamente.

- **program**: Es el componente encargado de dar forma a la mesa sobre la cual reposan los elementos del programa y ejecutar el bloque de código cuando recibe un evento 'run'. Este componente está parametrizado con el nombre del programa que representa y lo abstrae separando las variables del código que las emplea mediante dos entidades con los componentes 'scope' y 'code' respectivamente. Adicionalmente, este componente también crea una entidad con el componente 'program-controls' para controlar su ejecución y otra con el componente 'recyclebin' para permitir el borrado de entidades. En la figura 5.5-c se puede ver el aspecto que presenta un programa con sus elementos fundamentales, una entidad con el componente 'recyclebin' y otra con 'program-controls'.

- **recyclebin**: Componente que transforma la entidad a la que se le asocia un modelo 3D de una papelera, mediante la cual, al agarrar una entidad



(a) (b) (c)

Figura 5.5: Representación gráfica de los componentes 'parameter' (a), 'program-controls' (b) y 'program' (c)

y arrastrar la mano que la sostiene hasta la papelera, permite eliminarla. Es de especial utilidad a la hora de limpiar el entorno de programación y destruir aquellas entidades que no se van a utilizar más pero que, al estar flotando por la escena, pueden resultar molestas para el usuario. El aspecto del modelo 3D antes mencionado puede verse en la figura 5.6-a

- **reference**: Complementando al componente 'variable', proporciona a las instrucciones una interfaz a través de la cual obtener o alterar el valor de la variable a la cual hace referencia a través de las funciones '*get()*' y '*set(value)*'. Gracias a la abstracción que aporta este componente, es posible tener una representación gráfica de los usos de una variable en concreto, ya que visualmente, la referencia a una variable copia su color (diferente para cada tipo de dato) y su icono. En la parte superior de la figura 5.6-b se pueden ver dos referencias a variables numéricas y en la inferior, a variables booleanas.

- **scope**: La función de este componente es agrupar las variables que forman parte del ámbito de un programa y, aunque gráficamente es invisible, se encarga de distribuir las variables que penden de ella de manera uniforme y ajustando el espacio entre ellas para evitar que se coloquen fuera de la mesa del programa.

- **selector**: Componente creado para modificar los valores de las variables al ser desplazado a lo largo de un eje invisible. A nivel gráfico, informa con un texto del valor de la variable y con su color de fondo del tipo de dato de la misma a la que se asocia (azul si es numérica, rojo si es booleana y tiene el valor 'falso' o, verde en caso de ser también booleana pero almacena el valor 'verdadero'), tal y como se puede apreciar en la figura 5.3-c

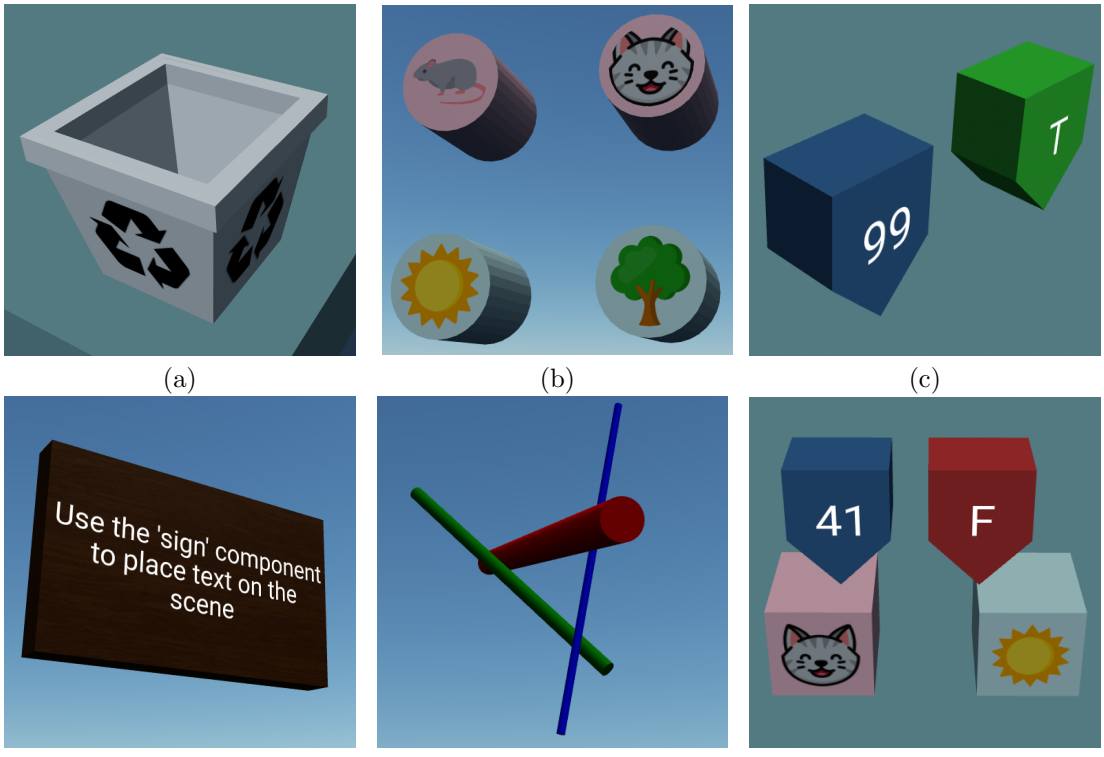
  A través de sus parámetros se puede configurar para evitar que se selecciones valores fuera de un determinado rango, el tipo de la variable, su valor (para

mantener ambos componentes sincronizados) y la longitud a lo largo de la cual se podrá desplazar el selector.

- **sign**: Este componente facilita la creación de carteles informativos como el mostrado en la figura 5.6-d, al generar visualmente una caja delgada, cuyo ancho y alto es configurable, así como el texto que mostrará en uno de sus lados. El componente hace uso de una textura de madera que aplica a la caja, así como permite determinar el ancho de las líneas, a través del cual se puede regular el tamaño de las letras.

- **thick-line**: La función de este componente es generar un cilindro con un diámetro configurable, cuyos extremos se encuentren en las coordenadas indicadas por el programador, con el objetivo de representar líneas rectas tridimensionales de grosor configurable como las que se ven en la figura 5.6-e.

- **variable**: Este componente se encarga de almacenar valores que puedan ser utilizados por el programa y facilitar que estos valores puedan ser modificados tanto por el usuario como por el programa. Por diseño, la variable se ubica dentro del elemento con el componente 'scope' y no se puede utilizar en las instrucciones, para lo cual será necesario agarrar la variable y tirar de ella hasta que se genere una referencia asociada a sí misma que sí que se podrá utilizar en el programa tantas veces como sea necesario, ya que se podrán generar más volviendo a tirar de la variable. Para visualizar y modificar el valor de la variable, este componente coopera con el componente 'selector' descrito anteriormente para adoptar el aspecto que se muestra en la figura 5.6-f.



(a) (b) (c)

(d) (e) (f)

Figura 5.6: Representación gráfica de los componentes 'recyclebin' (a), 'reference' (b), 'selector' (c), 'sign' (d), 'thick-line' (e) y 'variable' (f)

6
Validación

Este capítulo recoge la descripción del experimento realizado para validar prototipo desarrollado como entorno de programación en realidad virtual, así como los resultados obtenidos.

6.1. Descripción del experimento

El experimento se ha diseñado para poder ser realizado por cualquier persona con conocimientos mínimos de programación (debe conocer al menos el paradigma de programación estructurada), sin necesidad de tener experiencia con dispositivos de realidad virtual, por ello, previo a la realización del experimento, se ha previsto preguntar a los participantes si desean realizar un pequeño tutorial de uso del visor de realidad virtual, para tener la certeza de que al empezar el experimento, todos ellos tienen un conocimiento similar del funcionamiento del visor con el que lo realizarán.

El experimento está dividida en tres partes, las cuales se detallan a continuación:

1. **Entrenamiento**: El participante aprenderá a utilizar la herramienta a través de tres escenas, en cada una de las cuales permanecerá cinco minutos y, las cuales están diseñadas para presentar las características del prototipo de manera incremental. Durante esta parte, el experimentador

podrá resolver las cuestiones que le plantee el participante, siempre que no resulten excesivas y traten sobre aspectos del prototipo que no hayan quedado suficientemente explicadas a través de los carteles presentes en las escenas.

En la primera escena, se presenta al participante un ejemplo de programa secuencial, el cual podrá ejecutar y modificar a voluntad para familiarizarse con el método de programación, para lo cual la escena contiene también carteles informativos. La segunda escena difiere de la anterior en que el programa de ejemplo contiene un bucle y un condicional, con el objetivo de que el participante conozca su existencia, así como la de las variables booleanas que los controlan y aprenda a trabajar con ellas.

La tercera y última escena de esta parte, presenta un programa completamente vacío, con el propósito de forzar al participante a crear instrucciones, variables y parámetros con los que crear cualquier programa que desee, de manera que al terminar el tiempo asignado, haya usado todas las características del lenguaje. A diferencia de las dos escenas anteriores, esta escena no cuenta con los carteles informativos, de manera que el participante debe hacer uso de lo aprendido en las dos primeras escenas para desenvolverse.

Para facilitar su trabajo, se pone a disposición de los experimentadores una guía del experimentador que puede ser consultada en el Apéndice A, la cual cubre aspectos importantes del funcionamiento del prototipo de manera que pueda ayudar adecuadamente al participante durante este entrenamiento.

2. **Evaluación**: El participante dispondrá de veinte minutos para implementar un programa que se le indica a través de un cartel que verá al entrar en el modo de realidad virtual, junto con dos paneles que muestran todos los iconos y objetos que puede encontrarse junto con pequeños carteles de referencia.

   Esta parte se compone de una única escena, en la cual el usuario aparece de espaldas al entorno de programación, en el cual deberá implementar un programa que realice las tareas indicadas en el cartel. El tiempo se empieza a contar desde el momento en el que el usuario se de la vuelta para empezar a programar, dejando así un margen de tiempo para que éste haga una primera lectura detallada del programa que se le pide implementar. El experimentador no podrá responder ninguna duda referente al funcionamiento del prototipo, pero si, en caso de que se produzca una situación anómala, como puede ser la aparición de un aviso de batería baja, el bloqueo de la escena o la imposibilidad de realizar ciertas acciones.

   El experimentador indicará al participante que debe avisarle cuando alcance un hito concreto a lo largo del desarrollo así como cuando considere que el programa que ha realizado se corresponde con la descripción proporcionada, en este sentido, cualquier programa que realice la secuencia de instrucciones descrita se considerará válido. En caso de que el participante agote los veinte

minutos previstos para esta tarea, se le preguntará si desea continuar y, en caso afirmativo, se le informará cada cinco minutos del tiempo transcurrido.

3. **Cuestionario**: Se realizará una entrevista al participante, cuyas preguntas se recogen en el Apéndice B y a través de las cuales se pretende obtener su impresión sobre el prototipo y el experimento propuesto, así como su punto de vista acerca de la idea de programar en entornos virtuales. Esta parte no tiene límite de tiempo, finalizando junto a ella el experimento cuando se haya contestado la última pregunta.

Con el objetivo de facilitar la realización del experimento, se ha creado una página web[1] que recoge las partes y tiempos de cada parte del experimento, así como un documento de guía para el experimentador y las preguntas que debe realizar durante la entrevista.

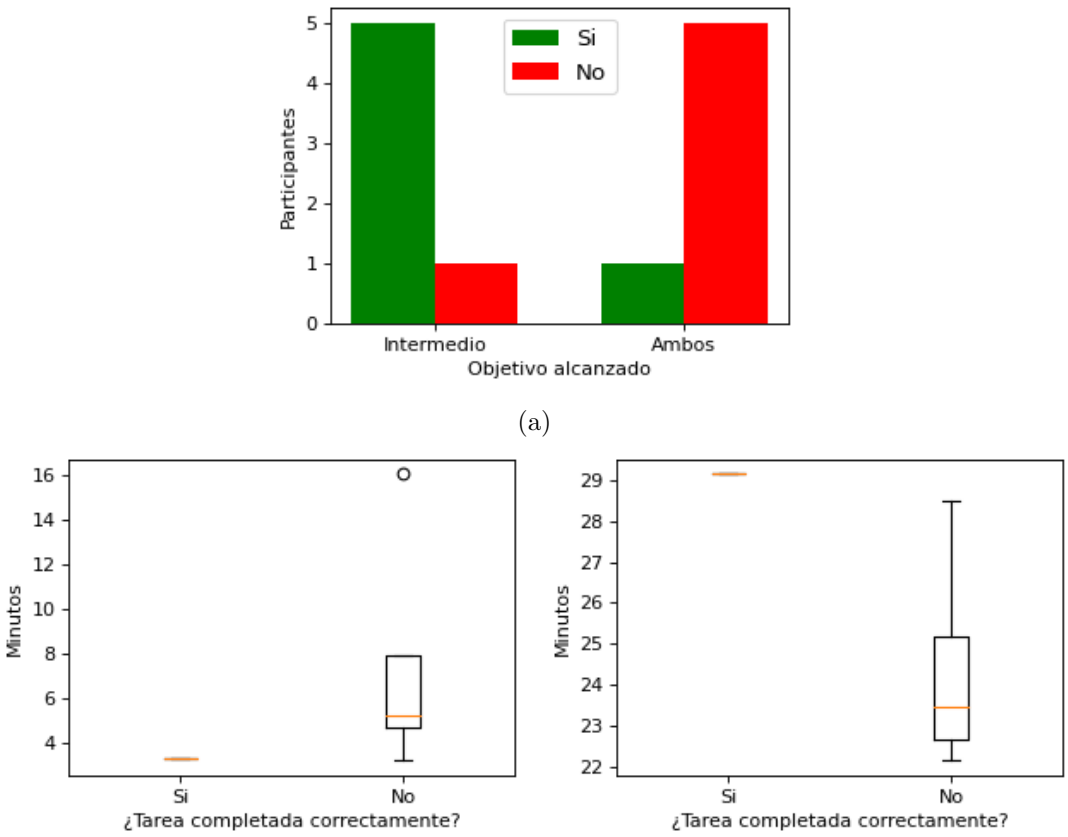
6.2. Resultados

A continuación se exponen los resultados obtenidos tras la realización del experimento a seis personas. La figura 6.1 resume los datos cuantitativos obtenidos, en los cuales se puede ver que, si bien un gran número de participantes no completo con éxito ambos objetivos, casi la totalidad de ellos alcanzó el objetivo intermedio (crear programas secuenciales) en menos de diez minutos.

Al analizar las respuestas de los cuestionarios, se han obtenido los siguientes resultados:

- A la pregunta de qué es lo que consideran más difícil de hacer, los participantes respondieron 'no teletransportarse accidentalmente' y 'encajar las piezas correctamente'.
- Cuándo se les preguntó si se habían sentido cómodos programando en realidad virtual, la mayoría de participantes respondió afirmativamente, explicando que la realidad virtual les aportó una perspectiva del programa que estaban creando muy diferente a los entornos de texto.
- Después se preguntó a los participantes si habían echado en falta algo en el entorno, a lo que la práctica totalidad respondió afirmativamente y aquello que echaron en falta fue un índice con los ejes XYZ para orientarse, disponer de instrucciones que permitan mover partes del dron y mejoras en la interacción persona-ordenador como cambiar la forma y color de fondo de los elementos del menú para crear entidades de manera que se correspondan con los del objeto que se va a crear o cambiar el botón de teletransporte por uno menos accesible.

[1] https://jdjuli.gitlab.io/virto-experiment/public/



(a)

(b)

(c)

Figura 6.1: Resultados cuantitativos del experimento de validación. La gráfica (a) muestra el número de participantes que alcanzaron satisfactoriamente el objetivo intermedio y ambos (implementando correctamente el programa). En (b) y (c) se pueden apreciar respectivamente el tiempo que destinaron a alcanzar el objetivo intermedio y el que permanecieron en total en la escena, divididos en ambos casos en función de si su solución fue correcta o no

- A la última pregunta, sobre si usarían entornos de programación virtuales en lugar de los actuales basados en texto (suponiendo que irán apareciendo más y mejores entornos del primer tipo), las respuestas fueron diversas, prefiriendo dedicar estos entornos a enseñar a programar o tareas sencillas o apostando por un modelo de programación híbrido, mientras que otros participantes manifestaron que de avanzar lo suficiente, programarían en un entorno completamente virtual.

- Una vez terminadas las preguntas se dejó un tiempo para que aportaran algún comentario si así lo deseaban, momento que algunos participantes aprovecharon para sugerir funcionalidades como poder eliminar variables (en la versión `v0.2` utilizada en el experimento, se pueden destruir las referencias, pero no las variables a las que apuntan), utilizar las tres caras

restantes de las figuras para mostrar información o priorizar la corrección de fallos en VIRTO.

En líneas generales, a pesar de los fallos de VIRTO, los participantes reaccionaron de manera positiva al uso del prototipo, esforzándose por completar la tarea propuesta y asimilando el funcionamiento del entorno en los quince minutos que dura la fase de entrenamiento, por lo que se puede concluir que VIRTO es un entorno de programación en realidad virtual adecuado para la realización de programas sencillos cuyo objetivo sea controlar un móvil.

7 Conclusiones

En este capítulo se revisará la consecución de los objetivos planteados, así como los contenidos del Grado en Ingeniería Informática que más han contribuido al desarrollo del presente trabajo. También se expondrá la planificación temporal seguida y las posibles líneas a través de las cuales el prototipo podría seguir desarrollándose.

7.1. Consecución de objetivos

A través del desarrollo de VIRTO, documentado a través de un blog[1], se han alcanzado los objetivos establecidos, ya que para llegar a su funcionamiento y aspecto actual, fue necesario explorar, mediante la creación de prototipos, cómo podría ser un entorno de desarrollo en realidad virtual. Para todo ello, se ha empleado el framework *A-Frame*, gracias al cual, las escenas han podido ser visualizadas desde el navegador web integrado en las *Meta Quest 2* y, por otro lado, la arquitectura *entidad-componente-sistema*, ha permitido crear componentes reutilizables y en muchos casos, reemplazables, lo cual, unido a la modularidad del intérprete-depurador, facilita la tarea de ampliar las funcionalidades de VIRTO.

Inicialmente se trabajó en posibilitar la creación y ejecución de programas secuenciales, para después extender el lenguaje para incluir las estructuras de control de bifurcación ('*if*') y bucle ('*while*'), logrando ofrecer un catálogo de instrucciones, variables y parámetros visible en la figura 7.1.

[1] `https://jdjuli.github.io/virto/`

De forma paralela al desarrollo del lenguaje de ViRto y del intérprete-depurador, se ha logrado implementar con éxito la serialización de programas a través de la estructura del DOM y su correspondiente des-serialización, haciendo posible escribir programas en HTML y poder ejecutarlos y modificarlos a posteriori desde la realidad virtual. Para validar el prototipo desarrollado, se diseñó un experimento

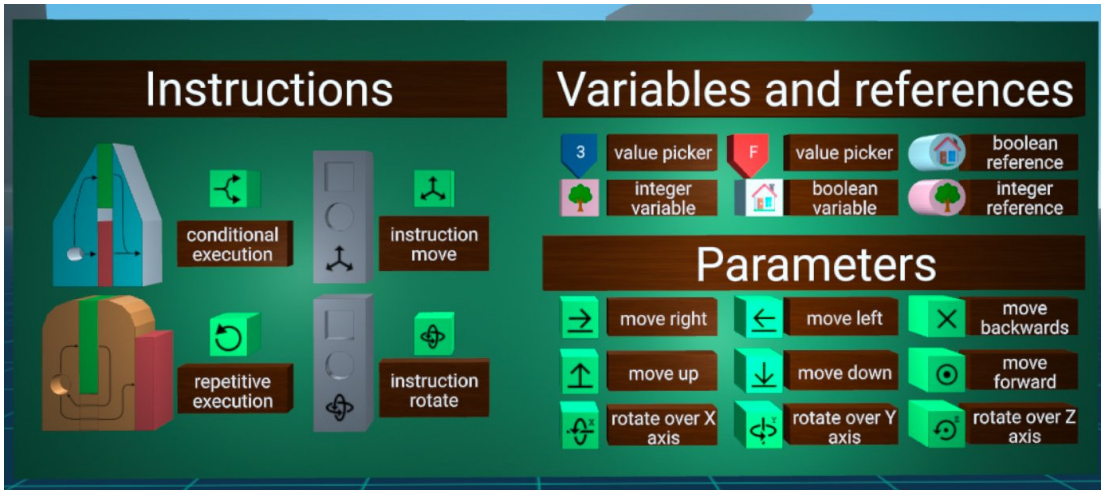


Figura 7.1: Panel de ayuda extraído del experimento que muestra todas las entidades disponibles para la creación de programas.
Fuente: Elaboración propia

con el objetivo de comprobar la idoneidad de la interfaz implementada y recabar la impresión de los participantes a través de una entrevista, cuyos resultados han sido favorables y han permitido además, conocer las expectativas de los usuarios de cara a mejorar ViRto.

La herramienta ViRto está disponible para descarga clonando el repositorio que aloja el proyecto[2] y se puede utilizar ejecutando el archivo '*HTTPS_server.bat*' en Windows. Dicho repositorio se distribuye bajo licencia MIT[3], salvo aquellas partes excluidas en la licencia, que se distribuyen bajo las licencias establecidas por sus respectivos autores.

Además de alcanzar los objetivos planteados, se han alcanzado otros que no estaban previstos inicialmente:

- Se presentó la idea en el CUSL[4] (edición 2021), bajo el nombre *VR-Programming*[5], llegando a la fase final y obteniendo una 'Mención de Honor'[6] por ello.

[2] https://github.com/jdjuli/virto
[3] https://choosealicense.com/licenses/mit/
[4] Concurso Universitario de Software Libre
[5] https://concursosoftwarelibre.org/2021/proyectos/59.html
[6] https://concursosoftwarelibre.org/2021/node/57.html

- Se ha redactado un artículo titulado '*ViRto: A web-based virtual reality environment for programming*' para la conferencia de Visualización de Software VISSOFT[7] 2022, presentando la heramienta ViRto y su enfoque sobre el desarrollo de entornos de programación en realidad virtual.

7.2. Aplicación de lo aprendido

Para el desarrollo del presente trabajo han sido especialmente útiles las siguientes asignaturas del *Grado en Ingeniería Informática*:

- Ingeniería del Software
- Programación Orientada a Objetos
- Programación Declarativa
- Ampliación de Ingeniería del Software
- Sistemas Distribuidos
- Procesadores de Lenguajes
- Informática Gráfica

Las asignaturas de programación y Sistemas Distribuidos han sido esenciales para aprender a trabajar con el framework *A-Frame* y desarrollar los componentes. Estas asignaturas unidas a Procesadores de Lenguajes, han hecho posible también el desarrollo de un lenguaje sencillo para ViRto, así como la creación de un intérprete-depurados que ejecute sobre el *drone* los programas creados con él.

Por otra parte, las asignaturas de Ingeniería del Software han permitido determinar que el modelo de desarrollo iterativo incremental es ideal para este tipo de trabajos y, para cada iteración, planificar los cambios a realizar para avanzar hacia la consecución de los objetivos.

La asignatura de Informática Gráfica ha sido de utilidad a la hora de crear los modelos 3D y sus texturas gracias a los conceptos adquiridos en ella, pese a lo cual, se ha necesitado aprender a crear modelos 3D y texturizarlos, ya que quedaba fuera del alcance de dicha asignatura. Además de modelado 3D, también ha sido necesario aprender el lenguaje *MarkDown* para poder mantener el blog[8] del trabajo (haciendo uso de *HUGO*) y LaTeXpara la elaboración de esta memoria.

[7] https://vissoft.info/
[8] https://jdjuli.github.io/virto/

7.3. Planificación temporal

El trabajo se ha desarrollado a lo largo de un año natural, comprendido entre junio de 2021 y 2022, trabajando en las iteraciones una tras otra, intensificando el esfuerzo durante la participación en el CUSL y la redacción del artículo, como se puede apreciar en el diagrama de Gantt de la figura 7.2.

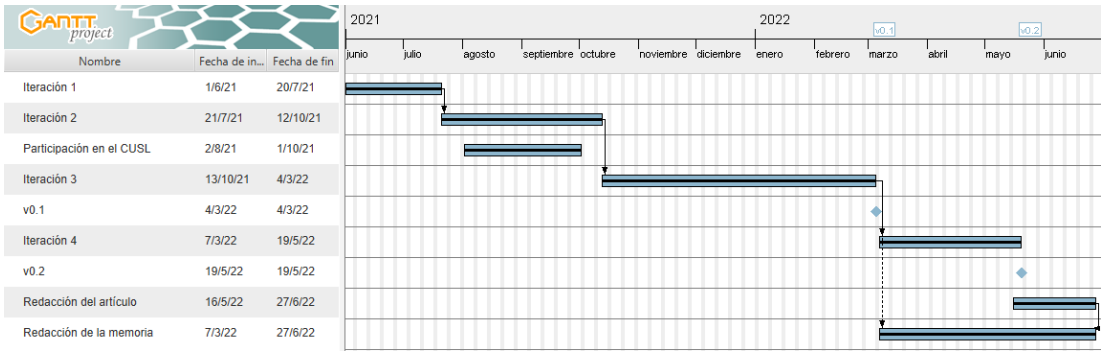


Figura 7.2: Diagrama de Gantt mostrando el tiempo invertido en cada una de las etapas del trabajo, así como las dependencias entre ellas.
Fuente: Elaboración propia

7.4. Trabajo futuro

En esta sección se proponen, a partir de la experiencia adquirida durante la realización de este trabajo, posibles cambios u objetivos futuros.

1. Prestar más atención a la interacción persona-ordenador, corrigiendo los fallos de la versión actual (v0.2), identificando en qué medida se deben a los componentes de la comunidad, y colaborando en su caso, con los respectivos autores para corregirlos o extenderlos.

2. Integrar algún protocolo *Peer-to-Peer* como *WebRTC* para que dos o más usuarios puedan trabajar juntos en el mismo entorno de programación.

3. Ampliar el lenguaje de `ViRto` incluyendo instrucciones de asignación y operaciones aritméticas y lógicas, además de posibilitar que los programas se puedan invocar entre sí.

4. Investigar la programación basada en flujo[12] como alternativa o complemento al modelo de programación basada en bloques que implementa `ViRto`.

Bibliografía

[1] ERIC S. HINTZ, "The mother of all demos," https://invention.si.edu/mother-all-demos, accessed: 2022-06-24.

[2] J. Maloney, L. Burd, Y. Kafai, N. Rusk, B. Silverman, and M. Resnick, "Scratch: a sneak preview [education]," in *Proceedings. Second International Conference on Creating, Connecting and Collaborating through Computing, 2004.*, 2004, pp. 104–109.

[3] I. E. Sutherland, "Sketch pad a man-machine graphical communication system," in *Proceedings of the SHARE Design Automation Workshop*, ser. DAC '64. New York, NY, USA: Association for Computing Machinery, 1964, p. 6.329–6.346. [Online]. Available: https://doi.org/10.1145/800265.810742

[4] T. E. Johnson, "Sketchpad iii: A computer program for drawing in three dimensions," in *Proceedings of the May 21-23, 1963, Spring Joint Computer Conference*, ser. AFIPS '63 (Spring). New York, NY, USA: Association for Computing Machinery, 1963, p. 347–353. [Online]. Available: https://doi.org/10.1145/1461551.1461592

[5] C. Solomon, B. Harvey, K. Kahn, H. Lieberman, M. L. Miller, M. Minsky, A. Papert, and B. Silverman, "History of logo," *Proc. ACM Program. Lang.*, vol. 4, no. HOPL, pp. 36–41, jun 2020. [Online]. Available: https://doi.org/10.1145/3386329

[6] M. Najork and S. Kaplan, "The cube languages," in *Proceedings 1991 IEEE Workshop on Visual Languages*, 1991, pp. 218–224.

[7] ——, "A prototype implementation of the cube language," in *Proceedings IEEE Workshop on Visual Languages*, 1992, pp. 270–272.

[8] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, "The scratch programming language and environment," *ACM Trans. Comput. Educ.*, vol. 10, no. 4, nov 2010. [Online]. Available: https://doi.org/10.1145/1868358.1868363

[9] B. Harvey and J. Mönig, "Bringing "no ceiling" to scratch: Can one language serve kids and computer scientists," *Proc. Constructionism*, pp. 1–10, 2010.

[10] R. J. Segura, F. J. del Pino, C. J. Ogáyar, and A. J. Rueda, "Vr-ocks: A virtual reality game for learning the basic concepts of programming," *Computer Applications in Engineering Education*, vol. 28, no. 1, pp. 31–41, 2020. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cae.22172

[11] Ecma International, "Ecmascript: A general purpose, cross-platform programming language," https://www.ecma-international.org/wp-content/uploads/ECMA-262_1st_edition_june_1997.pdf, accessed: 2022-05-19.

[12] S. Alexandrova, Z. Tatlock, and M. Cakmak, "Roboflow: A flow-based visual programming language for mobile manipulation tasks," in *2015 IEEE International Conference on Robotics and Automation (ICRA)*, 2015, pp. 5537–5544.

Apéndices

A

Guía del experimentador

Consideraciones generales

- El experimento consta de cuatro escenas de realidad virtual, las tres primeras están destinadas a aprender a usar la interfaz y la cuarta, requiere implementar un programa desde cero a partir de una descripción del mismo.
- Se grabará todo el experimento sin audio para descubrir puntos de mejora de cara a futuras versiones de la herramienta.
- Se requerirá al participante que avise antes de empezar a usar cualquiera de las escenas para medir el tiempo transcurrido en ellas.
- Se resolverá al usuario cualquier duda sobre el entorno, siempre que las cuestiones no estén explicadas en la escena y/o puedan resolverse explorando los elementos de la escena. Durante la parte de evaluación, solo se podrán resolver preguntas en caso de que se produzca una situación anómala (por ejemplo, mensajes de batería baja, bloqueo de la escena o imposibilidad de realizar una determinada acción)
- El participante podrá reiniciar la escena tantas veces como desee dentro del tiempo establecido para cada escena, para ello, se le indicará que debe pulsar el botón con el icono ≡, recargar la página y por último, pulsar el botón *[VR]* de la esquina inferior derecha de la pantalla para activar el modo de realidad virtual.
- En la escena de evaluación, los valores utilizados por el usuario podrán ser arbitrarios, siempre que la ejecución cumpla con la descripción proporcionada. Las unidades no son lineales y, en el caso de las rotaciones, un valor de 90 produce un giro de aproximadamente 90^{o}, pero esto no es extrapolable a otras cantidades.

Consideraciones técnicas

- Hay dos tipos de instrucciones, las simples (mover y rotar) que son de color gris y las de control, que tienen otros colores. Las instrucciones simples rechazarán parámetros que

no sean apropiados para esa instrucción en lugar de mostrar un error (por ejemplo, el dron puede moverse hacia arriba o hacia abajo, pero no en torno al eje X).

- Todos los movimientos del dron son relativos a sí mismo, esto se debe tener en cuenta principalmente después de realizar una rotación, porque por ejemplo, .ªvanzar"podría no desplazar el dron en la dirección esperada.
- Hay dos tipos de variables, numéricas (rosas y válidas en las instrucciones de mover y rotar) y booleanas o lógicas (azules y válidas en las estructuras de control)
- Si surgen problemas con interfaz (objetos que no se pueden agarrar o fallos visuales), se recomienda interactuar con otros elementos antes de intentarlo de nuevo con los problemáticos. Como último recurso, se puede salir del modo de realidad virtual pulsando el botón ≡ en el controlador izquierdo y recargar la página.

Descripción del programa

En la escena de evaluación, el participante deberá crear un programa de acuerdo a la siguiente descripción:

> El *drone* tiene primero que incrementar su altura, después de eso, rotar horizontalmente hasta que se vea su lateral y a continuación, hacer *loopings* moviéndose hacia arriba o hacia abajo (la dirección se podrá modificar en cualquier momento).

B

Cuestionario

[DATE] Fecha:

[ID] Test Nº:

[TEST_YN] ¿Programa implementado correctamente? ☐ Si / ☐ No.

[HALF_TIME] Tiempo hasta llegar al objetivo intermedio:

[FULL_TIME] Tiempo total invertido en la evaluación:

A continuación, el experimentador formulará las preguntas y tomará nota de las respuestas del participante

[AGE] ¿Cuántos años tiene? :

[GEN] ¿Cuál es su género?:

- ☐ Masculino
- ☐ Femenino
- ☐ Otro (especifique):

[JPOS] ¿Cuál de las siguientes opciones describe mejor su trabajo actual?:

- ☐ Estudiante
- ☐ Desarrollador
- ☐ Investigador
- ☐ Jefe de proyecto
- ☐ Otro (especifique):

[**ELUTVP**] ¿Cuál es su nivel de experiencia con herramientas de programación visual? (p.ej. *Scratch*, etc.)

- ☐ Ninguna
- ☐ Principiante
- ☐ Experimentado
- ☐ Avanzado
- ☐ Experto

[**PEXP**] ¿Cuántos años de experiencia tiene como programador(a)?

- ☐ Menos de 1
- ☐ Entre 1 y 3
- ☐ Entre 4 y 6
- ☐ Entre 7 y 10
- ☐ Más de 10

[**TUTVP**] ¿Cuántos años de experiencia tiene usando herramientas de programación visual?

- ☐ Menos de 1
- ☐ Entre 1 y 3
- ☐ Entre 4 y 6
- ☐ Entre 7 y 10
- ☐ Más de 10

[**EVRDEV**] ¿Cómo describiría su experiencia utilizando dispositivos de realidad virtual?

- ☐ Ninguna
- ☐ Principiante
- ☐ Experimentado
- ☐ Avanzado
- ☐ Experto

[**ELUTVP**] ¿Qué opina de la frase '*Este experimento es difícil*'?

- ☐ Muy en desacuerdo
- ☐ En desacuerdo
- ☐ Neutral
- ☐ De acuerdo
- ☐ Muy de acuerdo

[MCTTBD] ¿Qué considera que ha sido más complicado de hacer?

[CPVR] ¿Se encontraba cómodo programando en realidad virtual?¿Por qué?

[MSIITS_YN] ¿Ha echado en falta algo importante en la escena? (Si/No), ¿Qué era?

[WUVRP] Imagínese programando en realidad virtual en el futuro, ¿Preferiría utilizar programas como este (suponiendo que evolucionarán) u otros más similares a los entornos de desarrollo actuales basados en texto?

[EXTRA] ¿Quiere añadir algo más sobre la escena de realidad virtual?