UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

PERMACULTURE SIMULATOR

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

CRISTIAN IGNACIO BUSTOS ROJAS

PROFESOR GUÍA:
JÉRÉMY BARBAY

MIEMBROS DE LA COMISIÓN:
NELSON BALOIAN TATARYAN
CESAR GUERRERO SALDIVIA

SANTIAGO DE CHILE
2023

Resumen

Simulador de Permacultura

Debido al fácil acceso a plataformas digitales y su habilidad de transmitir información en cosa de segundos, muchas instituciones educacionales han comenzado a utilizarlas para hacer clases en línea. El *aprendizaje activo* es una parte esencial del proceso formativo que lamentablemente se pierde en algunos cursos porque sus contenidos no pueden ser puestos en práctica en un entorno virtual. Un ejemplo de esto es la permacultura, un campo que estudia a la naturaleza en el que dado las limitantes temporales de la naturaleza, solo pueden validar su aprendizaje mediante las observaciones de sus instructores.

¿Será posible desarrollar una herramienta digital que mejore la experiencia formativa de los estudiantes de permacultura al permitirles ilustrar conceptos y poner en práctica los conocimientos adquiridos en clases? Por ello, se propone la creación de un **Simulador de Permacultura** que permita ilustrar conceptos al trabajar en actividades interactivas.

El sueño infantil siempre será una noble ambición.

Agradecimientos

Agradecimiento eterno a mi familia, que a pesar de lo duro y los sacrificios que se hecho los últimos años, siempre me han apoyado para yo poder formarme como profesional. Siempre han cuidado de mi y gracias a ello, pude sobrellevar mis problemas de salud de salir y ser lo que hoy soy. De verdad, muchas gracias, a pesar de que muchas veces no lo de muestre, los quiero mucho.

Mis *panas* de *Anime no Seishin Doukoukai*. Gracias por darme un pequeño rinconcito para existir en la Universidad que a veces puede tornarse tan hostil. Espero que siempre estén y sigan haciendo lo mismo por mi y otros. Una mención especial a *Pipeño*, *Ivaicito*, *Gus*, *Chelo*, *Seba* y *Lulo*, que siempre han estado para mi en todo momento, alegrando mis días, tornándolos en circos con sus tallas fomes.

Tía Bárbara, muchas gracias por tratarme como un hijo más. Siempre dándome palabras de apoyo y estar para mi cuando lo he necesitado.

A mis mascotitas, Café, Mimi, Eva, Manchi, Oso, Gordis y Flaca, que constantemente me permiten apreciar lo simple de la vida. También a Copito y Pepa que ya no están con nosotros, pero que tanto hicieron. Mimi, muchas gracias por sentarte sobre el teclado para llamar la atención, gracias a eso muchas veces encontré bugs o errores de los que no fui capaz de darme cuenta antes.

Denisse, mi Nissoide. Gracias por estar siempre conmigo, gracias por cada día enseñarme a amarme a mi mismo y motivarme a cada día ser la mejor versión de mi. Si no fuese por ti, no creo que hubiese sido capaz de estar en la posición que estoy hoy en día. Contigo siento que crezco cada día y espero poder representar por siempre lo mismo para ti.

Si alguna mención ha sido pasado por alto, no significa que no estén en mi corazoncito. Gracias infinitas.

Tabla de Contenido

Índice de Tablas

Índice de Ilustraciones

Capítulo 1

Introducción

1.1. Contexto

Permacultura puede ser definido como *el diseño consciente y el mantenimiento de ecosistemas agrícolamente productivos que mantienen la diversidad, estabilidad y resistencia de los ecosistemas naturales* [14]. Los métodos y modelos postulados por esta rama académica del diseño ecológico pueden ser la clave para aprender como utilizar de forma óptima recursos naturales que son cada vez más escasos como el agua.

Los cursos de permacultura también se han unido a la tendencia de las clases en línea utilizando múltiples plataformas como lo son Google Meets, Zoom o plataformas propias de distintas instituciones educacionales. Un ejemplo de ello es Andrew Millison, un instructor de Horticultura que hace clases de permacultura en *Oregon State University* en niveles introductorios y avanzados.

Algunos de estos cursos son en línea, lo que le permite a Andrew alcanzar los hogares de cientos de estudiantes. En dichos cursos el utiliza un *Arenero de Realidad Aumentada* descrito en la **sección 2.1**, el cual le permite crear distintos tipos de escenarios o terrenos e ilustrar conceptos relacionados a la permacultura. El además tiene un canal de YouTube [2] donde sube videos relacionados a la permacultura, principalmente orientado a la difusión de ésta.

1.2. Problema

Al igual que Andrew, hay muchos instructores y profesores a lo largo del mundo que hacen uso de plataformas digitales. Si bien es cierto que dichas plataformas tienen pros, como el largo alcance que tienen y la capacidad de romper las barreras culturales y lingüísticas (lo que es muy útil para hacer clases), tienen un gran problema: Las clases en línea pierden el *aprendizaje activo* que solo puede adquirirse con clases prácticas. En el caso puntual de la permacultura hay otro problema: *La naturaleza requiere de tiempo para crecer*. Naturalmente

esto va a resultar en una brecha entre el conocimiento teórico y práctico.

1.3. Hipótesis

Dado que vivimos en una sociedad en la que la tecnología se ha vuelto una parte esencial de nuestras vidas y prácticamente todos tienen acceso a un computador o algun *smart phone*, es posible desarrollar herramientas digitales que permitan a estudiantes de permacultura poner en práctica su conocimiento en un entorno virtual.

¿Podrían aplicaciones de fácil acceso que permiten a sus usuarios experimentar y observar el impacto de distintos modelos y patrones de diseño vistos en clases, facilitar a profesores disminuir la brecha entre el conocimiento práctico y teórico?

Herramientas con dichas características podrían cambiar radicalmente el diseño de los cursos en línea de permacultura. Por ejemplo, en un entorno virtual, el tiempo requerido para ver como la naturaleza se desarrolla dejaría de ser un problema gracias a que este permite hacer simulaciones en tiempo acelerado.

1.4. Objetivo

Por ello proponemos la creación de un simulador interactivo que permita replicar tanto como sea posible lo que puede ser hecho con el *AR Sandbox*. Por ende, las principales características de esta aplicación deben ser:

1. Permitir a sus usuarios crear terrenos y moldearlos.
2. Mostrar curvas de nivel sobre el terreno creado.
3. Poner a disposición del usuario múltiples objetos o "actores" tales como árboles o plantas.
4. Visualizar como el agua interactúa con el terreno dentro de lo que el las limitantes técnicas lo permitan. Esto pues simular fluidos es complejo a nivel computacional y requiere de un computador de alta gama al que no muchos tendrían acceso.

Aprovechando los beneficios propios de que la aplicación es ejecutada en un entorno virtual, también se planea que:

1. Simulaciones de tiempo acelerado sobre los actores.
2. Permitir a instructores crear escenarios de prueba o ejercicios para sus estudiantes.
3. Compatibilidad con la mayor cantidad de plataformas posibles. Por ello, una **aplicación web** podría ser la mejor opción ya que no requiere instalación de programas y su facilidad para portear el resultado a otras plataformas, como smart phones o tabletas.

Para poder lograr la meta final, se fijaron los siguientes objetivos:

1. Definir el publico objetivo del simulador junto a las características fundamentales que debe tener.
2. Determinar cual es el nivel de simulación que queremos lograr, tratando de tener un balance entre fidelidad y rendimiento.
3. Establecer el lenguaje/framework más apropiado para la tarea.
4. Diseñar la arquitectura de la aplicación, priorizando un diseño modular para facilitar la implementación de nuevas características.
5. Usar la información adquirida durante los ciclos de desarrollos para determinar cuales son las siguientes características a implementar.

1.5. Validación

Las principales aristas a validar de un proyecto como este son:

- Usabilidad: Cuan fácil es de usar o entender como funciona.
- Accesibilidad: Que el resultado final sea utilizable por la mayor cantidad de usuarios posibles.

Para ello, se hicieron ciclos de *testeo* con usuarios que daban su opinión sobre las características que la aplicación tenía en ese momento, además de notificar como el programa rendía en sus computadores. Esta información fue utilizada para decidir nuevas características podían ser implementadas y como hacerlo con tal de preservar la estabilidad y rendimiento de la aplicación.

No todos los usuarios de los ciclos de *testeo* tenían experiencia sobre permacultura, ya que el principal objetivo de sus validaciones era adquirir información sobre que tan complejo resultaba usar al aplicación y como rendía en sus equipos. Por ello, a lo largo del proyecto se llevaron a cabo múltiples reuniones con Andrew Millison, quien nos oriento para focalizar las prioridades del proyecto.

Todo la información adquirida durante la validación del proyecto es explicada en detalle en el **capítulo 5**

Capítulo 2

Estado del Arte

Si bien es cierto que la permacultura es una rama que propone una serie de innovaciones y modelos que pueden ser claves para enfrentar una crisis medioambiental como la que vivimos hoy en día, no es un área que reciba mucha atención del ojo público. Que muchos ni siquiera sepan que existe algo como la permacultura en primer lugar es la razón por la cual no hay muchos recursos dedicados específicamente a esta área. Es por ello que se suelen usar herramientas diseñadas para otros campos que podrían tener utilidad en la enseñanza o estudio de la permacultura.

Considerando que el propósito de este proyecto es crear una aplicación que permita ilustrar conceptos de permacultura en un formato similar al *AR Sandbox* y hacer simulaciones de bajo nivel, se analizaron distintos recursos que potencialmente podrían cumplir ese rol. Primero, analizemos la estructura de los cursos de permacultura:

2.1. Cursos de Permacultura

Existen distintos tipos de cursos que cubren distintas necesidades. Estos van desde cursos introductorios que se dedican a la parte teórica, y otros avanzados, orientados principalmente al diseño de modelos de permacultura. Tal como se mencionó previamente, existen cursos en linea y otros "en terreno", siendo muchas veces notablemente distintos dado las limitantes de las plataformas digitales.

2.1.1. Cursos en Terreno

Este tipo de cursos son usualmente orientados a niveles más avanzados. Requieren que sus participantes viajen a un lugar especifico que cuente con todas las herramientas necesarias para que los estudiantes puedan poner en práctica sus conocimientos. Normalmente cuentan con distintos tipos de "biomas" como por ejemplo, tropical húmedos o tierra seca.

En ellos se realizan distintos tipos de actividades, como trabajar sobre la tierra para poner

en práctica distintos tipos de modelos. Además, siembran distintos tipos de plantas y discutir como estas podrían responder ante las modificaciones hechas sobre el terreno.

Lamentablemente, la barrera de ingreso a este tipo de cursos es muy alta. Solo se imparten en algunos países dado los requisitos que estos tienen, por lo que si alguien interesado no vive en uno de ellos, es sumamente difícil participar si no cuenta con los recursos monetarios, eso sin siquiera mencionar la barrera lingüística.

Algunos ejemplos de cursos "en terreno", son los impartidos por *Northern School of Permaculture* [12]:

- "Ecological Garden Design"
- "Urban Gardening and practical skills"
- "Design Course at Todmorden, Bury and Broadbottom (Northern Hyde)"

En esos cursos se ven temas como: Jardines auto-sustentables, el arte del compostaje, diseñar y plantar una espiral de plantas y hiervas, jardinería vertical comestible, entre otros.

2.1.2. Cursos en Línea

La distancia deja de ser una restricción y la gran mayoría de estos cursos consisten en ver vídeos, asistir a clases mediante videollamadas por plataformas como Zoom o Google Meets. Otros se basan en la lectura de libros que suelen ser proporcionados por quienes imparten el curso y realizar ejercicios que buscan evaluar el conocimiento adquirido mediante las lecturas o las clases. Algunos cursos ofrecen actividades prácticas pero estos asumen que quienes es inscribieron a ellos tienen un terreno en primer lugar.

En resumen, en la gran mayoría de estos cursos se pierde una parte esencial de proceso de aprendizaje ya que no todos cuentan con un terreno propio.

Un ejemplo de cursos online es este que ofrece PermacultureUK [15], que ofrece: descuentos en libros, y acceso a PDFs o videos complementarios. En él, las unidades son los principios de la permacultura, como aplicarla, y como diseñar modelos, finalizando el curso presentando un modelo de terreno propio.

2.2. Simuladores

Tal y como se mencionó previamente, actualmente no hay programas que hayan sido desarrollados específicamente para el área de la permacultura, mucho menos para realizar simulaciones. Es posible usar herramientas desarrolladas para otras ramas del estudio de la naturaleza, como lo son la horticultura o la agricultura. Algunos ejemplos de este tipo de aplicaciones son: QGIS [16] y RiverFlow2D [9], utilizadas principalmente para analizar o visualizar información de terrenos.

Aunque no diseñados con fines académicos, también existen videojuegos que intentan ser simuladores. Algunos ponen al jugador a cargo de granjas, terrenos, ciudades o civilizaciones enteras (Algunos ejemplos son descritos en detalle en la **sección 2.2.3**). Muchos videojuegos han sido motivo de estudios por su capacidad de ilustrar imágenes aproximadas de la realidad gracias a la naturaleza interactiva del medio. Por ejemplo, en el artículo *The untapped potential of virtual game worlds to shed light on real world epidemics* [8], se analiza en detalle como un evento en el videojuego *World of Warcraft* sirvió como un caso de estudio sobre el comportamiento humano durante eventos similares a una pandemia. Gran parte de los conocimientos adquiridos durante ese evento fueron aplicados para el manejo de la presente pandemia, lo cual demuestra el gran impacto que puede tener un entorno virtual interactivo para poder experimentar con cosas que en la realidad simplemente no se podría.

2.2.1. Simuladores de Alto Nivel

1. **QGIS / Quantum GIS**: En un software de código libre que opera como un sistema de información geofráfico ó GIS (Geographic Information System). Permite al usuario analizar y editar información de un terreno y se caracteriza por su gran variedad de extensiones que agregan nuevas características al programa. Algunos ejemplos de estas extensiones son:

    - CSWI: Calcula el índice de estrés hídrico de los cultivos utilizando imágenes térmicas del suelo.
    - Precision Agriculture Tools: Proporciona al usuario toda la información necesaria para garantizar que los cultivos y el suelo reciban exactamente lo que necesitan para una óptima salud y productividad.

2. **RiverFlow2D**: Es un programa de modelación de fluidos en 2D (Top view). Se caracteriza por su complejidad y fidelidad al hacer simulaciones. Requiere la creación de un modelo terreno mediante el uso de herramientas externas, en algunos casos el uso de drones para escanear una localización sobre la cual se quieren hacer simulaciones. En el contexto de la permacultura puede ser utilizado como para hacer experimentos sobre el manejo de recursos hídricos en distintos tipos de terrenos.

2.2.2. Simuladores con Fines Pedagógicos

1. **Augmented Reality Sandbox (AR Sandbox)**: Usando un computador, un proyector, un sensor *Kinect* y un arenero, se creó lo que puede ser descrito como la forma ideal de ilustrar conceptos de permacultura de forma interactiva. Usando el sensor *Kinect*, el computador escanea el estado del arenero, la cual es procesada para determinar sus curvas de nivel y con ello, generar una imagen y proyectarla sobre dicho arenero.

    Además, detecta gestos hechos con las manos, los cuales pueden ser interpretados por el sensor para ejecutar distintos comandos, como proyectar agua en una zona del arenero al poner la palma abierta sobre ella.

Lamentablemente, es difícil acceder a un AR Sandbox dado su alto precio, por lo que está destinado principalmente a profesores o instructores para usar durante sus clases para ilustrar conceptos en tiempo real.

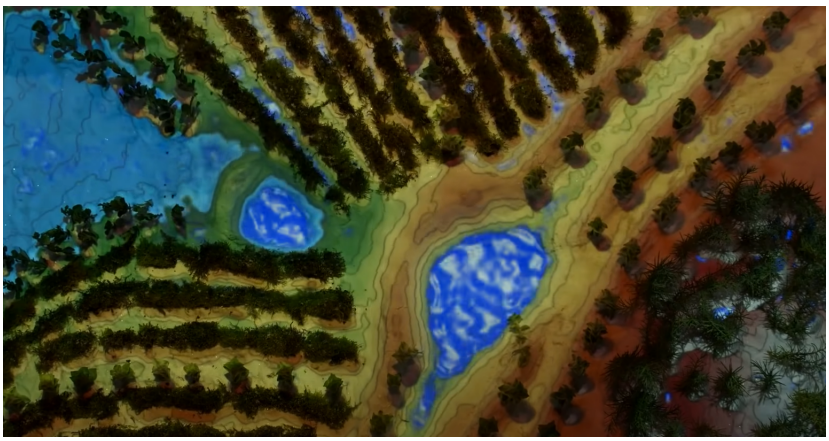

Figura 2.1: Millison, Andrew: *This Farm Design Can HEAL the PLANET*. YouTube [1]

2.2.3. Simuladores "Game-Like"

1. **Populous**: Videojuego publicado por *Electronic Arts* y desarrollado por *Bullfrog Productions*. Es el precursor del genero "God Game", en el que el jugador suele ser una entidad omnipotente que cuenta con múltiples herramientas para modificar el mundo y los recursos disponibles para una civilización humana. Aplicando distintas modificaciones a su entorno, el jugador es capaz de ver como las civilizaciones responden ante ellos y evolucionan.

Figura 2.2: *Populous (Video Game)* [6]

2. **Sim City**: Videojuego que pone al jugador en el papel de un alcalde. En dicho rol, el jugador parte con un lienzo en blanco sobre el cual debe construir una ciudad y proveer a todos sus futuros habitantes con los recursos esenciales para la vida.

3. **Farm Manager**: Videojuego lanzado en 2021 que pone al jugador en el rol de un granjero que debe mantener su tierra y observarla crecer. Las características del juego van desde la compra y venta de productos para poder mantener la tierra. La creatividad del jugador para utilizar de forma óptima los recursos con los que cuentan son claves para mantener su granja.

2.3. Renderizado y Desarrollo de Aplicaciones Web

En la **sección 1.4** se propuso el desarrollo de una *aplicación web*. Aunque el desarrollo de un simulador o ilustrador que sea capaz de ejecutarse en un navegador pueda sonar un poco descabellado, durante los 10 últimos años las herramientas de desarrollo para este ambiente han evolucionado a pasos agigantados.

WebGL [22] es una API desarrollada en JavaScript que permite a desarrolladores web utilizar la GPU del computador del usuario para generar imagenes 2D o 3D en un formato similar a **OpenGL** [13]. Si bien es cierto que no cuenta con todas las herramientas, como podrían ser la ausencia de *Geometry Shaders*, WebGL ha demostrado mucho potencial al ser capaz de usar técnicas sofisticadas como *Ray Tracing* [11].

Existen Frameworks que hacen uso de WebGL que facilitan considerablemente el desarrollo de aplicaciones o videojuegos que corren en navegadores web, como lo son THREE.js[20] y Babylon.js [5].

A fecha de este informe, se está estandarizando el uso de WebGPU [23]: una API similar a WebGL pero que es aún más poderosa por el tipo de herramientas que entrega y el rendimiento de las aplicaciones ya que su arquitectura está basada en **Vulkan** [21].

Capítulo 3

Diseño

En este capítulo se explicará en detalle la arquitectura de Permaculture Simulator junto a observaciones sobre el trabajo realizado. El presente capitulo se divide en las siguientes secciones:

- **Conceptos Claves**: Definición de la terminología aplicada durante este capitulo y el capitulo de Implementación.
- **Sistemas de la aplicación**: Descripción de los sistemas que posee el programa.
- **Arquitectura del motor del simulador**: Explicar en detalle cuales son los componentes de la aplicación y como interactúan entre ellos.
- **Flujo de Usuario**: Se expone cual es la experiencia planeada para los distintos tipos de usuarios finales.

3.1. Conceptos Clave

1. **Renderer**: API o programa a cargo de renderizar objetos 2D ó 3D. Se le debe entregar como parámetros una *escena* y una *cámara*. Para poder generar múltiples imágenes, el **Renderer** debe ser invocado de forma indefinida en un loop al que llamaremos como **Loop de Animación** ó **Función de Renderizado**.
2. **Escena**: Estructura de datos que almacena todos los objetos que serán renderizados en pantalla. Se debe entregar como parámetro al **Renderer**.
3. **Cámara**: Como lo indica su nombre, es una serie de programas que permiten mostrarle al usuario el entorno virtual o escena al usuario. Hay distintos tipos de cámaras, como las *PerspectiveCameras* que se caracterizan por imitar como el ojo humano ve.
4. **Geometry**: Estructura de datos que contiene la información necesaria para construir una malla, línea o punto geométrico. Incluye posiciones de vértices, índices de las *caras* que forman una malla, vectores normales, colores por vértices, etc.

Esta estructura suele venir acompañada de una serie de funciones que permiten modificar los parámetros previamente mencionados.

5. **Material**: Información que declara la apariencia de los objetos renderizados.

6. **Actor**: También llamados objetos, son el resultado final de mezclar una geometría con un material. Dentro de esta categoría entran modelos 3D, *sprites*, puntos, lineas, mallas geométricas, entre otros.

   Tienen modificadores propios que alteran el como este objeto interactúa con su entorno o como es renderizado.

7. **Motor de Videojuego**: Frecuentemente referido por su nombre en inglés, *game engine*, se define como conjunto de herramientas que permiten el diseño, creación y funcionamiento de un videojuego.

   Sus características principales son: Motor gráfico, motor de físicas, sistema de creación de scripts (programas que se ejecutan en tiempo real dentro del juego), entre otros.

3.2. Sistemas y Características deseables en la Aplicación

Permaculture Simulator, en términos simples, es un videojuego que busca crear un entorno que facilite la ilustración de conceptos complejos del área de la permacultura y ayude el proceso proceso formativo en dicha rama académica. Al ser una aplicación pensada para profesores y estudiantes por igual, debe contar con herramientas y características que incentiven el uso por parte de esos perfiles de usuario.

Es por ello, que se establecieron los siguientes sistemas y principios:

3.2.1. Mecánica principal

Permaculture Simulator recibe al usuario con un modelo 3D o malla poligonal que represente un terreno. El programa pone a disposición del usuario una serie de herramientas que permitan modificar dicho terreno. Además, el terreno cuenta con opciones extras que facilitan su manipulación o análisis, como modificar su tamaño o visualizar sus curvas de nivel, similar a como lo hace el *AR Sandbox*.

Sobre el terreno, el usuario puede además poner distintos *actores* como si fuese una maqueta. Dentro de esta categoría de actores entrarían: árboles, plantas, edificios, casas, entre otros.

El agua y su uso es uno de los pilares fundamentales de la permacultura. Por lo que al usuario se le otorgarán distintas herramientas para poder agregar al agua a la *escena* y ver como el resto de actores responden ante ella. Es importante notar que la simulación de fluidos tiene un costo computacional muy alto, por lo que la prioridad es buscar formas de **ilustrar** y visualizar el agua, **no simularla**

Finalmente, la aplicación cuenta con una serie de herramientas que le permitan a instructores **crear y compartir** ejercicios con sus estudiantes. De la misma forma, se necesita la opción de poder compartir las escenas creadas dentro de la aplicación.

3.2.2. Modos de Uso

La aplicación tiene tres modos de uso:

- Un modo totalmente libre en donde sus usuarios pueden experimentar con las herramientas que la aplicación ofrece. Al inicio de este modo, el usuario puede seleccionar el tipo de escena sobre la cual desea trabajar, donde la diferencia radica en con que tipo de terreno desean partir:
  1. Terreno plano: Es el tipo de escena básico. Ideal para crear ejercicios sobre él.
  2. Terreno generado por imagen: El usuario puede cargar una imagen a la aplicación, la cual es utilizada como un **heightmap**[10].
  3. Terreno generado de forma aleatoria: Utilizando **Perlin Noise**[17], se generan valores que serán utilizados como un **heightmap**
- Un modo en donde se pueden crear distintos escenarios con una serie de objetivos diseñados para poner a prueba conocimientos de permacultura.
- Un modo en donde ejecutar los ejercicios que alguien más desarrolló.

Naturalmente, esto supone que la aplicación y sus componentes deberían poder tener distintos comportamientos en función del contexto en el cual están siendo ejecutados.

3.2.3. Interacción y Herramientas

Tal y como se mencionó en la **sección 3.2.1**, Permaculture Simulator cuenta con una serie de herramientas que le permitan al usuario interactuar con una escena. Estas herramientas son accesibles para el usuario en cualquier momento desde la interfaz del programa. Cada herramienta cuenta con los siguientes atributos:

Nombre del Paramatro	Tipo	Rol
ID	String	Identificar a la herramienta del resto. Su valor debe ser único
Type	String	Deben existir distintos tipos de herramientas que permitan al usuario interactuar con todos los tipos de actores en escena, como el terreno, arboles, plantas, entre otros.
Icon	Img	Todas las herramientas tendrán un botón asociado en la interfaz de usuario para que éste pueda seleccionarlas y utilizarlas. Es por ello que cada herramienta debe tener un ícono que la represente
Callbacks	Functions (Ref)	Funciones que son llamadas cuando se gatillan distintos tipos de eventos como: Seleccionar o deseleccionar una herramienta, mover el cursor o presionar un botón.
Key Map	Array / Dictionary	Indica cuales son los botones que la activan los distintos callbacks de la herramienta. Esta información puede ser utilizada para detectar cuando una herramienta es usada o mostrar mensajes explicando como utilizarla.

5.0pt

Tabla 3.1: Diseño: Estructura de las herramientas

Por conveniencia, se estableció que cierta información de la herramienta que está seleccionada debe ser visible de forma global, como su ID y su tipo.

3.2.4. Ejercicios

Previamente se mencionó que Permaculture Simulator cuenta con un conjunto de herramientas que le permitan a profesores e instructores crear ejercicios. Se estableció que las características y reglas fundamentales de los ejercicios deben ser las siguientes:

1. Los ejercicios son creados desde la misma aplicación sin el uso de herramientas externas, por lo que debe existir un menú dentro del programa que guíe al usuario en la creación del ejercicio. Cuando el profesor o instructor termine de crear el ejercicio, se generará un archivo que puede ser compartido con sus estudiantes. Dicho archivo al ser cargado en la aplicación, pondrá al estudiante en un ambiente controlado donde completar una serie de **objetivos**.
2. Los ejercicios pueden tener múltiples objetivos, cada uno con una "**condición de victoria**". Al completar uno, pasas al siguiente, y cuando ya no queden objetivos pendientes, se asume que el ejercicio fue completado con éxito.
3. Se asume que los profesores que creen ejercicios no saben programar, por lo que las **condiciones de victoria** de los objetivos están predefinidas. De esta forma, el instructor o profesor modifica parámetros que le permitirán personalizar la experiencia. Algunos ejemplos de estos objetivos y condiciones de victorias "predefinidas" son:

   - **Replicar configuración de terreno**: En este escenario, el profesor crea un terreno y se lo entrega al creador de ejercicios como parámetro. La condición de victoria es que el alumno debe crear un terreno similar al que el profesor le entrega como ejercicio.

     Además, el profesor puede adjuntar mensajes o imágenes que expliquen que es lo que hace a esa configuración de terreno especial.

- **Uso de árboles y plantas**: El profesor marca áreas del terreno que son entregadas al creador de ejercicios como parámetro. La condición de victoria en este caso sería que los alumnos deben insertar una cantidad arbitraria de árboles o plantas en el área marcada por el profesor.

  El objetivo iría acompañado de una explicación sobre porqué posicionar arboles o plantas en esas zonas es una buena o mala idea.
- **Uso de Recursos Naturales**: El profesor fija unas condiciones climáticas y recursos naturales que son entregadas al creador de ejercicios. La condición de victoria sería que el alumno debe mantener un ambiente con dichos recursos por una cantidad de tiempo arbitraria.

  El objetivo pondría aprueba los conocimientos sobre como trabajar en conjunto con la naturaleza que posee el estudiante.

4. Es posible añadir un nombre y descripción a cada objetivo del ejercicio que se vean en pantalla cuando dicho objetivo es inicializado. Esto con la intención de poder guiar al estudiante que esté trabajando en él. También, se pueden agregar mensajes que aparezcan durante el desarrollo de un objetivo que le expliquen al estudiante que está haciendo y porqué lo está haciendo.

3.2.5. Importar y Exportar

Como se mencionó previamente, la aplicación le permite al usuario poder compartir o cargar escenas y ejercicios creados por terceros, ya sean profesores u otros alumnos. En la sección anterior se explicó como funcionaría esto con los ejercicios: Tras crear uno, la aplicación genera un archivo que puede ser enviado a estudiantes.

En cuanto a compartir escenas o terrenos, el principio es similar: "Rescatar la información en un momento dado de la ejecución de todos los actores que estén en la escena, de forma que podamos recrearlos en el futuro".

Bajo la premisa de que la escena está conformada por actores, podemos programar a estos últimos de forma que cada uno tenga una función que retorne la información esencial que permita recrear su estado. En términos simples, sacar una "foto" de cada actor, preservando así la información del estado que tenía en el momento que la escena fue guardada.

Toda esta información será guardada en un archivo que el usuario puede cargar en la aplicación cuando él quiera, y al hacerlo, recrear la escena con todos sus actores.

3.2.6. Deshacer y Rehacer Cambios: *State Manager*

A diferencia de AR Sandbox, donde el usuario puede utilizar directamente sus manos para manipular la "escena", los medios virtuales no son muy precisos. Es altamente probable que alguien cometa errores al trabajar sobre ella, por lo que se propone la implementación de un "**buffer de estados**". Su rol sería guardar la información esencial que permita recrear el estado de la escena y sus actores en un momento dado de la ejecución, característica

escencial para una plataforma que trata de recrear un entorno seguro donde sus usuarios puedan aprender a base de prueba y error.

La aplicación tendrá implementado distintos mecanismos para detectar cuando debe generar un nuevo estado. Algunos ejemplos podrían ser, hacer modificaciones al terreno por más de una cantidad "x" de tiempo, agregar o remover un nuevo actor a la escena, entre otros. Cuando se active alguna de estas condiciones, se generará una copia de la escena y será guardado en este **buffer de estados**.

El usuario tendrá acceso a un par de botones que le permita moverse hacia adelante y atrás en este **buffer**, dando la ilusión de que está deshaciendo o rehaciendo los cambios hechos sobre la escena.

3.2.7. Task Buffer

Inspirado en los motores de videojuegos, se decidió implementar un **Task Buffer** o *Contenedor de tareas*. Estas *Tasks* son funciones que se caracterizan por tener un parámetro que dictamina la prioridad con la que se deben ejecutar, y porque al cumplir con su objetivo, son borradas del *buffer*.

La decisión de implementar este sistema de tareas es para poder manejar la validación de los objetivos de ejercicios, pero también porque puede tener potenciales utilidades a futuro.

3.2.8. Terreno

Con tal de mantener simplicidad, el terreno es representado por un plano. Sobre él, se podrán usar múltiples herramientas integradas en la aplicación para modificar las coordenadas de los vértices que dan forma a dicho plano.

Por ejemplo, el usuario selecciona una herramienta para crear cerros. Luego, mediante el cursor, clickea sobre una parte del terreno, de forma que todos los vértices que estén en dicha área, verán su altura modificada siguiendo un patrón similar al de un cerro. Esto se puede hizo con funciones matemáticas como la función de curva de la *Distribución de Gauss*

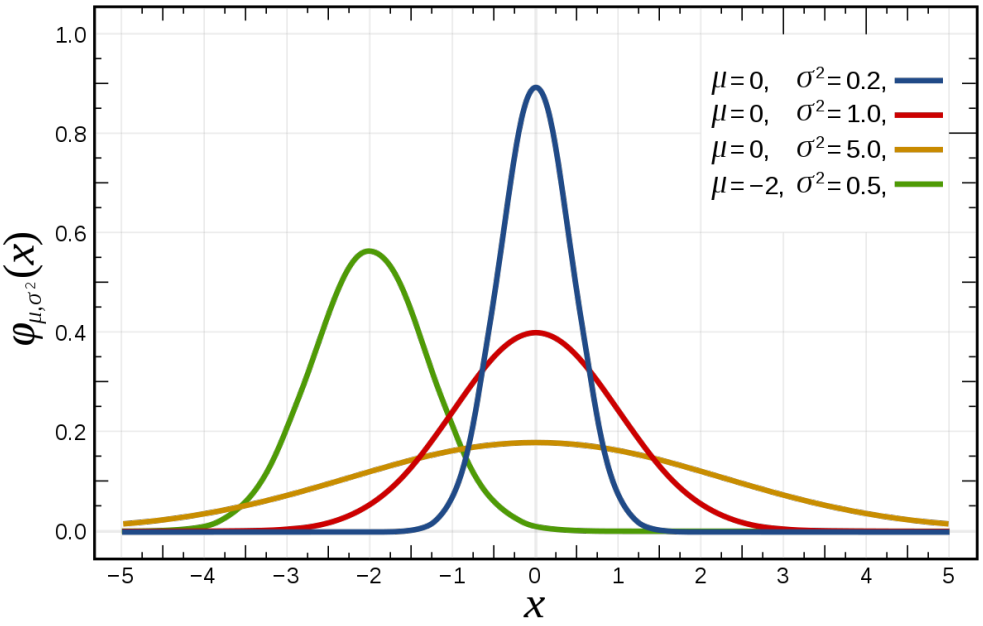

Figura 3.1: Visualización de las funciones de curva de la distribución normal/de Gauss

3.2.9. Simulación de Agua

La simulación de fluidos es un tópico altamente estudiado en computación gráfica que se caracteriza por su complejidad al buscar mayor fidelidad. Tal y como se mencionó antes, la idea tras Permaculture Simulator es ilustrar conceptos, no hacer simulaciones de alto nivel.

Por ello, se propuso utilizar "**planos**" para poder representar el agua. La idea es crear un plano de tamaño equivalente al terreno pero con usando texturas y materiales apropiados para dar la sensación de que es agua.

Interacción Usuario-Agua

Al clickear una zona del terreno en donde se desea agregar agua, se selecciona una colección de vértices

En cuanto a como el usuario interactuaría con dicho plano, lo que haría sería clickear en las zonas del terreno en donde se desee añadir agua. Luego, se tomarán vértices específicos del plano del agua y se les modificará la altura para quedar sobre el plano del terreno.

Es importante señalar que hay casos en los que el plano de agua puede atravesar otros objetos, como el plano de terreno. Para ello, se utiliza **Z Buffering**, un sistema empleado por el sistema de renderizado para poder simular profundidad en la escena. Con esto, podemos forzar a que el plano de agua tenga tal prioridad en el Z Buffering que siempre quede detrás de otros objetos, ocultando el plano de agua.

Figura 3.2: Ilustración que representa la modificación del plano de agua

Un ejemplo en el que se aplica este sistema de "simulación de agua" es en The Legend Of Zelda: Ocarina of Time, videojuego desarrollado y publicado en 1998 por Nintendo.

En la **figura 3.3** podemos ver un escenario del juego en el que hay un lago. Mediante el uso de herramientas externas, podemos ocultar el resto de actores de la escena y darnos cuenta de que el lago no es más que un plano con texturas de agua que se extiende hasta el horizonte.

Figura 3.3: The Legend of Zelda: Ocarina Of Time, 1998. Aplicación de planos de agua.

Con este tipo de técnicas es posible implementar agua como actor a la aplicación sin obligar a los usuarios a tener un equipo de alta gama para poder ejecutarla.

3.2.10. Plantas y Árboles

Son implementados como *sprites*, es decir, imágenes o texturas que siempre miran directo a la cámara. Esto nos permite dar la ilusión de que el objeto en cuestión es 3D si es integrado apropiadamente

Este tipo de técnicas se usaban mucho en videojuegos de antaño, ya que los equipos y su capacidad de renderizado de escenas 3D en tiempo real era limitada. Un ejemplo de esto un videojuego de plataformas en 3D desarrollado y publicado por Nintendo en 1997, Super Mario 64. En la **figura 3.4**, podemos ver una escena del juego que está decorada con multiples árboles que parecen ser 3D, pero si manipulamos la cámara con herramientas

externas, podemos apreciar como realmente el árbol no es más que una imagen que siempre mira de frente a la cámara.

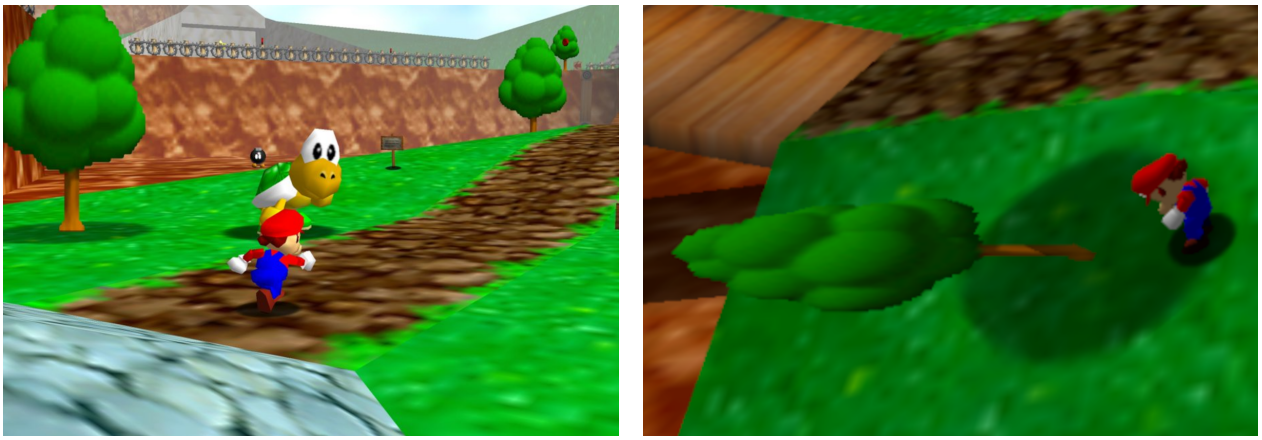

Figura 3.4: Super Mario 64, 1997. Nintendo. Uso de sprites para decoración de una escena 3D

3.3. Interfaz de usuario

Con tal de facilitar el uso y comprensión del programa, el diseño de la interfaz de usuario apunta a ser similar a programas que orientados a la edición de algo, algo equivalente a **Adobe Photoshop** o **GIMP** para la edición de imágenes. Estos programas se caracterizan por los siguientes componentes:

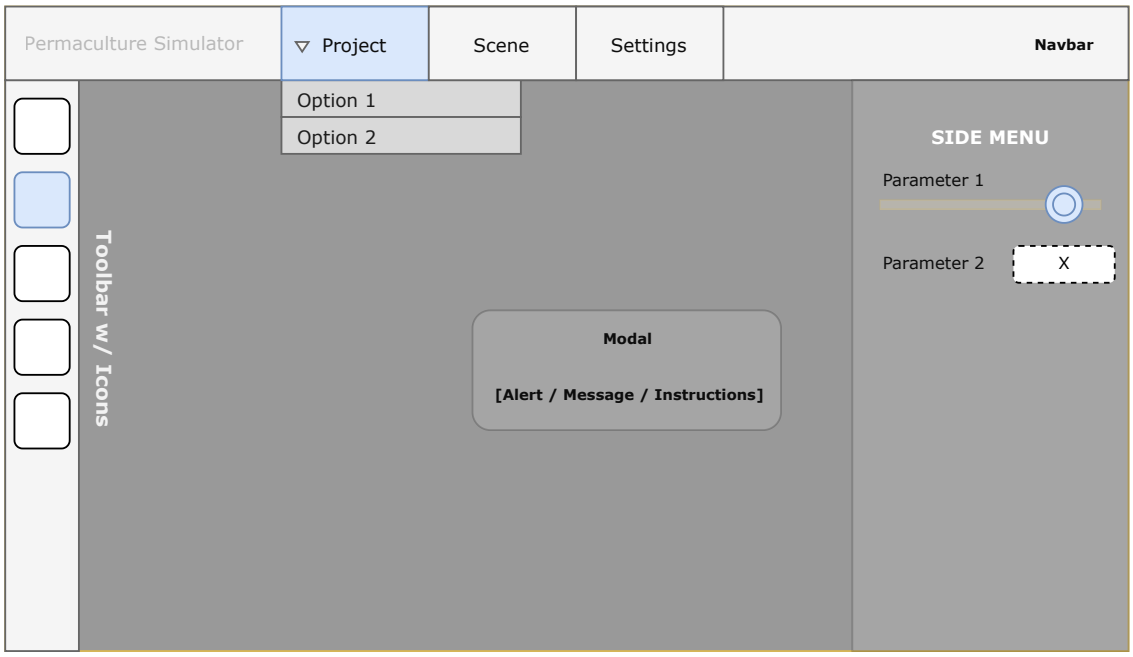

Figura 3.5: Diagrama conceptual: Interfaz de Usuario

- **Barra de Herramientas**: Suele encontrarse en lado derecho o izquierdo de la pantalla para garantizar un rápido acceso a ellas. Cada herramienta tiene un ícono asociado que trata de representar que es lo que hace.
- **Barra de navegación**: Se encuentra en el borde superior de la aplicación. Aquí deberían estar las opciones de importar/exportar contenido y botones que permitan editar parámetros globales de la escena.
- **Menús Laterales/Ventanas modales**: Con tal de no tener demasiados elementos en pantalla que puedan confundir al usuario, se planea tener menús que contienen información que no necesita estar siempre en pantalla.

3.4. Flujo de Usuario

En las siguientes figuras se presentarán diagramas que representan el flujo de usuario, diseñados en función de los *modos de uso* definidos en la **sección 3.2.2**:

3.4.1. Modo Libre

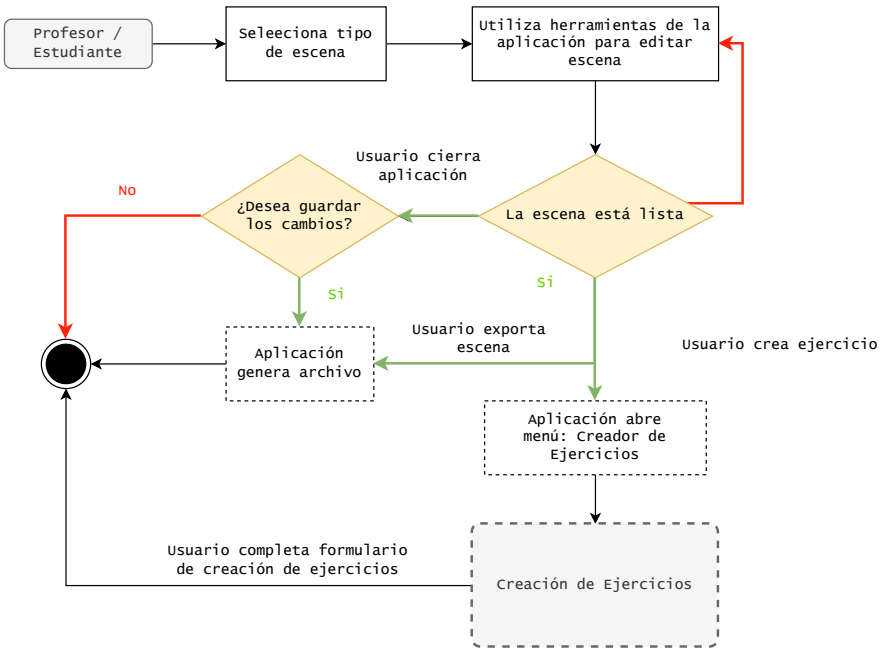

Figura 3.6: Flujo de Usuario: Representación de las interacciones de un usuario con la aplicación.

3.4.2. Creación de Ejercicios

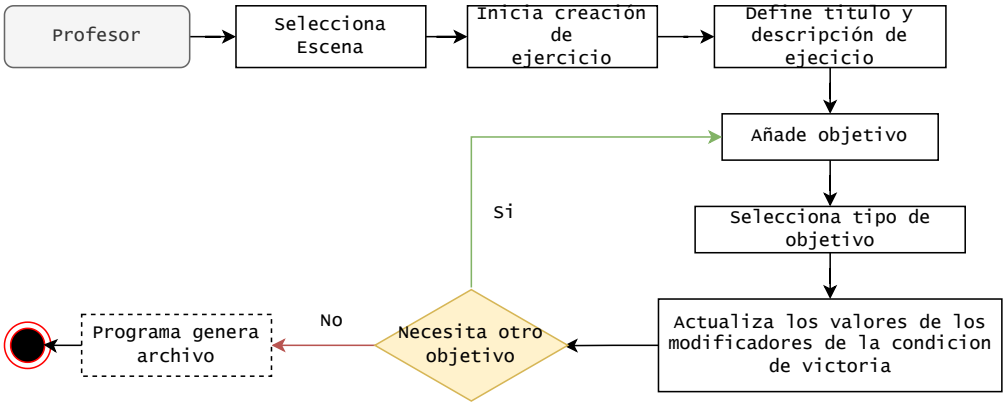

Figura 3.7: Flujo de Usuario: Representación de la creación de ejercicios.

3.4.3. Ejecución de ejercicios

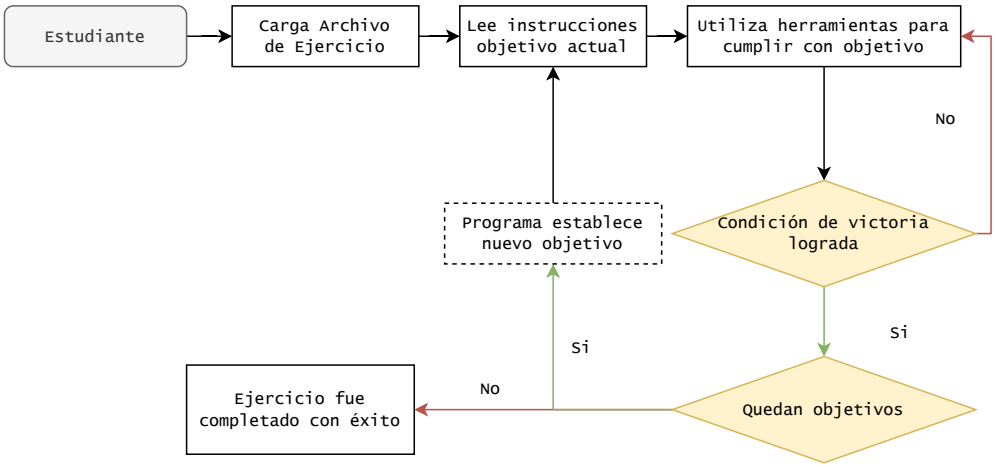

Figura 3.8: Flujo de Usuario: Como se espera que los usuarios deben interactuar con los ejercicios.

3.5. Arquitectura

La **figura A.2** ilustra una simplificación de la arquitectura de Permaculture Simulator. Esta conecta todos los conceptos básicos vistos en las **secciones 3.2** y **3.4**. Unas características del diagrama que es necesario explicar son:

1. **Función** `animate()`: El componente Game Engine tiene una función llamada `animate()`. Esta función es la responsable del renderizado de toda la aplicación, a cargo del **Renderer** descrito en la sección 3.2. Dentro de ella se hacen tareas como: *Revisar y procesar*

los inputs ejecutados por el usuario; actualizar a todos los actores de la escena según corresponda; revisar estado de tareas, como los objetivos de ejercicios o revisar estado de la escena y almacenarlos si es necesario.

2. **Función update()**: Todos los actores deben tener un método `update()`. Este es ejecutado en cada ciclo de renderizado de la aplicación, además de aplicar y manifestar los cambios de parámetros de los actores hechos por la activación de ciertos inputs o tasks.

3. **GLOBALS_VARS**: **Game Engine** tiene un parámetro de tipo diccionario llamado `GLOBAL_VARS`. Este contiene todos los componentes del motor y actúa como una suerte de HashMap. Esto nos permite, por ejemplo, que todos los actores puedan acceder a información de **Inputs Engine**, para que cuando cada uno use su método **update()**. La razón de esto es netamente para evitar referencias circulares y conflictos con el *scope* de ciertas variables.

El resto de atributos serán analizados en el capítulo de implementación.

Capítulo 4

Implementación

En este capíitulo se explica de forma general el problema que motiva este trabajo y la solución propuesta junto con las tecnologías utilizadas.

4.1. Stack Tecnológico del proyecto

En la sección 1.4, se mencionó que por motivos de accesibilidad, Permaculture Simulator debería ser aplicación web. Además, en la sección 2.3 se mencionaron múltiples frameworks como THREE.js o Babylon.js, cada uno con sus pros y cons.

Para el desarrollo del presente proyecto se optó por utilizar THREE.js. La razón principal es por la activa y longeva comunidad que posee. Además, la documentación de dicho framework es más completa.

THREE.js al usar JavaScript trabaja directamente con HTML, por ello la interfaz de usuario de la aplicación fue desarrollada completamente a base de HTML y CSS. Por conveniencia, se importó Bootstrap al proyecto.

4.2. Estructura Directorios

Dado el stack tecnológico del proyecto, se optó por la estructura ilustrada en la **figura 4.1**. En ella se destacan dos archivos: `index.html` y `main.js`. El primero es el archivo que genera la vista principal de la aplicación (**figura A.3**) y el segundo es quien da acceso a todo el código de la aplicación.

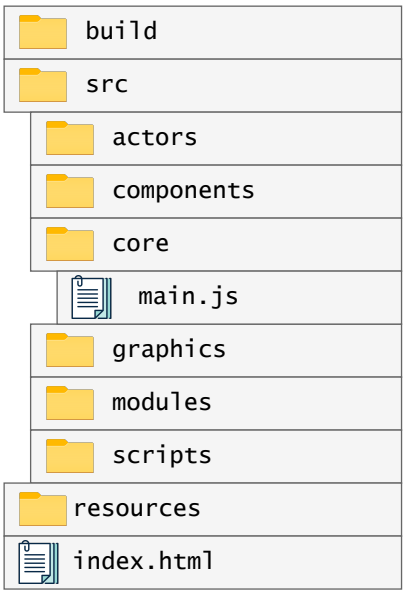

Figura 4.1: Estructura Básica del Proyecto

El rol de cada directorio es el siguiente:

1. `build`: Contiene los archivos de THREE.js.
2. `src`: Contiene el código fuente (JavaScript) de todo el proyecto.
3. `actors`: Contiene a todos los archivos en los que se definen las clases de cada actor.
4. `components`: Contiene distintos archivos con roles varios que van desde la generación de UI, a estructuras de datos que ayudan a gestionar actores.
5. `core`: Contiene los archivos que dan forma al motor del videojuego.
6. `graphics`: Contiene texturas y la definición de los *shaders* que utiliza cara objeto de la aplicación.
7. `modules`: Contiene librerías externas aplicadas sobre el proyecto, como OrbitControls o Stats.js.
8. `scripts`: Contiene módulos que se encargan de gestionar eventos asociados a actores.
9. `resources`: Contiene todos los archivos usados por index.html, como CSS o íconos.

4.3. Interfaz de Usuario

Para la implementación de la interfaz de usuario se utilizó Javascript, CSS y Bootstrap. Usando la librería **RE:DOM** [19], se crearon múltiples clases que representan los distintos componentes de la interfaz. Empleando el modelo de *programación orientada a objetos*, se creó una clase padre llamada `Component` que cuenta con métodos esenciales para la gestión

de lógica y renderización de la *user interface*. Luego, múltiples clases hijas "extienden" dicha clase padre, sobrescribiendo métodos para acomodarse a las necesidades de cada una de ellas.

Haciendo uso de CSS y Bootstrap se crearon menús laterales, pop-ups, y botones estilizados con una codificación de colores amigable que busca guiar al usuario sin tener que dar constantes explicaciones que puedan resultar tediosas para este último.

Listing 4.1 Fragmento de Component.js

```
export class Component {
  constructor({ name = '', componentID, parent = document.getElementById
     ('temporal-container') }) {
    this.id = componentID;
    this.name = name;
    this.callbacks = {};
    this.n_children = 0;
    this.parent = parent;
    this.children = [];
        ...
  }

  function  generateComponent() {...}

  function render() {
    this.generateComponent();
    mount(this.parent, this.el); // Part of RE:DOM Library
    this.setEventListeners(); // Setup of HTML event listeners
  }

  function update() {...}
  function addChildren({ children, props }) {...}
  function setParent(parent) {...}
  function setEventListeners() {...}
  function setParam({ paramName, value }) {...}
    ...
}
```

Esta implementación orientada a objetos permite crear con facilidad nuevos componentes ya que todos comparten los mismos métodos base al ser hijos de la misma clase.

Algunas características destacables de esta implementación son el método `update()`, que permite que la UI se actualice en tiempo real en forma similar a como operan frameworks como *React* [18]. El método `setEventListeners()` permite agregar o remover eventos HTML como `onclick` que son comúnmente empleados en la creación de *User Interfaces*.

Mediante el uso de la librería RE:DOM, se implementaron métodos que permitan actualizar fácilmente la jerarquía de etiquetas o elementos propia de HTML, como `addChildren()` ó `addParent()`.

Algunos ejemplos de componentes que extienden esta clase `Component` son los botones de

la barra de herramientas o la barra de navegación de la aplicación.

La implementación de esta clase y algunos casos de uso pueden ser vistos en el **anexo B**.

4.4. Implementación de Motor de Juego

Tal y como se explicó en la **Sección 3.1**, el motor de videojuego es el pilar fundamental de proyectos como Permaculture Simulator. Con él podemos gestionar como presentamos y como permitimos interactuar al usuario con un entorno virtual.

Su implementación se segmentó en diferentes componentes basado en la arquitectura descrita en el **Anexo A.2**. Cada uno de ellos tiene un rol clave, como la renderización de la aplicación y el manejo de los comandos ingresados por el usuario.

4.4.1. Main

Este componente está a cargo del arranque de la aplicación y su proceso principal: la función de renderizado. Además, se encarga de gestionar el *scope* de ciertas variables.

Uso de Variables Globales

Para evitar conflictos con el uso de variables globales, se optó por crear un archivo llamado `Globals.js`, el cual tiene un diccionario llamado `VARS`. Éste es exportado y por ende, visible por todo otro componente de la apliación que lo necesite.

Listing 4.2 Fragmento de Global.js

```
export let VARS = {
    MAX_SEED_VALUE: 1024,
    FRAME_INTERVAL: 1 / FRAMERATE,
    GLOBAL_DELTA_TIME: 0,
    CURSOR_SIZE: 12,
    FLAGS: {
      MOUSE_ON_GAME: true,
      MENU_IS_OPEN: false,  // Allows to deactivate in-game events if a
          menu is opened
    },
    CURRENT_ACTION: "",
    CURRENT_ACTION_TYPE: "",
    ACTORS: {}; // Access to information of every actor if needed
    (.)
}
```

Arranque de la aplicación

Tal y como se mencionó en la sección [4.2], `index.html` es el archivo que contiene la vista principal de la aplicación. Esta vista consiste en una serie de opciones que le permiten al usuario seleccionar el entorno sobre el cual quieren trabajar, para luego arrancar la aplicación. Cada Opción tiene un `id` único, mediante el cual se le asocia un *event listener* de tipo `onclick`. Esto permite que al clickear el botón, se ejecute una función que crea la escena a partir de cual fue seleccionado.

Listing 4.3 Fragmento de la función de arranque de la aplicación. Esta es asociada mediante un event listener a los botones de la vista principal

```
function handleMenuSelection(event) {
  switch (event.srcElement.name) { // Type of the selected option
    case 'image':
      VARS.MAIN = new Game({mode: "FREE_MODE",...});
      // Generate from Image
      break;

    case 'plane':
      VARS.MAIN = new Game({mode: "FREE_MODE",...});
      // Generate Plane
      break;

    case 'exercise':
      VARS.MAIN = new Game({mode: "EXERCISE",...});
      // Load Exercise from JSON File
      break;
        ...
  }
}
```

Función de Renderizado

La función de renderizado o animación es ejecutada de forma indefinida, mediante la cual se generan fotogramas tras fotogramas. Con tal de garantizar estabilidad, esta función implementa un *framelimit*. En otras palabras, esta función genera un nuevo fotograma en intervalos de tiempo fijos, los cuales son calculados mediante un objeto de THREE.js llamado `THREE.Clock()`. Un método de dicho objeto permite calcular el tiempo que ha pasado entre el llamado actual de la función de renderizado y el anterior. Para más detalles, revisar la **Figura A.1**.

4.4.2. Motor Gráfico

Está a cargo de la renderización de la aplicación y la gestión de la escena. Basándonos en lo ilustrado **figura A.2**, los atributos fundamentales son la cámara, la escena y el renderer,

los cuales están implementados de forma nativa en THREE.js

Además, debe encargarse del sistema de iluminación. THREE.js cuenta con múltiples fuentes de luz que deben agregarse a la escena, ya que si no se hace, simplemente se ve todo negro dentro de ella.

Un fragmento **simplificado** de la implementación del motor gráfico se puede ver en la **Sección 4.4.2**. En donde se inicializan los componentes fundamentales del motor gráfico dentro de la clase `GraphicsEngine`

Listing 4.4 Fragmento de la implementación del motor gráfico de Permaculture Simulator

```
export class GraphicsEngine {
    constructor(fov = 45) {
        this.fov = fov;
        widthContainer = getVW(98);  // % of the container's width
        heightContainer = getVH(96); // % of the container's height
        this.renderer = new THREE.WebGLRenderer();
        this.camera = new THREE.PerspectiveCamera(60,
                        widthContainer / heightContainer,
                        minRenderDistance, maxRenderDistance);
        this.currentScene = new THREE.Scene();
       ...
    }
...

   defineLight(x_coord, y_coord, z_coord, intensity) {
        let light = new THREE.DirectionalLight(light_color,intensity);
        light.position.set(x_coord, y_coord, z_coord);
        light.target.position.set(0, 0, 0);
        light.castShadow = true;
        this.currentScene.add(light);
    }
}
```

4.4.3. Motor de Inputs

Está a cargo de gestionar las interacciones del usuario con la aplicación. Para ello, hacemos uso de los event listeners nativos de JavaScript: **keydown, keyup, mousedown, mouseup y mousemove**. Estos eventos le permiten a la aplicación activar una función cada vez que se presione o suelte un botón del teclado o del mouse, además de detectar movimientos de éste último. Por ejemplo:

- **keydown**: Evento que se dispara cada vez que un **botón del teclado** es presionado.
- **keyup**: Evento que se dispara cuando un **botón del teclado** que estaba siendo presionado, es liberado.
  (Comportamiento análogo para mousedown/mouseup)

Los eventos por si solos no son suficientes dado que tienen ciertas limitantes a la hora de aplicarlos en el motor. Siendo la más importante, que el motor del videojuego procesa los inputs en intervalos fijos, por lo que es posible que un botón sea presionado fuera de dicho intervalo, siendo incapaz de procesarlo.

Por ello, el Motor de Inputs tiene un diccionario llamado `states`, donde el par (llave, valor) viene dado por el identificador (keycode) de un botón de teclado/mouse, y su **estado**. Este último es parámetro de tipo `Integer` con valores entre 0 y 3, donde cada uno representa los siguientes estados:

Valor	Rol	Descripción
0	Inactiva	Estado **base** de cada tecla. Esto quiere decir que el usuario no ha interactuado con dicha tecla.
1	Pulsada	Estado asociado a un botón solo durante el **primer fotograma procesado** desde que fue apretado por el usuario. En ciclos posteriores, esta tecla pasa al estado siguiente: **Presionada**
2	Presionada	Estado asociado a un botón en el resto de fotogramas después de ser pulsado, es decir, cuando el jugador mantiene la tecla presionada. En este estado el Motor de Inputs empieza a llevar registro de cuanto tiempo el botón lleva siendo presionado.
3	Liberada	Estado asociado a un botón después de ser soltado. Al fotograma siguiente de recibir este estado, el Motor de Inputs se encarga de cambiar el estado de esta tecla a Inactiva.

Tabla 4.1: Descripción de los estados posibles que puede tener un botón.

Gestión de Estados

Para la gestión de estados de los botones mezclamos los *events listeners* de JavaScript con unas funciones extras definidas en la clase del Motor de Juego:

1. Cuando el usuario presiona una tecla, se activa la función asociada al *event listener* keydown/mousedown. Por como están implementado los *events* en JavaScript, podemos saber que tecla fue pulsada leyendo su `keycode`. Dicho keycode, es usado para cambiar el estado de la tecla a **presionado** y además es guardado en un array definido dentro del Motor de Inputs: `inputBuffer`.
2. Cuando el usuario suelta una tecla, se activa la función asociada al *event listener* keyup/mouseup. Tomamos el `keycode` de la tecla que activó el botón que activó el evento y forzamos su estado a `released`. Es importante recalcar que si se activó el evento `keyup`, significa que el keycode del botón soltado ya está en el `inputBuffer`
3. Durante el ciclo de renderizado, se ejecuta el método `checkInputBuffer()`. Esta función se encarga de leer iterativamente el `inputBuffer` y utilizamos los valores para leer el dicionario `states`.
    - Si una tecla está en pulsado, se vuelve **pulsada**.

- Si una tecla está en held, se mantiene en **presionado**.
- Si una tecla está en released, se guarda su keycode en un array llamado `releasedKeys`. Dicho array es revisado en el ciclo siguiente y todas las keys que estén allí cambian su estado a **inactivo**.

4. En cuanto a la gestión de los movimientos del mouse, esto se hace mezclando el evento `mousemove` y los **raycasters** de THREE.js. Estos últimos nos permiten disparar vectores que pueden chocar con los actores de una escena. En el contexto de la aplicación. Esto nos permite, por ejemplo, saber sobre que parte del terreno el mouse estaría o el tipo de actor con el que impactó.

Todos estos métodos y atributos son definidos dentro de una clase llamada `InputsEngine`, la cual al ser inicializada, se guarda una referencia al objeto en Global.VARS, de forma que la su información sea accesible para todos los actores o componentes.

Esquema de Controles

Con tal de que la aplicación sea simple de utilizar, se optó por un esquema de controles utilizado por otros programas, de forma que a los usuarios les resulte familiar:

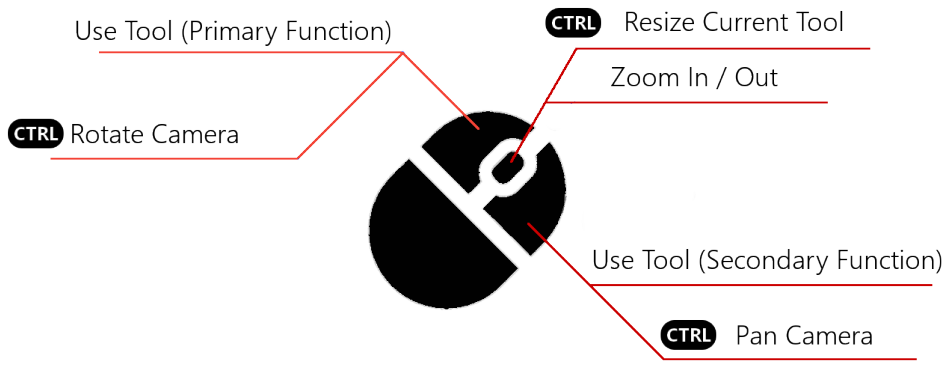

Figura 4.2: Esquema de controles de Permaculture Simulator

4.4.4. State Manager

Es una clase que contiene una `Queue` FIFO y una serie de funciones que permiten navegar dentro de ella. Se dedica a almacenar información que permite **recrear los distintos estados** que ha tenido la escena a lo largo de la ejecución. Para ello:

- Todos los actores tienen un método llamado `parseData`, el cual retorna un objeto/diccionario que contiene toda la información necesaria para recrear al actor en un momento

dado de la ejecución. De la misma forma tienen un método llamado `loadParsedData` que carga la información y manifiesta los cambios en la escena.

- La Queue tiene un tamaño máximo. Si está llena y se crea un nuevo estado, se elimina el estado ingresado más antiguo.
- La Queue lleva registro de la posición del estado actual usando un índice.
- Para moverse entre los estados, hay dos botones en la barra de navegación que permiten cambiar entre los estados. Para ello, los botones reducen o aumentan el valor del índice en 1.
- Cada actor es capaz de **notificar** que recibió cambios que ameritan generar un nuevo estado. Para ello guardan su identificador en un array de State Manager.
- Finalmente, en cada ejecución del ciclo de renderizado se revisa si el array que contiene el nombre de los actores que requieren guardar un nuevo estado. Si el largo de dicho array, es mayor a 0, generamos una copia del estado previo y remplazamos los valores de los actores que solicitaron la creación de un nuevo estado.

4.5. Herramientas

Para la gestión de herramientas, se implementó una clase llamada `Toolbar`, la cual recibe como parametros objetos con la misma estructura definida en la **Sección 3.2.3** y además se encarga de:

- Genera e inyecta el código HTML de los íconos asociados a cada herramienta.
- Asocia una función mediante *event listeners* de tipo `onclick`, la cual le permite a `Toolbar` llevar registro de la herramientas seleccionadas.

Listing 4.5 Función a cargo de gestionar la selección de herramientas

```
1  initializeSelectedTool(toolName) {
2      const toolIcon = document.getElementById(toolName);
3      toolIcon.className = 'toolbar-icon-active';

5      const callbacks = this.icons[toolName].callbacks;

7      Object.keys(callbacks).forEach(callbackType => {
8        this.activeToolCallbacks[callbackType] = callbacks[
             callbackType];
9      });

11     this.activeToolId = toolName;
12     this.activeToolLabel = this.icons[toolName].label;
13     this.activeToolControls = this.icons[toolName].controls;
14     this.activeToolCallbacks.onToolSelection();
15 }
```

El resto de componentes pueden acceder a la información de la `Toolbar`, lo cual es clave para la actualización de los actores.

Siguiendo el esquema de controles de la **Figura 4.6**, si el click izquierdo está activado en el `InputManager`, sabemos que se está usando una herramienta.

Luego, el método `handleInputs()` de `InputManager`, adquiere el ID de la herramienta actualmente seleccionada y con él, accede al callback `onLeftClick` de la herramienta seleccionada y lo ejecuta, aplicando así cambios sobre los actores.

4.6. Actores

Todos los actores de la aplicación deben tener ciertos atributos o métdos, por lo cual se decidió implementar una **clase abstract** llamada `actor` que luego el resto de actores extienden.

Listing 4.6 Clase base para todos los actores

```
export class Actor {
  constructor() {
    if (this.constructor == Actor) {
      throw new Error("Abstract classes can't be instantiated.");
    }
    this.requireSaveState = false;
    this.uniforms = {}:
  }

  setValue(paramName, parameterVal) {
    this.uniforms[paramName] = { value: parameterVal };
  }

  getValue(paramName) {
    let output = this.uniforms[paramName].value;
    return output;
  }

  update() { throw new Error("Method 'update()' must be implemented.");}
  parseData() { throw new Error("Method 'parseData()' must be
      implemented."); }
  loadParsedData(data) { throw new Error("Method 'loadParsedData()' must
       be implemented.");}
}
```

Cada actor, además debe tener un *modelo*, objeto que se añade a las escenas y permite renderizarlos. Estos suelen estar compuestos por una geometria y un material. Hay casos en los que en vez de usar una geometría, se pueden usar imagenes (sprites). Además, cada uno tiene distintos métodos mediantes los cuales el usuario puede interactuar con ellos.

4.6.1. Terreno

Geometría

Generamos la geometría con el objeto `THREE.PlaneBufferGeometry()`. Este objeto necesita como parámetros el tamaño del plano; su altura, y cuantos vértices o segmentos tiene a lo ancho y a lo alto. Estos últimos parámetros dictaminan la cantidad de segmentos tiene el plano, y por ende, cuantos *polígonos* son empleados para su generación. A más polígonos, más "suave" y natural se ve el terreno.

La adquisición de estos parámetros se hace mediante la vista principal de la aplicación: index.html. En ella, el usuario puede seleccionar que tipo de escena quiere utilizar. Esa selección además determina que cosas debemos hacer sobre esa geometría:

- Si el usuario decide generar la escena a partir de un plano, el usuario debe ingresar mediante un formulario el tamaño de éste. Luego, usando esa información podemos calcular la cantidad de segmentos de la siguiente forma:

```
1          const m = Math.max(planeWidth, planeHeight);
2          if (m > VARS.MAX_TERRAIN_POLYGON_COUNT) {
3              heightSegments = planeHeight * (
                   VARS.MAX_TERRAIN_POLYGON_COUNT / m);
4              widthSegments = planeWidth * (
                   VARS.MAX_TERRAIN_POLYGON_COUNT / m);
5          }
```

  Esto es necesario pues se comprobó que utilizar planos con más de 128x128 polígonos compromete demasiado el rendimiento de la aplicación. Además de que no es necesario tener tal nivel de precisión

- Si decide generar la escena a partir de una imagen, el tamaño viene dado por las dimensiones de la imagen. Para el cálculo de los segmentos hacemos lo mismo que en el caso anterior.

  Luego extraemos la información de los píxeles de la imagen, de forma que los utilizamos como un **Heightmap**. Este tipo de técnica se emplea en el área de la Geología, en donde se utilizan imágenes en escala de grises. Donde a mas cerca del color blanco, mayor altitud tendría esa zona del terreno.

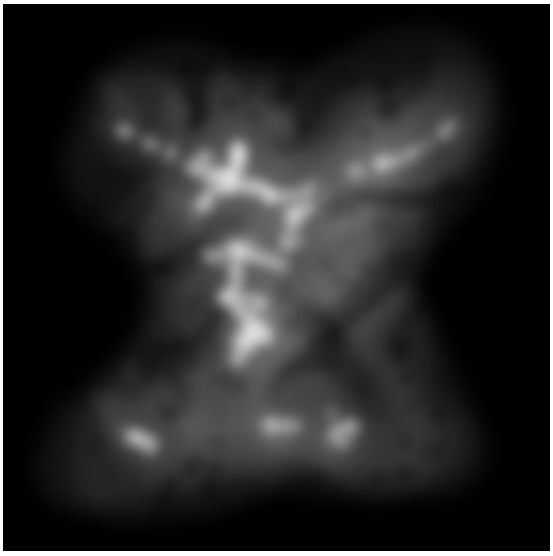

Figura 4.3: Ejemplo de imágenes utilizadas para generar terrenos (Heightmaps)

La información de cada pixel viene dado por el esquema RGB, es decir la "cantidad" de rojo, verde y azúl empleado para generar el color que tiene. Por lo que para transformar eso en color simplemente empleamos la ecuación $(r + g + b)/(3 * 255)$

- Si el usuario decide generar el terreno de forma aleatoria, usamos la librería de **Perlin Noise**. Empleando un sistema similar a la generación mediante imágenes, es decir, utilizar una estructura de datos para generar altura para los vértices del plano, utilizamos las funciones de esta librería para generarlas.
- Si decide generar la escena desde un ejercicio, la aplicación carga la información del archivo y simplemente aplica la data de él para la creación de la geometría.

Material

La generación de material, y por ende, la aplicación de texturas, se hace mediante el objeto `THREE.RawShaderMaterial`. Mediante él, podemos intervenir directamente en la *Render Pipeline* (o Tubería de Renderizado en español), proceso llevado a cabo en la GPU que se encarga de traducir los gráficos 3D a lo que vemos en pantalla.

Usando funciones programadas en **GLSL**, podemos pasar información definida en JavaScript directamente a la tarjeta de video. Esto se hace mediante `Uniforms`, parámetro definido en la clase base: `Actor` mencionada en [4.6]. Mediante este objeto, podemos entregar distinta información, como texturas o la posición del mouse sobre el terreno, la cual es empleada para dibujar un cursor en él.

Modelo

Finalmente, utilizando la geometría y el material creados con los procesos previamente descritos, se puede definir el modelo del terreno mediante el objeto `THREE.Mesh(geometry,`

material).

Gestión de Eventos

Utilizando el raycaster integrado en `InputsEngine`, podemos seleccionar distintas partes del plano y adquirir información de él. Existen distintas **herramientas** que permiten editar áreas marcadas del terreno. Por ejemplo:

1. **Create Hill**: Utilizando el raycaster, recupera el identificador de los vértices que estén en un radio rrespecto a la posición del cursor. Luego, a la altura actual sumamos el valor retornado por la siguiente función:

```
function quad(cursorRad, distanceCursorVertex) {
 return 1.0 - Math.pow(distanceCursorVertex / cursorRad, 2) / 5
     .0;
}
```

2. **Lower Terrain**: Se aplica el mismo sistema de *Create Hill*, solo que en vez de sumar, restamos.
3. **Normalize Terrain**: Toma la altura de todos los vértices en área, calcula el promedio y gradualmente hace que sus alturas se acerquen al promedio. Esta herramienta permite suavizar la forma de los terrenos cuando se tornan muy pronunciadas debido a la imprecisión de las herramientas en ciertos escenarios.

   También puede ser utilizada para aplanar superficies.
4. **Modify Manually Selected Vertices**: Esta herramienta vuelve visible los vértices del plano y le permite al usuario seleccionarlos manualmente. Una vez marcados, el usuario puede usar las flechas del teclado para modificar la altura de los vértices.
5. **Show Isolines**: Mediante el uso de shaders, la aplicación segmenta el terreno dibujando curvas del nivel sobre el terreno
6. **Show Heightmap**: Similar a la herramienta anterior, colorea el terreno entero aplicando una escala de colores en función de la altura de cada vértice del terreno.

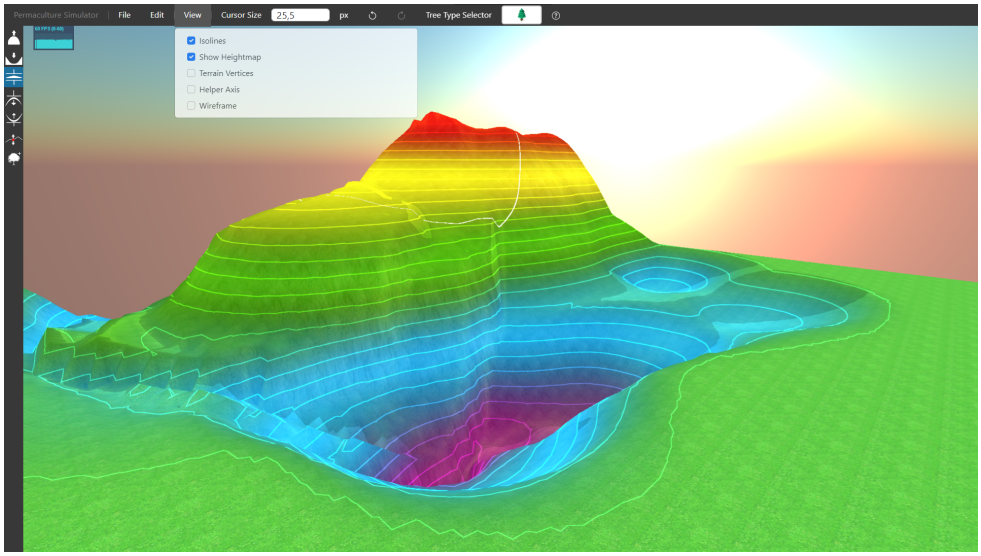

Figura 4.4: Terreno Mostrando su Heightmap y Curvas de Nivel en Permaculture Simulator

4.6.2. Árboles y Plantas

Generación de Modelo

Tal y como se explicó en la **Sección 3.2.10**, los árboles y plantas son sprites. Esto nos permite agregar rápidamente nuevas variantes de este tipo de actor, además disminuir la carga sobre el equipo al ser menos costoso renderizar una imagen que un modelo 3D.

Para generar el sprite de estos actores simplemente usamos el objeto `THREE.Sprite`. Sobre el cual nosotros podemos establecer la posición, escala y la imagen que cargamos.

Gestión de Eventos

El usuario previo a agregar un árbol o planta puede seleccionar su tipo desde la barra de navegación

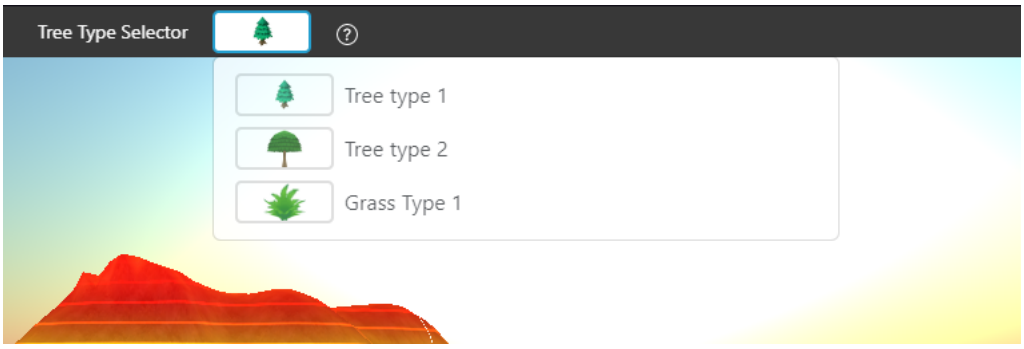

Figura 4.5: Menú de selección de árboles y plantas.

Una vez seleccionado el tipo, el usuario puede agregarlo a la escena simplemente clickeando sobre el terreno. El raycaster nos hará saber las coordenadas en las que poner el actor.

Para removerlos, el usuario debe hacer click derecho sobre un actor de tipo árbol o planta. Nuevamente, usando el raycaster podemos determinar si el usuario tocó uno, y si es el caso, entonces lo elimina de la escena con el método `THREE.Scene().removeObjectByName(id)`

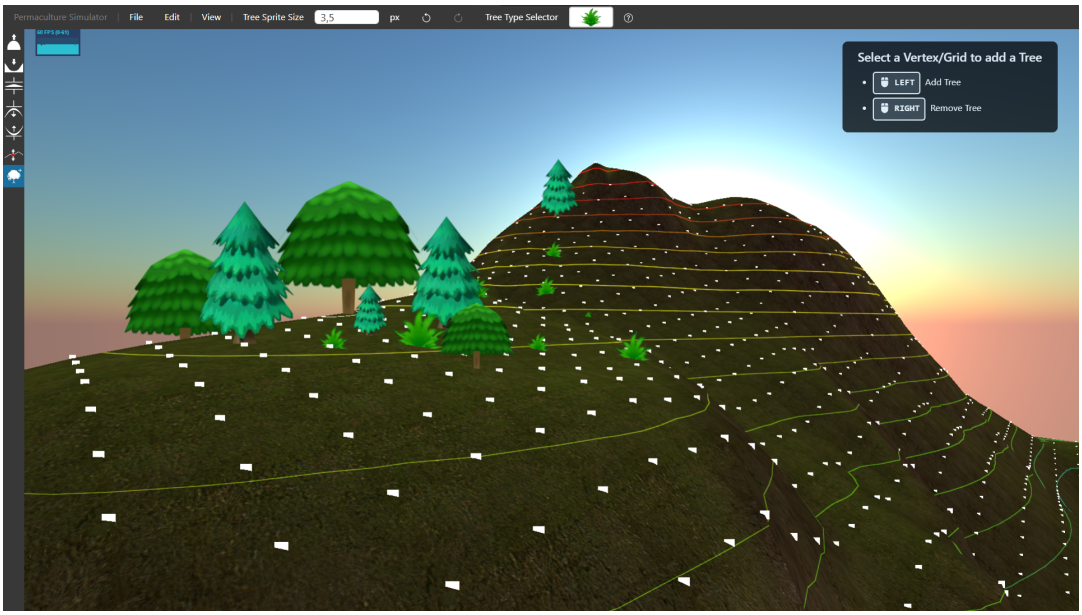

Figura 4.6: Distintos tipos de árboles y plantas con factores de escala variable agregados en la escena.

Los árboles y plantas también responden ante los cambios hechos sobre el terreno. El método `update()` de este tipo de actores verifica si al momento de su invocación está siendo utilizada una herramienta que edite el terreno. Si es el caso, toma la posición del árbol y ejecuta la función asociada a dicha herramienta y calcula la altura que debería tener.

4.6.3. Planos de Agua

Por restricciones de tiempo, la implementación de este actor no fue finalizada, pero si se logró poner a prueba los principios base planteados en la **Sección 3.2.9**.

Generación de Modelo

La generación de la geometría de los planos de agua consiste en replicar la geometría del plano. Es decir, deben tener el mismo tamaño y cantidad de segmentos.

El material es una textura de agua cargada mediante un `THREE.RawShaderMaterial()`. Mediante los `Fragment Shaders` del programa GLSL, hacemos que la textura se mueva de arriba a abajo, dando la sensación de movimiento.

Gestión de Eventos

La herramienta **Add Water Node**, permite al usuario marcar una zona del terreno en la que le gustaría añadir agua, y luego definir la altura a la que el agua debería llegar.

Tras eso, la aplicación utiliza la información disponible en la geometría, especificamente, como están conectados los vértices. Entonces, tomamos el ID del **vértice del plano del terreno** más cercano al punto seleccionado y recorremos todos los vértices que hay alrededor de él de forma recursiva. La forma en la que están conectados los vértices del plano puede ser interpretada como un grafo, lo que nos permite usar el algoritmo **Depth-First Search**. De esta forma, recopilamos los IDs de todos los vértices que tengan una altura inferior o igual a la altura a la que el agua debería estar, y nos detenemos cuando nos topemos con un vértice que tenga una altura mayor.

Como la geometría del plano del agua fue hecha en función de la del terreno son los mismos, podemos usar los IDs adquiridos para modificar la altura de esos vértices y volverla equivalente a la altura a la que debe estar el plano de agua. Dando así con éxito la ilusión de que un punto de agua fue creado.

En caso de que la geometría del terreno sea modificada mediante las herramientas de la aplicación, debemos hacer el mismo llamado recursivo descrito previamente para encontrar los IDs de los vértices que aún satisfacen la condición de altura. Eso si, utilizamos los IDs de los vértices anteriormente adquiridos, lo cual nos permite agilizar la adquisición de los puntos que cumplen la condición.

4.7. Ejercicios

Bajo la estructura definida en la **Sección 3.2.4** se crearon dos componentes para la gestión de la creación y carga de ejercicios:

4.7.1. Exercise Creator

Este objeto se encarga de la generación del código HTML asociado al menú de creación de ejercicios. Dentro de este menú, el usuario puede definir el nombre del ejercicio y una descripción asociada a éste. Además, el usuario puede definir múltiples objetivos, cada uno con nombre, tipo, modificadores y descripción.

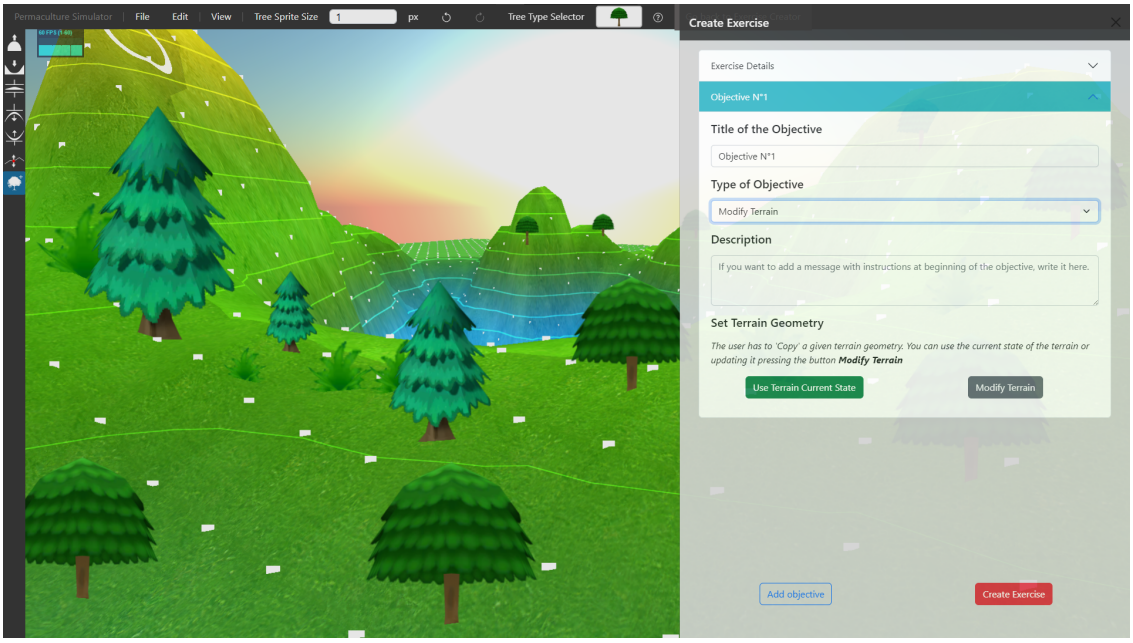

Figura 4.7: Menú a cargo de la creación de ejercicios

Toda la información que el usuario agrega mediante el formulario visto en la **Figura 4.7** es guardada en un diccionario. Cuando se termina de añadir objetivos, se crea el botón `Create Exercise`, lo cual toma la información del diccionario previamente mencionado y la guarda en un archivo JSON que puede ser descargado por el usuario. Finalmente, este archivo puede ser compartido con terceros para que lo carguen mediante la aplicación y puedan realizar el ejercicio.

4.7.2. Exercise Loader

Mediante un `<input type=‘‘file’’>` el usuario puede subir un archivo de ejercicio a la plataforma. La información contenida en dicho archivo, es utilizada para inicializar el objeto `ExerciseLoader`, el cual sigue la misma estructura ilustrada en la **Figura 3.8**.

El archivo creado por **Exercise Creator** almacena todos los objetivos en un array. Cuando se carga un objetivo, se lee su tipo, se determina que función es necesaria para verificar el estado de dicho objetivo y esta es guardada en el campo `this.checkObjectiveCallback` de `Exercise Loader`.

Esta función luego se vuelve accesible desde la **función de renderizado**, por lo que cada vez que se aplique un cambio sobre la escena, podemos llamar la función y verificar si el objetivo fue completado o no.

Cada vez que se completa un objetivo, se borra el objetivo del array, se pasa al siguiente y se repite hasta que no queden más objetivos. Cuando se llega a este punto, se determina que el ejercicio fue completado.

4.7.3. Ejemplo de Uso: Tutorial de la Aplicación

Para poner a prueba esta implementación, se diseñó un tutorial cuyo objetivo principal es enseñar a los usuarios como controlar la cámara y como utilizar las distintas herramientas que la aplicación ofrece. El objetivo final de este tutorial consiste en poner a prueba todo lo enseñado en los pasos previos.

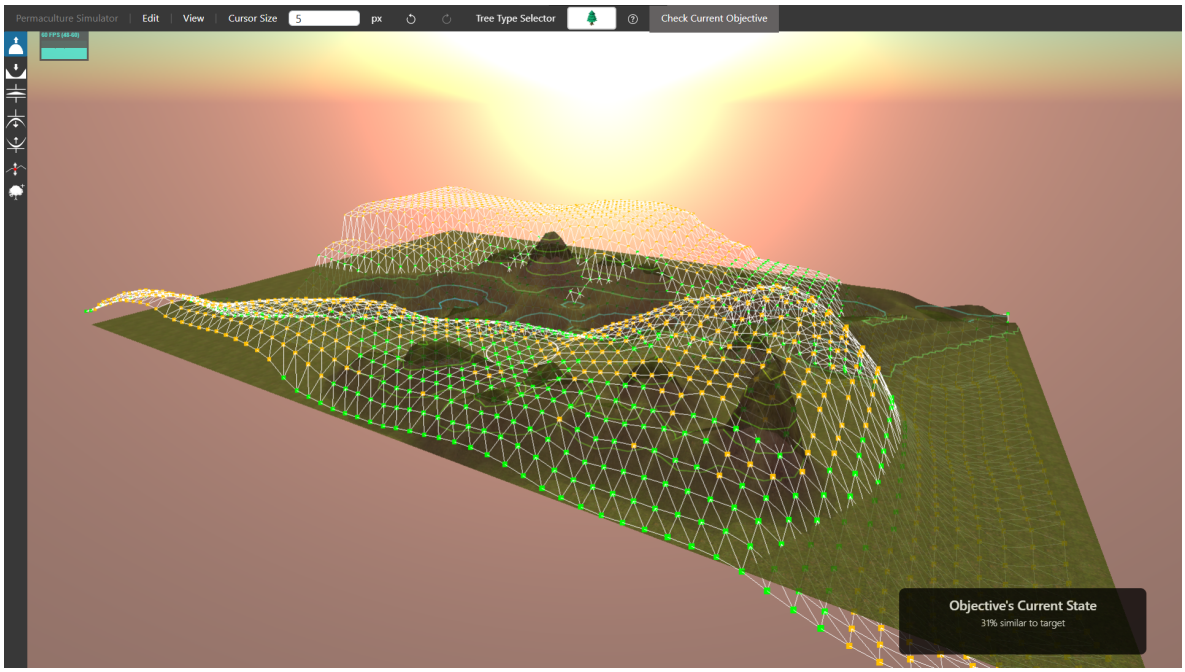

Figura 4.8: Último objetivo del tutorial: Replicar un terreno utilizando las herramientas de la aplicación.

Capítulo 5

Validación

En este cápıtulo se explicará como se llevó a cabo el proceso de validación de Permaculture Simulator. Este capítulo se divide en las siguientes secciones:

- Validación de los Principios Conceptuales de la Aplicación.
- Validación con usuarios y sus resultados.

5.1. Validaciones de conceptos

A lo largo del desarrollo de este proyecto, se llevaron a cabo múltiples reuniones mensuales con Andrew Millison, profesor del departamento de horticultura de la Oregon State University. En ellas, el actuaba como un consejero, ayudándonos a establecer potenciales características del proyecto y la prioridad con las que estas podrían ser implementadas. Algunos ejemplos de esto son:

- Mostrar curvas de nivel y heightmap del terreno
- Agregar árboles y plantas a la escena como si esta fuese una "maqueta".
- Desarrollar la aplicación bajo la premisa de que algunos usuarios no tendrán mucha afinidad con computadores, por lo que la simplicidad de uso es prioridad.
- Poder definir parámetros por zonas en el terreno, como la permeabilidad del suelo.

Dentro de estas reuniones, al ver el progreso del proyecto, se comenzó a discutir el potencial de herramientas como Permaculture Simulator y lo útil que podría ser tener simuladores de alto nivel más orientados a la investigación.

5.2. Validación con Usuarios

Durante el desarrollo del proyecto se llevaron a cabo dos ciclos de validación. El perfil de *tester* de estos ciclos eran personas que no estaban necesariamente vinculadas a la permacultura, ya que el proposito de este tipo pruebas era analizar el rendimiento y usabilidad de la aplicación.

En cada iteración, los usuarios debían responder un formulario hecho en Google Forms que era accesible de la aplicació. Al final de este, tenían una caja de comentario libre en donde podían dejar todas sus observaciones personales.

5.2.1. Primer Ciclo de Validación

Este se llevó a cabo la semana del 29 de agosto del año 2022. En él, la aplicación aún estaba en una fase temprana y solo era capaz de generar un terreno aleatorio y levantar/bajar zonas del terreno.

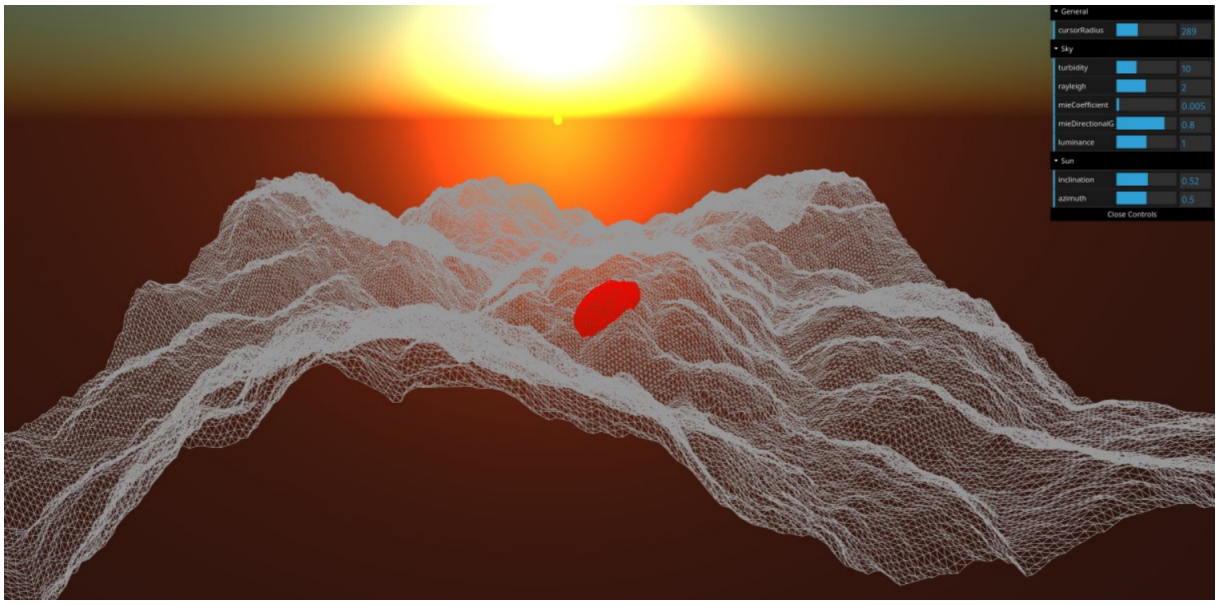

Figura 5.1: Estado de la aplicación durante el primer ciclo de validación

Las preguntas realizadas y respuestas recibidas durante este ciclo fueron:

1. **Desempeño de la Aplicación**: Se agregó un contador de FPS (Frames Per Second) en una esquina de la aplicación. con él, los usuarios podían ver el promedio de fotogramas que la aplicación fue capas de generar durante su ejecución.

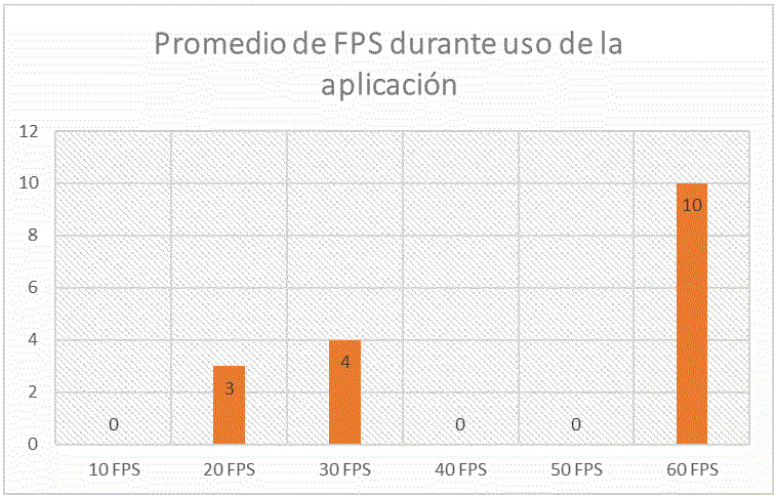

Figura 5.2: Resultados: Desempeño de la aplicación. Primer Ciclo de Validación

2. **Hardware del Usuario**: Se preguntó a los usuarios si su tarjeta de video era integrada o dedicada.

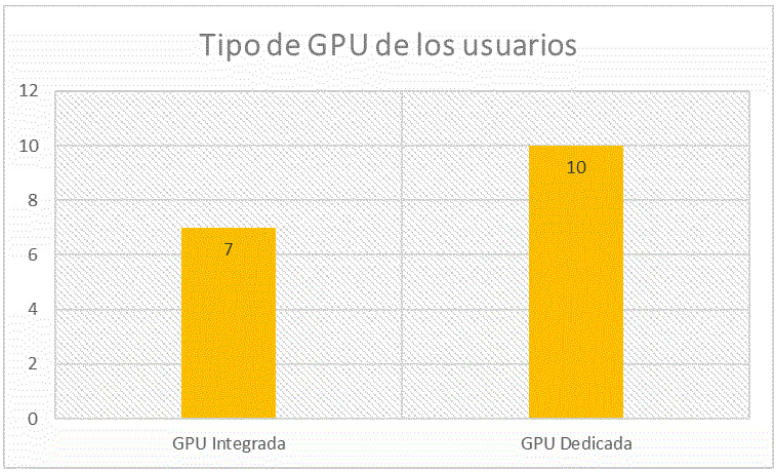

Figura 5.3: Resultados: Hardware de Usuario. Primer Ciclo de Validación

3. **Esquema de Controles de la Aplicación**: Los usuarios evaluaban en escala de, 1 a 5, que tal les parecían los controles de la aplicación.

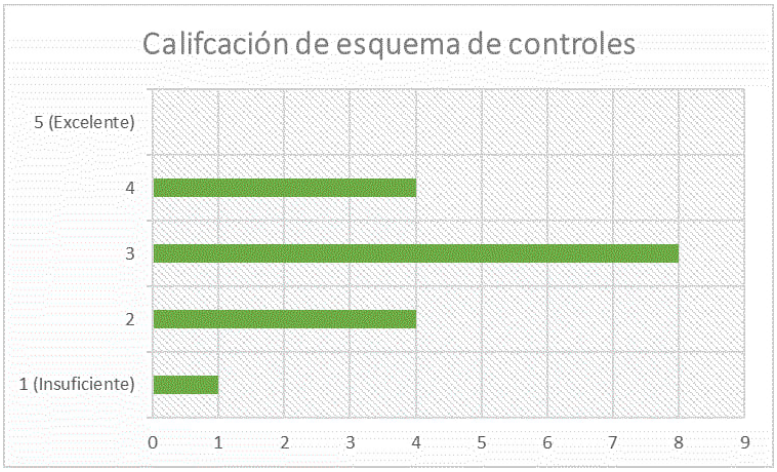

Figura 5.4: Resultados: Esquema de Controles. Primer Ciclo de Validación

4. **Comentarios Libres de Usuarios**: Durante esta validación, los comentarios más relevantes y usados para definir prioridades para el siguiente ciclo fueron los siguientes:
   - *Añadir texturas a la aplicación debe ser hecho cuanto antes. El terreno al ser solo un "tejido" cuesta mantener la noción de profundidad y por ende, trabajar sobre el terreno*
   - *Las herramientas de edición son poco precisas, debería existir una que permita seleccionar puntos manualmente para luego poder editarlos.*

5.2.2. Segundo Ciclo de Validación

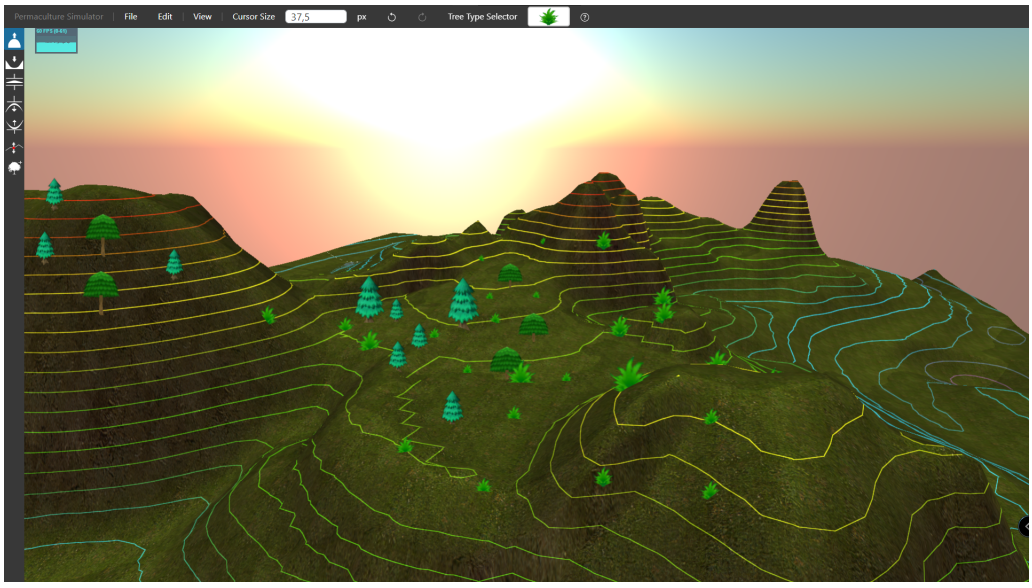

Figura 5.5: Estado de la aplicación durante el segundo ciclo de validación

Este ciclo de validación se llevó a cabo en la semana del 13 de febrero de 2023. En esta iteración de la aplicación, esta contaba con las siguientes características:

1. Vista principal desde la cual el usuario podía seleccionar el tipo de escena sobre la cual quería trabajar, las cuales consistían en: Generar terreno desde imagen, generar terreno desde un plano, generar terreno aleatorio.
2. Uso de texturas y materiales sobre el terreno. Lo cual permitía dibujar curvas de nivel o su heightmap.
3. Nuevas herramientas de edición de terreno, como el normalizador de alturas, selección manual de puntos, aplanador de terreno, entro otros.
4. Optimizaciones de código, como aplicación de restricciones sobre la cantidad de polígonos que podían ser empleados para la generación del terreno o la distancia de dibujado que tenía la cámara.
5. Capacidad de añadir árboles y plantas de distintos tipos o tamaños a la escena.
6. Tutorial que enseña al usuario como utilizar la aplicación.

En cuanto a las preguntas, se hicieron las mismas del ciclo anterior. Los resultados fueron los siguientes:

- **Desempeño de la Aplicación**:

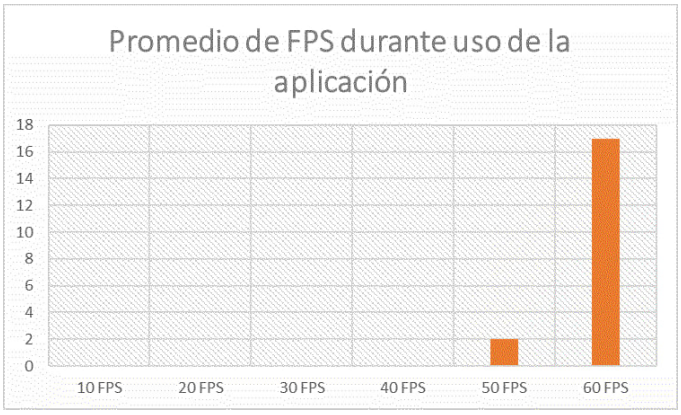

Figura 5.6: Resultados: Desempeño de la aplicación. Segundo Ciclo de Validación

- **Hardware del Usuario**:

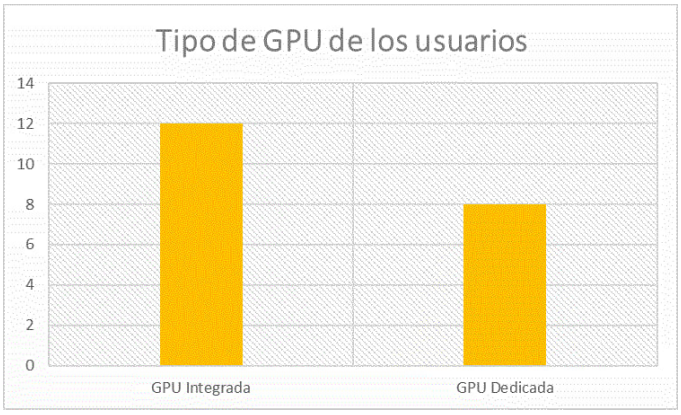

Figura 5.7: Resultados: Hardware de Usuario. Segundo Ciclo de Validación

- **Esquema de Controles de la Aplicación**:

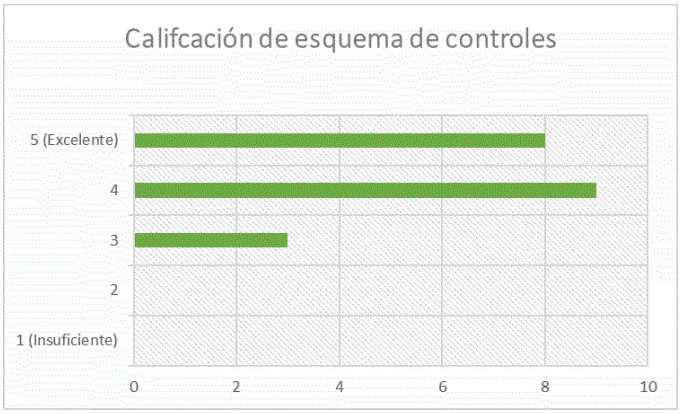

Figura 5.8: Resultados: Esquema de Controles. Segundo Ciclo de Validación

- **Comentarios Libres de Usuarios**: Durante esta validación, los comentarios más relevantes y usados para definir prioridades para el siguiente ciclo fueron los siguientes:
  - *Añadir hints visuales al usuario que lo guíen al agregar árboles o plantas. Cuando uno reajusta sus tamaños, no hay nada que te indique visualmente cual es su tamaño hasta que ya lo agregas a la escena.*
  - *El objetivo final del tutorial es demasiado complicado respecto a los pasos previos. Se debería hacer más simple o agregar herramientas más precisas.*
  - *El terreno a veces puede resultar pequeño, y la herramienta de rescalado del terreno realmente no soluciona el problema pues solo aumenta la distancia que hay entre los vértices.*
  - *Sería bueno tener una herramienta como un pincel que permita aplicar distintas texturas sobre el terreno para que no solo sea pasto y roca.*

Capítulo 6

Conclusión

Para concluir, presentamos un resumen de los resultados logrados (**Sección 6.1**), una discusión de sus limitaciones (**Sección 6.2**) y de sus extensiones futuras (**Sección 6.3**).

6.1. Resultados Logrados

El resultado de este trabajo es una aplicación web capaz de hacer todo lo que es posible con un AR Sandbox desde cualquier computador, además de la creación de ejercicios que podrían ser utilizados por profesores para hacer evaluaciones en un curso de permacultura.

En términos del diseño de la aplicación, se logró crear una arquitectura flexible y modular utilizando JavaScript y THREE.js. Esta ha ido mutando a lo largo del desarrollo del semestre con tal otorgar herramientas en caso de que en el futuro alguien desee implementar nuevas características a la aplicación. Un ejemplo de esto es la existencia del Task Buffer, que aunque en estos momentos no está siendo utilizado en la aplicación, si daría soporte para modificaciones futuras.

Siguiendo los modelos planteados en la **Sección A.2** se lograron implementar un motor de videojuego simple capaz de gestionar y renderizar múltiples componentes que en conjunto, permiten al usuario tener lo que básicamente es una virtualización de un AR Sandbox sin tener que invertir dinero en su construcción. Creación de ejercicios, uso de árboles o plantas en escenas, generación y edición de terrenos son parte de las herramientas implementadas en Permaculture Simulator que le permiten al usuario crear e ilustrar una gran variedad de escenarios.

A pesar de que el proyecto no se pudo poner a prueba en un curso de permacultura, si se pudo validar el estado del proyecto en distintas instancias. Las reuniones con el profesor Andrew Millison permitieron establecer las reglas y entender conceptos claves de un campo tan ajeno al saber popular como podría ser la permacultura. Sin esa información, hubiera sido imposible diseñar una aplicación que intervenir directamente en como se llevan a cabo las cursos de esa área.

La validación con usuarios fue clave para poder establecer cuales son las restricciones técnicas a las cuales un proyecto de esta escala debe apegarse para poder garantizar la accesibilidad por el mayor público posible.

6.2. Discusión

Si bien es cierto que en estos momentos Permaculture Simulator se puede utilizar de forma similar a un AR Sandbox, hay características que fueron planteadas como un requisito en el producto final que no pudieron ser implementadas. Permitir al usuario agregar agua a la escena y ver como distintos actores interactúan con ella es algo esencial para el estudio de la permacultura. Tal característica no pudo ser implementada por el tiempo requerido dada su complejidad. Respecto al resto de objetivos planteados en la sección:

1. Se logró implementar con éxito una serie de herramientas que permiten a los usuarios de la aplicación crear terrenos y moldearlos.
2. Es posible dibujar las curvas de nivel en tiempo real para cada terreno creado dentro de la aplicación
3. La simulación en tiempo acelerado de los actores no fue implementada, dado que para ello necesitamos tener agua en la escena. De cualquier forma, en el código del proyecto está documentado como hacer este tipo de simulaciones utilizando la implementación actual del Game Engine y el uso de las clases abstractas de actores.
4. Se logró establecer una gran compatibilidad al programar el proyecto con JavaScript y THREE.js. Aunque durante la la fase de validación de usuarios, se descubrió que hay versiones de Firefox en que la aplicación no puede ser ejecutada.

A lo largo del proyecto surgieron múltiples problemas que se transformaron en valiosas experiencias para futuros proyectos de desarrollo de software, uno de estos tiene que ver con los procesos de validación.

Inicialmente el plan era validar la aplicación con estudiantes de cursos de permacultura, idealmente de aquellos impartidos por *Andrew Millison*. Pero en cierto punto del desarrollo del proyecto, se decidió refactorizar todo el código y reimplementar componentes mayores desde cero, lo cual supuso un desfase considerable respecto al calendario de trabajo propuesto.

Como referencia, la **figura 6.1** contiene la Carta Gantt diseñada para la propuesta avanzada del presente trabajo escrita en junio de 2022, la cual marca un claro contraste con el desarrollo descrito a lo largo de todo el presente informe.

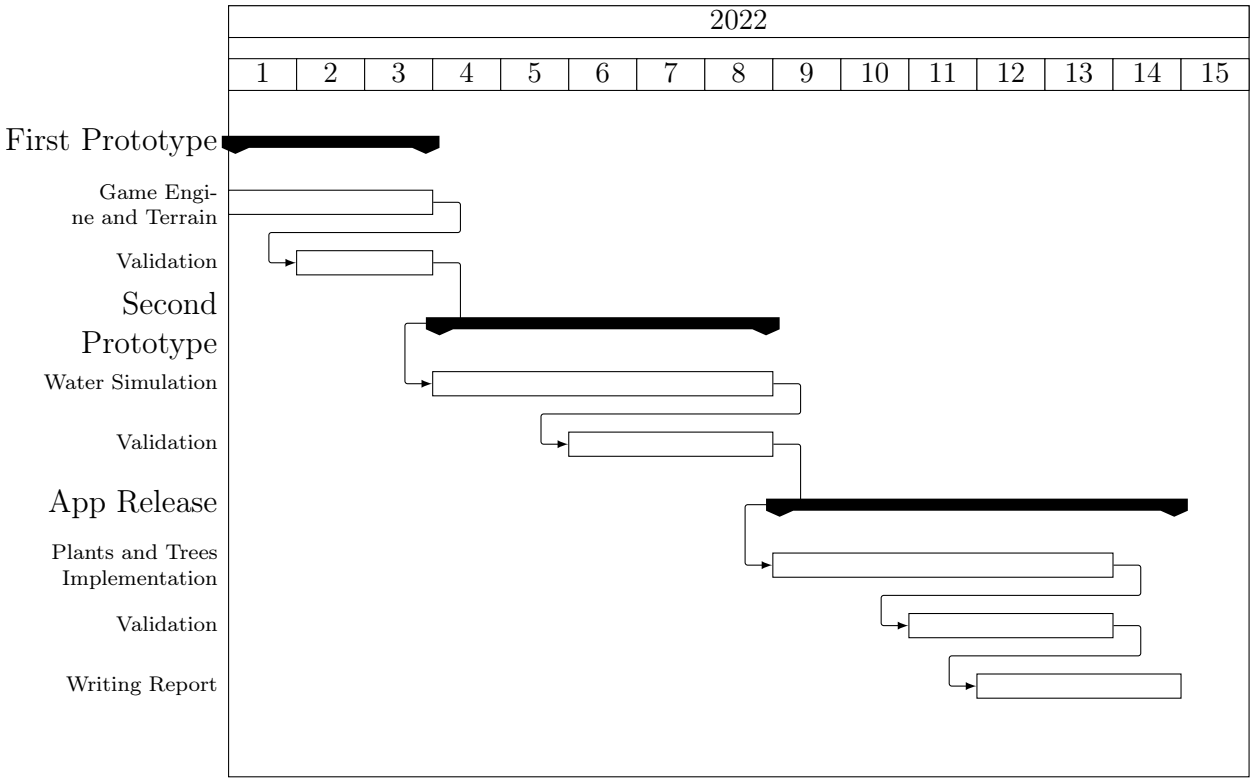

Figura 6.1: Carta Gantt creada para la propuesta avanzada del trabajo de título.

Como se mencionó previamente, gran parte de estas "discrepancias" surgieron por refactorizaciones de código y a nuevos descubrimientos que se hicieron respecto a la API de renderizado que suponían modificaciones mayores durante el desarrollo del proyecto.

No solo la refactorización de código fue el responsable de las discrepancias entre el plan propuesto y la ejecución de éste, sino que el tiempo previsto para la implementación de ciertas características del proyecto eran un tanto "irreales". Un ejemplo de esto son las cinco semanas propuestas para la implementación de simulaciones de agua. Este tema es famoso en la computación gráfica dada su complejidad y por la inexperiencia en el tema se propuso un plazo que, visto ahora, resulta imposible de cumplir.

Finalmente, se aprendió mucho sobre la relevancia de tener claro de principio a fin, cuales son los objetivos de un proyecto de software. El no estar pendiente de ello como es debido, resulto en la refactorización de código en primer lugar. Durante el desarrollo, al ver que el tiempo se acababa y los objetivos no estaban ni cerca de ser completados en las fechas propuestas, se decidió cambiar el enfoque a crear un entorno que permitiera una fácil creación e implementación de herramientas para que quien retomara el desarrollo del proyecto en el futuro, pudiera crear el simulador, lo cual claramente dista mucho del objetivo inicial.

Eventualmente, con más tiempo y más experiencia, se pudieron usar esas herramientas implementadas para poder dar forma al estado actual de Permaculture Simulator. Si bien es cierto que se puede decir que el objetivo original ha sido "completado" con éxito, muchas cosas pudieron salir mal por tomar una ruta que era radicalmente distinta a la inicial.

6.3. Trabajo Futuro: Potenciales Ideas e Implementaciones

Durante todo el desarrollo del proyecto y el proceso de validación, surgieron múltiples ideas que no fueron implementadas en la aplicación, principalmente por falta de tiempo o porque a momento de la implementación, no eran viables. Las ideas planteadas para desarrollos futuros del proyecto son:

1. **Implementar el simulador de agua**: Tal y como se mencionó en la sección [6.2], esta característica clave del simulador no pudo ser implementada, pero esta ya fue diseñada y planeada, solo resta programarla.

   También es importante mencionar que durante las fases finales del desarrollo, se descubrió que la librería Cannon.es, un *port* de una librería deprecada llamada Cannon.js, cuenta con una implementación del modelo de fluidos SPH [7]. Este podría ser potencialmente útil para desarrollar el simulador de agua.

2. **Generación de Terrenos por Fragmentos**: En el último ciclo de validación, recibimos más de un comentario señalando que el espacio sobre el cual el usuario puede trabajar, a veces puede tornarse pequeño.

   Por ello, se planeó un modelo de terreno separado por trozos o chunks. La principal razón por la cual el plano tiene el tamaño que tiene, es por las limitantes técnicas asociadas a la cantidad de polígonos que puede tener. Pero, si el terreno está implementado por chunks, entonces podemos implementar distintos tipos de técnicas para tener un terreno muchisimo más grande que el actual, sin comprometer el rendimiento de la aplicación. Por ejemplo:

   - Si la cámara está mirando en la dirección opuesta a donde está un fragmento de terreno, este no es cargado en la escena, ya que no es necesario pues no es visible para el usuario.
   - Si un fragmento del terreno está demasiado lejos de la cámara simplemente no se renderiza. La otra opción es aplicar el concepto de *Level of Detail* para generar una versión *Low Poly* de dicho fragmento y mostrarla. Ya que al estar tan lejos, no importa si se renderizado con utilizando una menor cantidad de polígonos pues no se notaría

     También es posible ocultar estas imperfecciones como distintas técnicas, como el uso `THREE.Scene.Fog()` (Detalles en **Anexo A.4**)

   En la rama de desarrollo del proyecto existe un prototipo de este concepto, el cual es ilustrado en la **figura 6.2**. En ella se ve como se generaron múltiples planos que en conjunto dan forma a un terreno. Dado que son múltiples objetos, es posible que se generen grietas entre ellos, por lo que para una futura implementación, se necesitaría generar parches o puentes que conecten cada fragmento.

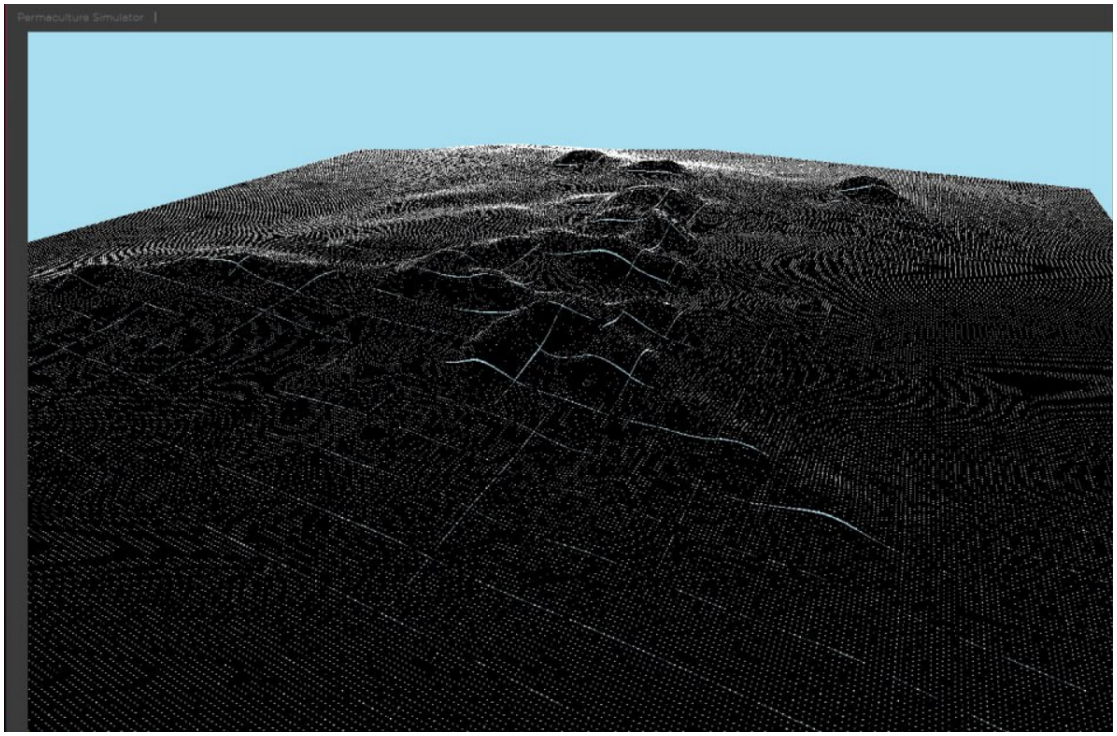

Figura 6.2: Prototipo: Terreno generado a partir de múltiples geometrías

3. **Crear menú para añadir distintos tipos de árboles o plantas**: La implementación de esos actores se basa en el uso de *sprites*. Por ende, para agregar un nuevo tipo a la aplicación realmente solo necesitamos una imagen. Bajo esta premisa, podemos implementar un formulario con HTML que le permita al usuario subir una fotografía de un árbol, definir modificadores como la cantidad de agua que necesita y finalmente usarlo en la aplicación.

4. **Agregar un modo "campaña"**: La aplicación actualmente cuenta con un tutorial que le enseña al usuario como utilizarla. Si utilizamos las herramientas de creación de ejercicios, podemos crear una serie de ejercicios que enseñen los principios básicos de la permacultura y estén disponibles en la plataforma para todo aquel que desee aprender.

5. **Soporte para distintos lenguajes**: La generación de la interfaz está a cargo de distintas funciones de JavaScript utilizando objetos JSON. Si se crean distintas variantes de estos objetos, una por cada idioma, sería posible crear una plataforma accesible para usuarios de todo el mundo.

Bibliografía

[1] **Andrew Millison**. *This Farm Design Can HEAL the PLANET. Youtube*, https://www.youtube.com/watch?v=V3tpaIf6Jcc&t=76s, [August, 2021].

[2] **Andrew Millison**. *Youtube Channel. Internet*, https://www.youtube.com/c/amillison, [Cited 2022 Apr 28].

[3] **Andrew Millison**. *Keyline in the AR Sandbox #1: Contour Lines and Water Flow. Internet*, https://www.youtube.com/watch?v=yKGvj50r_6w, [Cited 2022 Jul 17].

[4] **ARM**. *What is a Game Engine? Lode's Computer Graphics Tutorial*, https://www.arm.com/glossary/gaming-engines, [Cited at 2022 July 17].

[5] **Babylon.js**. *Welcome to Babylon.js. W3*, https://www.babylonjs.com, [Cited 2023 January 13].

[6] **Bullfrog Production**. *Populous.* [1989].

[7] **Cannon.ES**. *Ejemplo de implementación, SPH.* https://pmndrs.github.io/cannon-es/examples/sph, [Cited 2023 April 21].

[8] **Eric Lofgren and Nina H. Fefferman**. *The untapped potential of virtual game worlds to shed light on real world epidemics. The Lancet*, https://www.thelancet.com/journals/laninf/article/PIIS1473-3099(07)70212-8/fulltext, [September, 2007].

[9] **Hydronia**. *Riverflow2D. Internet*, https://www.hydronia.com/riverflow2d, [Cited 2022 Jul 21].

[10] **Learn OpenGL**. *Heightmap: definition. W3*, https://learnopengl.com/Guest-Articles/2021/Tessellation/Height-map, [Cited 2023 January 13].

[11] **Lodev**. *Raycasting: What it is? Lode's Computer Graphics Tutorial*, https://lodev.org/cgtutor/raycasting.html, [Cited 2022 Jul 21].

[12] **Northern School of Permaculture**. *Permaculture Design Courses Topics. Internet*, https://northernschool.info/node/75, [Cited 2022 Jul 17].

[13] **OpenGL**. *Home: OpenGL. Internet*, https://www.opengl.org, [Cited 2023 Apr 07].

[14] **Permaculture News**. *What is Permaculture? Internet*, https://www.permaculturenews.org/what-is-permaculture/, [Cited 2022 Apr 28].

[15] **PermacultureUK**. *Permaculture Course Listings. Internet*, https://www.permaculture.org.uk/education/course-listings, [Cited 2022 Jul 17].

[16] **QGIS**. *Welcome to QGIS Site. Internet*, https://www.qgis.org/en/site/, [Cited 2022 Jul 21].

[17] **Raouf Touti**. *Perlin Noise: A Procedural Generation Algorithm. Raouf's Blog*, https://rtouti.github.io/graphics/perlin-noise-algorithm, [Cited 2023 January 13].

[18] **React**. *What is React? Internet*, https://react.dev, [Cited 2023 Apr 07].

[19] **RE:DOM**. *Introduction: What is RE:DOM? Internet*, https://redom.js.org/#introduction, [Cited 2023 Apr 07].

[20] **THREE.js**. *THREE.js: Homepage. W3*, https://threejs.org, [Cited 2023 January 13].

[21] **Vulkan**. *Home: Vulkan. Internet*, https://www.vulkan.org, [Cited 2023 Apr 07].

[22] **WebGL**. *Getting Started: What is WebGL?* https://www.khronos.org/webgl/wiki/Getting_Started, [Cited 2023 January 13].

[23] **WebGPU**. *WebGPU: Introduction. W3*, https://www.w3.org/TR/webgpu/, [Cited 2023 January 13].

Anexo A

Figuras

A.1. Diagramas

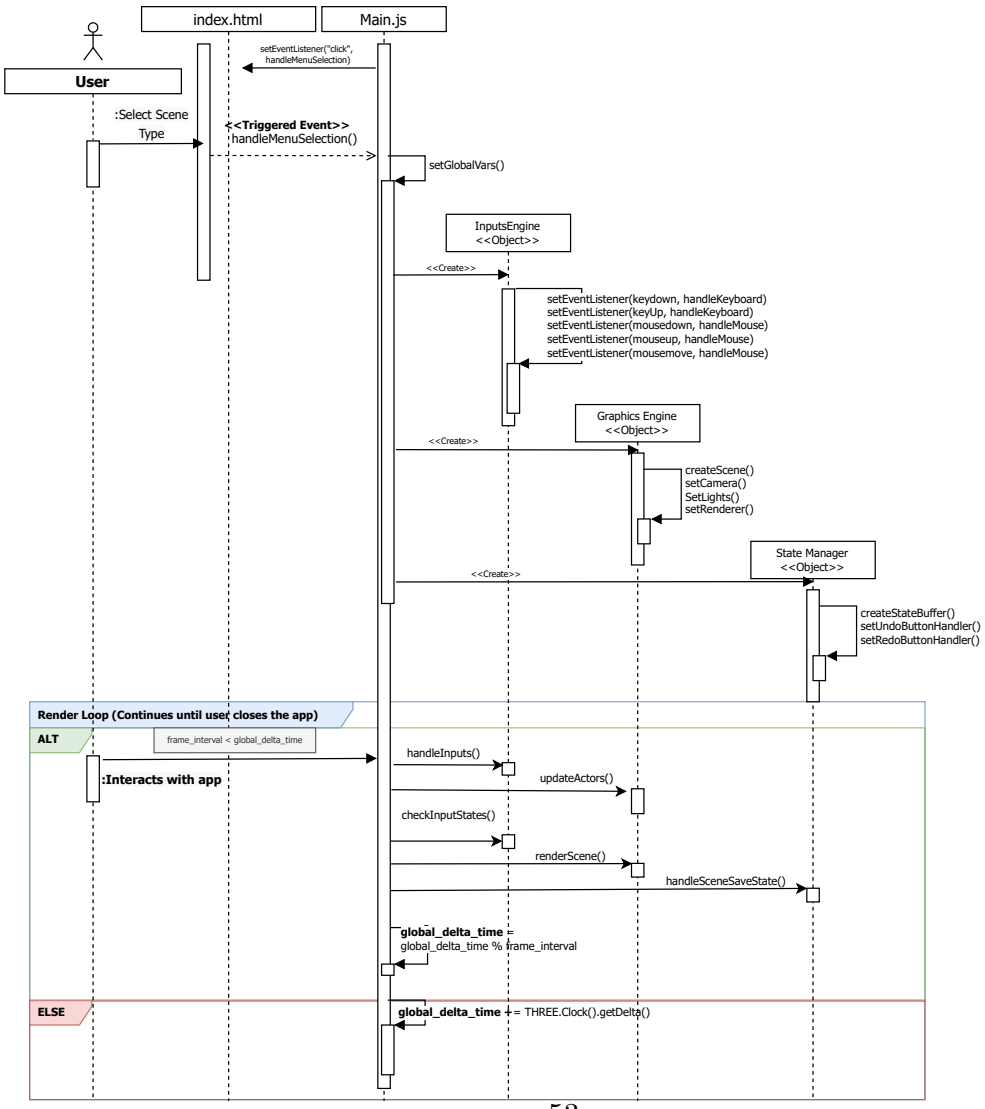

Figura A.1: Estructura Básica del Proyecto

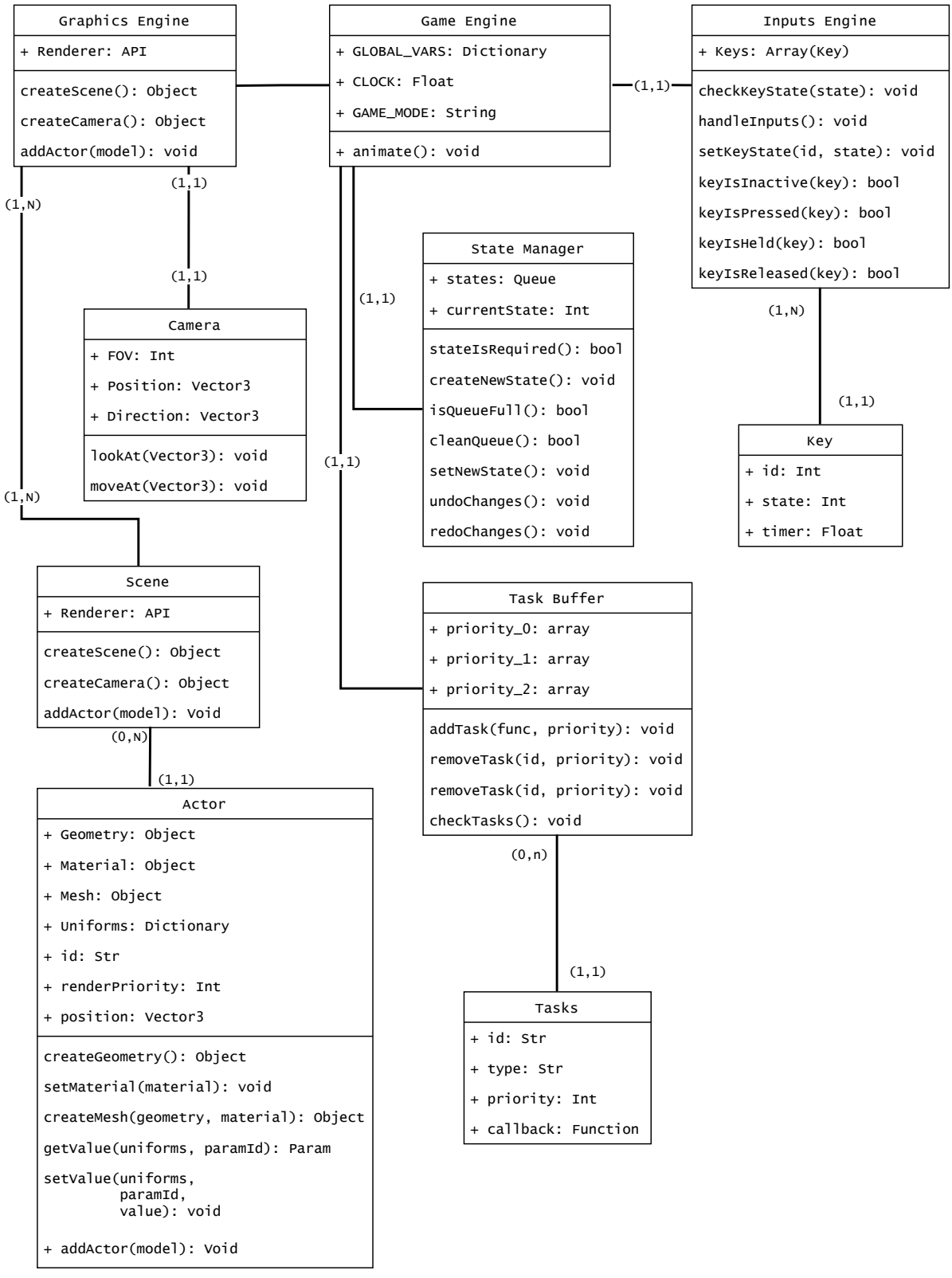

Figura A.2: Modelo entidad-relación (tipo pertenencia) de la arquitectura de la aplicación.

A.2. Imágenes

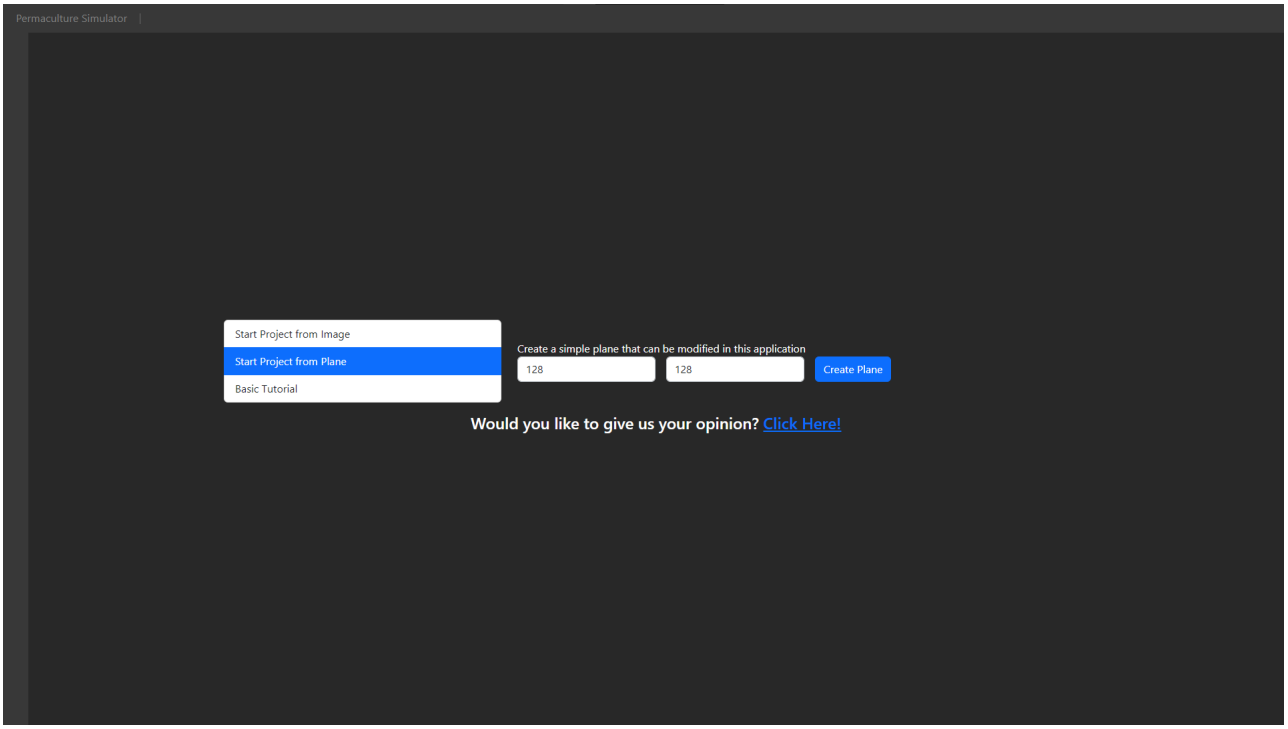

Figura A.3: Vista Principal de la aplicación

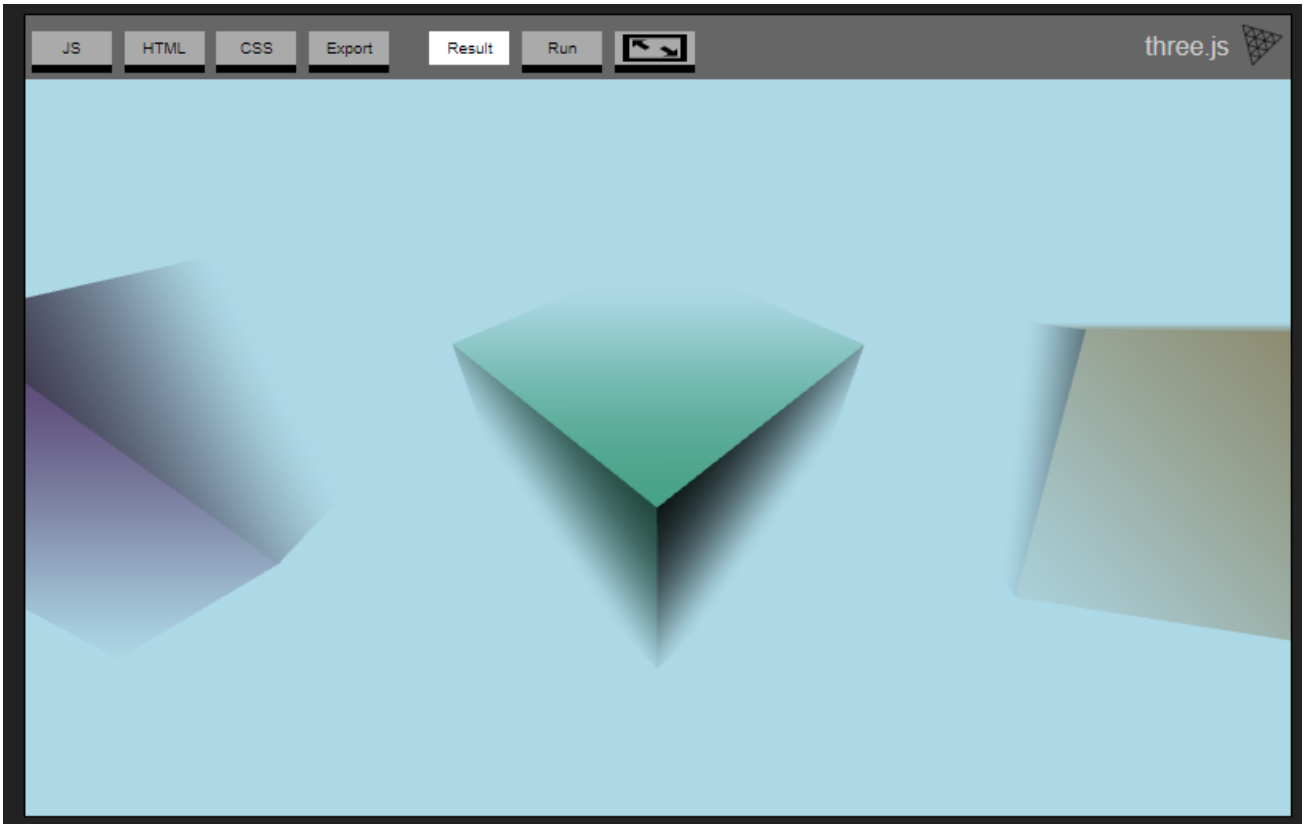

Figura A.4: Aplicación de Fog en una escena

Anexo B

Código

Listing B.1 Fragmento de Index.html. Aquí se aprecia como se segmenta la vista principal de la aplicación. Usamos el ID de cada <**div**> para poder inyectar lo que queramos usando JavaScript

```
<html lang="en">
    <head>
        (.)
    </head>
    <body>
        (.)
        <!- Top Menu ->
        <div class="top-options" id="options-navbar">
            <div class="" id="option-list">

            </div>
        </div>
        <div class="horizontal-container fontMain">
            <div class="d-inline-flex flex-column toolbar align-items-
                center" id="toolbar">
                (.)
            </div>
            <div class="container-fluid game-container" id="game-
                container">
                (.)
            </div>
        </div>
    </body>
</html>
```

Listing B.2 Función aplicada para la creación de los íconos de la barra de herramientas

```
addIconsToToolbar() {
    let iconIds = Object.keys(this.icons);
    let toolbar = document.getElementById('toolbar');
    toolbar.innerHTML = '';
    let toolbarHtmlCode = '';
    iconIds.forEach((iconId) => {
      const iconData = this.icons[iconId];
      let iconSrc = `./resources/icons/${iconId}.png`;
      if (window.VARS.debugging) {
        iconSrc = `../resources/icons/${iconId}.png`;
      }
      const img = `
                <img class="toolbar-icon"
                    src="${iconSrc}"
                    data-bs-toggle="tooltip"
                    data-bs-placement="right"
                    data-bs-title="${iconData.label}"
                    id="${iconId}"
                    name="${iconId}"
                />`;
      toolbarHtmlCode += img;
    });
    toolbar.innerHTML = toolbarHtmlCode;
  }
```

Listing B.3 Función de gestora de estados de botones

```
initializeSelectedTool(toolName) {
    /* Modify HTML Class to visualize icon as active and deactivate the
       previous selected tool */
    const toolIcon = document.getElementById(toolName);
    toolIcon.className = 'toolbar-icon-active';

    const callbacks = this.icons[toolName].callbacks;

    Object.keys(callbacks).forEach(callbackType => {
      this.activeToolCallbacks[callbackType] = callbacks[callbackType];
    });

    this.activeToolId = toolName;
    this.activeToolLabel = this.icons[toolName].label;
    this.activeToolControls = this.icons[toolName].controls;
    this.activeToolCallbacks.onToolSelection();
  }
```

Listing B.4 Función de gestora de estados de botones

```
checkInputBuffer() {
        const activeKeys = [];
        const releasedKeys = [];
        const inputBuffer = this.inputBuffer;

        inputBuffer.forEach(key => {
            if (this.keysAreReleased(key)) {
                releasedKeys.push(key);
            }

            if (this.keysAreHeld(key)) {
                this.heldTimers[key] += VARS.FRAME_INTERVAL;
                activeKeys.push(key);
            }

            if (this.keysArePressed(key)) {
                this.setKeyAsHeld(key)
                activeKeys.push(key);
            }
        });

        this.inputBuffer = activeKeys;
        this.releasedKeys = releasedKeys;
    }
```

Listing B.5 Declaración las herramientas

```
export const toolbarData = {
  hill: {
    name: 'HILL',
    type: 'TERRAIN',
    label: 'Create Hill',
    callbacks: {
      onLeftClick: () => {
        modifyVerticesHeightInRange({ modificationTool: 'hill' });
      },
      onMouseMove: () => {
        VARS.ACTORS.terrain.cleanVertexSelection();
        getCursorPosOnTerrain();
        getVerticesInCursorRange();
      },
      onToolSelection: () => {
        VARS.CURSOR_HANDLER.handleCursorSelection("terrain");
        toolWithCursorOnSelection({ toolName: 'hill' })
      },
      onKeyDown: () => dummy()
```

```
    },
    controls: [
      { KEY: `${mouseIcon}LEFT`, MOUSE: true, CTRL: false, SHIFT: false,
          ALT: false, DESC: 'Rise selected area' }
    ],
    triggerKeys: ["LEFT_CLICK"],
  },
  addTree: {
    name: 'ADD_TREE',
    type: 'TREE',
    label: 'Select a Vertex/Grid to add a Tree',
    callbacks: {
      onLeftClick: () => addTreeToTerrain(),
      onRightClick: () => removeTreeFromScene(),
      onMouseMove: () => getCursorPosOnTerrain(),
      onToolSelection: () => {
        VARS.CURSOR_HANDLER.handleCursorSelection("trees");
        pointSelectorOnSelection()
      }
      ,
      onKeyDown: () => dummy(),
      onToolDeselection: () => pointSelectorOnDeselection(),
    },
    controls: [
      { KEY: `${mouseIcon}LEFT`, MOUSE: true, CTRL: false, SHIFT: false,
          ALT: false, DESC: 'Add Tree' },
      { KEY: `${mouseIcon}RIGHT`, CTRL: false, SHIFT: false, ALT: false,
          DESC: 'Remove Tree' },
    ],
    triggerKeys: ["LEFT_CLICK", "RIGHT_CLICK"],
  }
  ...
}
```

Listing B.6 Declaración la clase padre Component.js

```
export class Component {
  constructor({ name = '', componentID, parent = document.getElementById
     ('temporal-container') }) {
    /*
      el -> Main HTML Component to Render
      Flow: generateComponent() -> validate() -> mount() ->
         setEventListeners()
    */
    this.id = componentID;
    this.name = name;
    this.callbacks = {};
    this.n_children = 0;
    this.parent = parent;
    this.children = [];
    this.el = null;
  }

  validate() {
    let msg = '';
    let status = true;

    if (!this.id) {
      msg += 'An ID must be defined\n';
      status = false;
    }

    if (this.el === null) {
      msg += 'An HTML element must be defined\n';
      status = false;
    }

    if (!status) {
      alert(msg);
    }

    return status;
  }

  generateComponent() {
    this.el = el('div'); // Just a place holder
  }

  render() {
    this.generateComponent();
    this.validate();
    mount(this.parent, this.el);
    this.setEventListeners();
  }
```

```
48    update() {
49      return;
50    }

52    addChildren({ children, props }) {
53      this.el.appendChild(children.el);
54      this.n_childrens += 1;
55    }

57    setParent(parent) {
58      mount(parent, this.el);
59    }

61    setEventListeners() {
62      const types = Object.keys(this.callbacks);
63      console.log(this.callbacks);
64      types.forEach((type) => {
65        const callback = this.callbacks[type]
66        this.el.addEventListener(type, (event) => callback(event));
67      });
68    }

70    setCallbacks(callbacks) {
71      this.callbacks = callbacks;
72      this.setEventListeners();
73    }

75    addCallback(type, callback) {
76      this.callbacks[type] = callback;
77      this.setEventListeners();
78    }

80    setParam({ paramName, value }) {
81      this[paramName] = value;
82    }
83  }
```