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

POSICIONAMIENTO PROCEDIMENTAL BASADO EN GPU

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

SERGIO ANDRÉS MORALES DELGADO

PROFESOR GUÍA:
DANIEL CALDERÓN SAAVEDRA

MIEMBROS DE LA COMISIÓN:
FRANCISCO GUTIÉRREZ FIGUEROA
ELÍAS ZELADA BAEZA

SANTIAGO DE CHILE
2023

POSICIONAMIENTO PROCEDIMENTAL BASADO EN GPU

El presente informe describe el desarrollo e implementación de un sistema de posicionamiento procedimental capaz de aprovechar las capacidades de cómputo paralelo de las GPU modernas. Es decir, un método para generar la posición de objetos como árboles, arbustos y rocas sobre un terreno virtual de forma mayormente automatizada. Este proyecto se inspira en diversos avances técnicos en lo que respecta a la generación y representación de mundos virtuales de gran extensión, y busca aportar una herramienta para hacer más accesible la creación de este tipo de mundos a desarrolladores más allá de los estudios de videojuegos con más recursos.

Los objetivos de diseño centrales fueron alcanzar un rendimiento apto para una aplicación gráfica en tiempo real, y el determinismo necesario para que al ejecutar el algoritmo repetidamente sobre el mismo sector del mundo se obtengan siempre los mismos resultados. Esto significa que no es necesario almacenar la información de cada elemento individualmente, sino que puede ser recreada rápidamente cada vez que se necesita. En su conjunto, esto permite que el sistema de posicionamiento reemplace las soluciones usuales al problema, que generalmente utilizan grandes cantidades de datos precalculados. La implementación de este sistema se encuentra disponible en forma de una biblioteca de código abierto, desarrollada utilizando C++ y OpenGL.

Tabla de Contenido

Índice de Tablas

Índice de Ilustraciones

Capítulo 1

Introducción

Durante la década pasada y hasta el presente, el mundo de los videojuegos ha experimentado un auge sostenido del "mundo abierto". Este es el término con el que se conoce a aquellos títulos que presentan como escenario locaciones geográficamente vastas, usualmente diseñadas de forma menos lineal de lo que es común en el medio. Exponentes notables de esta escuela de diseño son las sagas de Assassin's Creed, publicada por Ubisoft; Grand Theft Auto, de Rockstar Games; y The Elder Scrolls, de Bethesda.

Si bien han existido juegos con un diseño abierto al menos desde fines de la década del 90, es innegable que en los últimos años han representado un porcentaje cada vez más grande de las producciones de la industria. Estudios que tradicionalmente han estado ligados a un diseño más lineal han encontrado éxito con títulos como The Witcher 3 y Cyberpunk 2077, de CD Projekt RED; Horizon: Zero Dawn y Horizon: Forbidden West, de Guerrilla Games; y Elden Ring, de From Software. Todos ellos destacan por su gran volumen de ventas y, en la mayoría de los casos, también por su recepción crítica.

Esta tendencia no es fortuita. El desarrollo de un videojuego de este estilo tiene requerimientos considerablemente distintos a los usuales, tanto desde el punto de vista técnico como el artístico. Las técnicas de optimización usuales son inútiles o inefectivas cuando se trata de dibujar vastas extensiones de terreno cubiertas de numerosos objetos, actualizar periódicamente gran cantidad de actores, o leer desde el almacenamiento primario los datos requeridos. Así mismo, la aproximación al diseño artístico y de jugabilidad debe adaptarse para poder crear niveles extensos, poblados con elementos y actividades en cantidad y calidad suficientes.

Tradicionalmente, sólo estudios de la envergadura de Bethesda y Rockstar han tenido la

experiencia y los recursos para llevar a cabo proyectos de mundo abierto, e incluso para estos estudios, habiéndose dedicado casi exclusivamente al género, el desarrollo resulta una tarea sumamente compleja. Particularmente notable es el caso de los juegos de Bethesda, que son casi tan conocidos por su gran tamaño y ambición como por estar plagados de *bugs* y errores.

Los avances tecnológicos de la última década, tanto en hardware como en software, han contribuido significativamente a hacer del mundo abierto un problema más abordable, abriendo las puertas para que una mayor variedad de desarrolladores prueben suerte en el mercado. Dentro de estas mejoras tecnológicas se encuentran, por ejemplo, la masificación de los SSD y de las CPU con más dos núcleos, así como la incorporación del cómputo en GPU a las API gráficas más comunes.

Si bien estos avances hacen posibles mundos de mayor tamaño, lo que distingue a un mundo abierto de calidad no es la extensión de este, sino la densidad y variedad de actividades, lugares y personajes que lo habitan. Por esta razón es que una práctica común en el desarrollo de este tipo de software es tratar de minimizar el tiempo utilizado en llenar el mundo de elementos de alta frecuencia y bajo impacto, como árboles y rocas, para poder invertir más tiempo en los elementos centrales de la experiencia, como personajes y puntos de interés.

Una forma de lograr esto es utilizar técnicas de automatización para reducir el trabajo humano requerido. Por ejemplo, se pueden generar algorítmicamente las posiciones de los árboles en un bosque para luego almacenarlas junto a los demás recursos de la escena, o escoger la apariencia de los transeúntes de una ciudad en tiempo de ejecución. Así mismo, es posible automatizar más que sólo los detalles: en el juego de mesa *Los colonos de Catán* la posición de los recursos en el tablero se determina aleatoriamente al comenzar una nueva partida, mientras que en el videojuego Minecraft la geografía del mundo está completamente determinada por un algoritmo aleatorizado.

Colectivamente, esta clase de técnicas forman parte de lo que se conoce como "generación procedimental de contenido" (de ahora en adelante abreviada como GPC). De acuerdo a Togelius *et al*, "la GPC es la creación algorítmica de contenido de juego con entrada de usuario limitada o indirecta" [1]. Cabe hacer notar que en esta definición el usuario en cuestión se refiere al diseñador del juego, no al jugador. Así mismo, como se menciona en [2], "contenido" aquí se refiere a "la mayoría de lo que está contenido en un juego": elementos básicos como texturas, mallas poligonales y pistas de audio; pero también elementos más complejos como escenarios, personajes, objetos o historias.

En términos generales, el contenido de un juego son los datos que el software utiliza como entrada, excluyendo aquellos provistos directamente por el jugador para controlar el juego.

Un ejemplo ilustrativo de la distinción entre el contenido de un juego y sus sistemas se puede encontrar en Doom, el videojuego de 1993. La mayoría de las versiones de este juego constan de dos componentes: un archivo ejecutable (e.g. "doom.exe") y uno o más archivos WAD. El ejecutable corresponde a lo que comúnmente se denomina el motor del juego, como el sistema de rendering. Los archivos WAD, por contraparte, corresponden a lo que previamente se definió como contenido: imágenes, pistas de audio, mapas, enemigos, etc.

Del conjunto total del contenido de un juego, el presente trabajo de memoria se enfoca en la generación procedimental del escenario; esto es, del espacio virtual en que se ambienta un videojuego. En particular, se busca proporcionar una solución al problema anteriormente mencionado del posicionamiento de elementos de alta frecuencia pero bajo impacto individual sobre amplias extensiones de terreno.

Como se mencionó anteriormente, entre las funcionalidades de hardware y software que han adquirido relevancia como herramientas para el desarrollo de videojuegos y experiencias interactivas se encuentra el cómputo en GPU. Este concepto, a veces denominado GPGPU[1], se refiere a la utilización de la GPU para tareas de cómputo que no están directamente relacionadas con el procesamiento de gráficos. Por ejemplo, es posible realizar operaciones sobre matrices de dimensiones tan grandes que en CPU serían impracticables. Estas capacidades juegan un rol importante en campos como el *Deep Learning*, al punto que Nvidia, una empresa que tiene sus orígenes en el desarrollo de GPUs para videojuegos, ofrece servicios y productos orientados al desarrollo de inteligencia artificial y la ciencia de datos[3].

En vista de que herramientas para el cómputo en GPU han estado disponible en el ámbito del desarrollo de videojuegos al menos desde la octava generación de consolas (e.g. PlayStation 4 de Sony, y XBox One de Microsoft), y de que todas las APIs gráficas modernas (i.e. Vulkan y Direct3D) tienen capacidades de cómputo general, es natural considerar la posibilidad de aprovechar estas capacidades para solucionar el problema de posicionamiento anteriormente descrito. Así, el objetivo de este trabajo de título es desarrollar una solución de este tipo, un sistema de posicionamiento procedimental basado en GPU. En particular, las tareas centrales para cumplir este objetivo fueron, por un lado, la investigación y el desarrollo de un algoritmo de posicionamiento en paralelo y, por el otro, la implementación de dicho algoritmo en una biblioteca de las características requeridas para su utilización en un videojuego u otra aplicación en tiempo real.

En los próximos capítulos se expondrá, en primer lugar, los precedentes que existen para un sistema de este tipo y que han servido de inspiración para este trabajo. Luego, se explicarán las características del algoritmo utilizado, así como la arquitectura y el diseño de la biblioteca,

[1] Sigla en inglés que significa *General Purpose [compute on a] Graphics Processing Unit.*

para finalmente concluir con una demostración del resultado obtenido.

Capítulo 2

Estado del arte

2.1. Tecnologías de hardware y software

Como preámbulo a los capítulos posteriores, cabe hacer referencia al contexto de hardware y software en el que se ejecutan los videojuegos modernos. Por el lado del hardware, prácticamente todos los sistemas computacionales modernos poseen dos componentes de cómputo principales: la CPU y la GPU. La CPU (*central processing unit*) es un componente de propósito general que se ocupa la amplia mayoría de las tareas del sistema. Si bien en los últimos años ha habido una tendencia hacia un mayor número de núcleos en las CPUs, el funcionamiento general de esta componente se mantiene mayormente inalterado y es bien conocido, por lo que no hace falta entrar en mayor detalle.

La GPU (*graphics processing unit*), por contraparte, es una componente para tareas especializadas. Como dice su nombre, está enfocada en el procesamiento de gráficos. Está diseñada para aprovechar una característica común de las tareas de computación gráfica: el paralelismo de datos. En pocas palabras, estos problemas pueden ser resueltos repitiendo una misma serie de operaciones sobre distintos datos. Por esta razón, la GPU posee una arquitectura SIMD, *Single Instruction Multiple Data*, que aplica la misma instrucción sobre varios datos de forma paralela. La rasterización, el proceso mediante el cual la información geométrica se procesa para construir una imagen compuesta por píxeles, es fácilmente paralelizable de este modo. En la mayoría de los casos el color de cada píxel puede ser calculado sin necesidad de conocer el color de ningún otro píxel de la imagen y, por tanto, todos ellos pueden ser calculados simultáneamente, siguiendo el mismo procedimiento cada vez. De manera similar, otras operaciones sobre la información geométrica también se benefician de la arquitectu-

ra SIMD. Por ejemplo, el aplicar una traslación, rotación o escalado a una malla poligonal puede lograrse multiplicando las coordenadas de cada uno de sus vértices por una matriz de rotación de forma independiente.

Estas operaciones son ampliamente utilizadas y, por tanto, las GPU contienen hardware especializado, diseñado para acelerar su cómputo. Así mismo, el hardware gráfico moderno cuenta con lo que se conoce como un *pipeline* gráfico programable. Esto quiere decir, en términos simples, que el comportamiento de algunas de las etapas del proceso de rasterización puede ser ajustado mediante la ejecución de programas especiales conocidos como *shaders* ("sombreadores"[2]), los cuales se ejecutan en la GPU. Tradicionalmente, el shader que se ejecuta en la fase de preprocesamiento de geometría (e.g. aplicación de traslación, rotación y escala) recibe el nombre de *vertex shader* ("sombreador de vértices"), mientras que aquel que opera al calcular el color de los píxeles individuales se denomina *pixel shader* o *fragment shader*.

Como se mencionó con anterioridad, en las últimas década se ha hecho cada vez más común lo que se conoce como GPGPU[4], el uso de la GPU para tareas de cómputo general no necesariamente ligadas al ámbito de la computación gráfica. En el ámbito de los videojuegos, esto se ha visto reflejado en que las distintas APIs gráficas implementan lo que se conoce como *compute shaders*. Estos son programas que, al igual que los *shaders* tradicionales, se ejecutan en la GPU. Sin embargo, no se ejecutan en el contexto del pipeline gráfico tradicional, donde se realiza el procesamiento de vértices y el cálculo de fragmentos, sino que pueden ser utilizados en cualquier momento para realizar operaciones arbitrarias sobre datos arbitrarios. Es decir, son shaders de propósito general.

Finalmente, cabe mencionar que la anteriormente mencionada API gráfica es la capa de software de bajo nivel que permite al programador de una aplicación hacer uso de la GPU. En computadores tradicionales las más importantes son Direct3D, Vulkan y OpenGL. Si bien algunas de éstas están disponibles en ciertas consolas de videojuegos, históricamente lo usual ha sido que cada consola tenga su propia API. En la actualidad, todas las APIs implementan algún tipo de *compute shader*, por lo que los videojuegos modernos tienden a hacer uso regularmente de la GPU para tareas no directamente gráficas.

[2] Este nombre deriva, probablemente, de que su función principal es calcular el sombreado y la iluminación de los distintos objetos en una escena tridimensional.

2.2. Métodos generales de Generación Procedimental de Contenido

Como se mencionó anteriormente, la generación procedimental puede ser aplicada a una amplia variedad de contenidos. El estudio de Freiknecht y Effelsberg [5] sobre el tema establece que los elementos que más frecuentemente han sido sujetos a técnicas de generación procedimental son los siguientes:

- Paisajes naturales, donde destaca la generación de vegetación, agua y relieve del terreno. Las teorías y herramientas de software correspondientes han alcanzado un alto nivel de madurez.
- Autopistas, carreteras y calles.
- Construcciones.
- Animales, incluyendo humanos, y otras entidades con cierta autonomía.
- Historias y narrativas.

Cabe hacer notar que, a excepción de los paisajes naturales, todas estas categorías se conciernen con la creación de instancias individuales de los objetos correspondientes, más que su posicionamiento sobre el terreno. Es decir, generan la estructura, materialidad, distribución de habitaciones y mobiliario de un edifico, no su ubicación dentro de una ciudad. Por esta razón, la mayoría de las técnicas descritas en esta sección se enmarcan en la categoría de generación de paisajes naturales.

2.2.1. Terreno

En primer lugar, es necesario tener presente las características del terreno sobre el que han de posicionarse los objetos. En general, es sumamente probable que, en un proyecto donde la vegetación u otros objetos son posicionados procedimentalmente, la superficie sobre la que se realiza el posicionamiento sea también el producto de técnicas de generación procedimental. Por esta razón es razonable estudiar este campo aunque, en estricto rigor, se encuentre fuera del alcance de este trabajo.

En lo que respecta al relieve, una técnica comúnmente utilizada es la del mapa de altura (o *heightmap*, en inglés). Esencialmente, es lo mismo que un mapa topográfico tradicional: una proyección bidimensional del terreno, donde el color de cada punto de la imagen es una función de la altura de la superficie en el punto correspondiente del terreno. Lo más común es que la paleta de colores sea una escala de grises simple y la función, una interpolación lineal entre el punto más bajo y el más alto. Así, el color negro corresponde al punto más bajo del terreno y el color blanco, al más alto. A partir de esta información se construye luego una malla poligonal para *rendering* o una estructura para el cálculo de colisiones. Así, si la imagen que es utilizada como mapa de altura es generada procedimentalmente, entonces se puede decir que el relieve del terreno fue generado procedimentalmente. Por su parte, una forma común de generar mapas de altura es utilizando funciones de ruido (e.g. de Perlin[6]) o fractales.

Para determinar la textura del terreno, esto es, el color y otras propiedades ópticas, se pueden aplicar las técnicas tradicionales de texturización y crear manualmente las imágenes a proyectar sobre la superficie. Cuando se trabaja con superficies extensas esta aproximación puede ser muy costosa, tanto en tiempo humano como en espacio de almacenamiento. La solución más simple es crear una o más imágenes de tamaño reducido y repetirlas hasta cubrir la superficie completa. Sin embargo, es fácil darse cuenta de que esto conlleva la aparición de un patrón visible sobre la superficie, lo que muchas veces es indeseable desde el punto de vista artístico.

Otra solución aplicable a grandes extensiones de terreno es la utilización de un mapa de texturas. Esto es análogo a un mapa de altura, pero en este caso el color representa la "intensidad" de la textura en un punto de la superficie. Esto permite mezclar múltiples texturas, de modo que la textura final de un punto de la superficie es la interpolación de las texturas individuales con coeficientes dados por su intensidad en el mapa de texturas. Por ejemplo, se puede utilizar una imagen de tres canales (rojo, verde y azul) para mezclar tres texturas distintas (un ejemplo de esto se encuentra en [5], fig. 10). Esta técnica permite un buen grado de control artístico, puede generar resultados visualmente convincentes y ocultar la repetición de las imágenes, pero por sí misma no constituye un método completamente procedimental.

Finalmente, también es posible determinar la textura a través de un análisis del relieve del terreno, esto es, calcularla como una función de la altura y la pendiente. Por ejemplo, permite texturizar los terrenos más elevados como nieve, a los más bajos como suelo oceánico, las alturas intermedias como tierra, y las zonas con alta pendiente como roca (i.e. acantilados). Este método es completamente procedimental, pues no requiere intervención directa para determinar qué texturas corresponden a qué áreas (a diferencia de un mapa de texturas hecho manualmente).

2.2.2. Distribución de vegetación

En [5] se describen técnicas similares a las utilizadas para el modelado de terreno. Por ejemplo, utilizar un mapa en escala de grises donde la tonalidad determina la densidad de la vegetación sobre cada punto del terreno. De modo similar al texturizado del terreno, se puede crear un mapa a color y asociar cada color a un tipo de vegetación. Se menciona también la posibilidad de posicionar la vegetación en función del mapa de altura del terreno, analizando elevaciones y pendientes. Cabe mencionar que, estos métodos no dependen del carácter biológico de la vegetación y, por tanto, pueden ser generalizados para el posicionamiento de objetos arbitrarios.

Por contraparte, se menciona tanto en [7] como en [5] la existencia de métodos de posicionamiento procedural que toman en consideración ciertos aspectos biológicos. Un ejemplo notable es el modelo denominado *field of neighborhood* [8], que asigna a cada planta una zona de influencia circular sobre el resto de la vegetación. El tamaño de esta zona puede depender de factores como la fertilidad del suelo y la humedad del ambiente, así como el tamaño y la especie de la planta. Con ello se busca aproximar el efecto que la disponibilidad de recursos y la competencia por estos tiene sobre la distribución de la vegetación sobre el terreno. El trabajo de Deussen *et al* [9] presenta la arquitectura de un sistema de modelamiento procedimental tanto del terreno como de la vegetación.

Una aproximación más simple es posicionar instancias aleatoriamente siguiendo una distribución de Poisson. Para evitar que las plantas colisionen entre sí, se define la distribución "de disco" de Poisson imponiendo una distancia mínima entre instancias individuales; el disco en cuestión se refiere al área circular en torno a cada elemento dentro de la cual no puede posicionarse otra instancia.

2.3. Métodos de posicionamiento procedimental

En esta sección se describen técnicas de GPC enfocadas específicamente en el posicionamiento de elementos sobre el terreno. En primer lugar, se hace una revisión de la literatura académica en torno al tema. Luego, se exponen algunas técnicas y productos actualmente en uso en la industria de los videojuegos.

2.3.1. Literatura académica

El trabajo de Torres *et al* [10] presenta un esquema que integra la distribución procedimental de vegetación en tiempo de ejecución y el renderizado de esta en tiempo real. La arquitectura hace uso intensivo del paralelismo en GPU para lograr el rendimiento necesario para una aplicación en tiempo real. Las entradas principales del sistema son mapas de altura y agua para el terreno, así como una serie de "parámetros de adaptabilidad" para cada tipo de planta.

El posicionamiento se realiza calculando posiciones de acuerdo a una distribución de disco de Poisson; luego, para cada posición generada, se evalúan las características del terreno (altura, pendiente, agua y humedad) en conjunto con el mapa de densidad para determinar si una planta será colocada y, de ser así, cuál será su tipo. Un aspecto interesante de esta implementación es que utiliza distintas capas para separar plantas de distinto tamaño, de modo que las plantas más grandes pueden influenciar la distribución de la vegetación en los niveles inferiores. Finalmente, se especifican también una serie de optimizaciones de la arquitectura con el fin de obtener el mejor rendimiento posible en escenas muy extensas (del orden decenas de kilómetros cuadrados y más).

Enríquez y Akleman [11] han diseñado un sistema de posicionamiento procedimental para la creación de mapas pictóricos tridimensionales. En pocas palabras, un mapa pictórico es un mapa que, más que ser fiel a la geografía, busca caracterizar el paisaje y la gente de una región; por esto se incluyen representaciones gráficas de aspectos de la cultura local, así como de la flora y la fauna. La herramienta está diseñada para posicionar elementos de diversa índole sobre una escena de tamaño limitado. Un aspecto interesante de esta herramienta es la inclusión de un método simple para orientar las construcciones de modo que apunten al camino más cercano, utilizando como descripción de este camino nada más que una imagen en escala de grises.

Los datos de entrada de este sistema de posicionamiento son mapas de densidad en conjunto con una serie de definiciones de ecotopos; los datos de salida son conjuntos de posiciones para los distintos tipos de objetos. Los ecotopos son una estructura de datos tomada del sistema de posicionamiento utilizado en el videojuego Horizon: Zero Dawn [12] y deben su nombre al concepto biológico homónimo. En términos simples, los ecótopos especifican el tipo y distribución de la vegetación para una sección del terreno, es decir, especifican qué mapas de densidad utilizar y cómo interpretarlos.

2.3.2. Industria

En primer lugar, se presenta una breve revisión de las herramientas disponibles en algunos motores de videojuegos de uso común. En pocas palabras, un motor de videojuegos es un *framework* orientado específicamente al desarrollo de videojuegos. Por lo general, abstraen los aspectos de más bajo nivel (gráficos, físicas, sonido, *networking*, etc) y permiten manejarlos de manera más eficiente e intuitiva. En particular, se revisarán tres de los motores más utilizados [13]: Unreal Engine, un motor propietario desarrollado por Epic Games y disponible bajo licencia comercial; Unity Engine, desarrollado por Unity Technologies y distribuido bajo una licencia comercial un poco más permisiva que la del anterior; y Godot Engine, un proyecto de código abierto con licencia MIT. Los dos primeros son lejos los más utilizados en la industria, mientras que el tercero es el más popular entre los motores de código abierto actualmente disponibles.

En Unreal Engine se incluye la "Herramienta de Follaje Procedimental" (Procedural Foliage Tool) [14]. Ésta opera en conjunto con el sistema de terreno del motor, definiendo "volúmenes de follaje" dentro de los cuales se posicionan aleatoriamente objetos sobre la superficie del terreno, de acuerdo a los parámetros establecidos por el usuario. El principal inconveniente de esta implementación es que es parte de un motor propietario. Esto significa que la solución está restringida a los usuarios de este software y no puede fácilmente ser adaptada a otros motores. Al igual que otros métodos, está pensado principalmente para la vegetación y podría no ser adaptable a otras categorías de objetos.

La versión 5.2 de Unreal Engine "ofrece un primer vistazo a un marco de generación procedimental de contenido" orientado a hacer que "el proceso de creación de mundos titánicos sea más rápido y eficaz" [15], lo que se alinea de forma cercana con los objetivos previamente establecidos. Sin embargo, esta versión fue publicada de forma posterior al desarrollo de este trabajo y, por tanto, no formó parte de la investigación inicial.

En el motor Unity, la herramienta que más se acerca a lo buscado es *Landscaper*, un *plug-in* disponible comercialmente en la Unity Asset Store. Esta extensión está pensada específicamente para la creación de bosques y enfocada en obtener una distribución de las vegetación lo más verosímil posible. Implementa varias técnicas de posicionamiento procedimental de vegetación, incluyendo un modelo estilo field-of-neighborhood y la capacidad de emular la evolución del ecosistema generación por generación. Dicho esto, al igual que Unreal, Unity es software propietario y, por tanto, esta herramienta posee restricciones similares.

Finalmente, para Godot no se encontraron herramientas o extensiones similares. Sin embargo,

se puede mencionar la existencia de herramientas de generación procedimental de geometría, tanto nativas [16] como externas [17]. Existe también material [18] que trata el tema de la generación procedimental de mundos, aunque más enfocado en la construcción de niveles bidimensionales, usualmente organizados en una grilla y de tamaño reducido. También está disponible la extensión *Procedural World Map Generator* [19], que permite generar mapas (en el sentido geográfico) prácticamente infinitos de forma procedimental. Esto último es complementario a los objetivos de este trabajo, al encargarse de una fase previa de la generación de mundos.

Por otro lado, también existen precedentes importantes en videojuegos publicados. Como se mencionó anteriormente, el videojuego Horizon: Zero Dawn, desarrollado por Guerrilla Games y estrenado en 2017, utiliza un sistema de posicionamiento procedimental. Como explica Van Muijden [12], un integrante del equipo de Guerrilla Games, este sistema fue diseñado inicialmente para el posicionamiento de vegetación y posteriormente fue expandido para poder posicionar objetos arbitrarios. Al igual que muchos otros métodos, utiliza mapas de densidad que determinan la cantidad de elementos a posicionar en un área determinada. En su forma más básica, el algoritmo tiene tres pasos: primero, se generan posiciones aleatorias hasta llenar el espacio; luego, se le asigna a cada posición un valor en el rango [0, 1]; finalmente, se compara este valor con el mapa de densidad en la posición correspondiente y se posicionan elementos en todos los lugares donde el valor sea superior a la densidad.

Dicho esto, la implementación de Guerrilla Games tiene varios aspectos interesantes. En primer lugar, es un algoritmo altamente paralelizable, por lo que al ejecutarlo en GPU se hace posible realizar el posicionamiento en tiempo de ejecución. Esto permite ahorrar espacio de almacenamiento al sólo ser necesario almacenar los mapas de densidad, calculando el posicionamiento exacto de los objetos a medida que el jugador se mueve por el mapa. Además, permite a los diseñadores del juego ver los cambios que realizan en el escenario de forma inmediata, facilitando una rápida iteración. Crucialmente, el algoritmo es determinista, por lo que la distribución que ve el diseñador es la misma que experimenta el jugador.

Otra fortaleza de esta implementación es el sistema de ecotopos. Estos determinan el mapa de densidad y el tipo de objetos que pueden ser posicionados en cada zona. Además de esto, los ecotopos pueden ser organizados en una estructura jerárquica, de modo que el mapa de densidad utilizado para los objetos de un ecotopo determinado es el resultado de una operación arbitraria entre el mapa de dicho ecotopo y el mapa del ecotopo padre. Esto evita cierta duplicación de información entre distintos mapas de densidad, como la posición de los caminos (donde no deben posicionarse plantas).

Entre las distintas aproximaciones al problema del posicionamiento, la desarrollada por Gue-

rrilla Games destaca por su versatilidad y rapidez. Además, el algoritmo se ejecuta en GPU, haciendo posible posicionar objetos en tiempo de ejecución con un bajo impacto en el rendimiento de un videojuego. Por estas razones es que ha sido utilizada como inspiración para el presente trabajo.

Capítulo 3

Diseño y metodología

El producto de este trabajo es una biblioteca que proporciona un sistema de posicionamiento procedimental. Este capítulo describe el diseño de ésta y se encuentra dividido en tres secciones. La primera expone los objetivos de diseño para de la solución. La segunda presenta una descripción general del algoritmo de posicionamiento y las etapas que lo componen. Finalmente, la tercera sección se concierne con el diseño de la biblioteca misma, incluyendo el modo de uso y las tecnologías en las que se basa.

3.1. Objetivos de diseño

Para el desarrollo de la solución, se establecieron tres principios centrales para guiar las decisiones de diseño:

1. **Rapidez**: debe ser más rápido que los métodos tradicionales de posicionamiento. En particular, debe ser suficientemente rápido como para utilizarse en el contexto de una aplicación gráfica que se ejecuta en tiempo real, como un videojuego.
2. **Determinismo**: el resultado de las operaciones de posicionamiento debe depender únicamente de los parámetros explícitamente especificados. En particular, no debe depender del contexto de hardware y sistema operativo en que se ejecuta, y debe tomar en cuenta el carácter concurrente y potencialmente paralelo de las operaciones en GPU. En conjunto con el punto anterior se hace innecesario precalcular y almacenar la ubicación de

una gran cantidad de objetos, pues pueden estas posiciones pueden ser calculadas en tiempo de ejecución obteniendo el mismo resultado cada vez.

3. **Compatibilidad**: las tecnologías utilizadas deben ser compatibles y razonables de utilizar en el contexto de una aplicación gráfica en tiempo real.

En adición a esto, se identificaron ciertas características comunes a los videojuegos de mundo abierto que el diseño de la solución debe tener en cuenta. Estas son:

- El mundo de juego es extenso. Esto significa que no es razonable almacenar todos los objetos en juego en memoria RAM o VRAM, requiriendo espacio de almacenamiento en el orden de las decenas de gigabytes.
- El mundo de juego es continuo. Es posible recorrer todo el mapa sin toparse con una pantalla de carga. Esto significa que el mundo de juego es cargado y descargado dinámicamente por secciones. Esto se conoce como *streaming*. Ejemplos de esto son el sistema de *chunks* ("pedazos") utilizado en Minecraft y la tecnología de Unreal Engine 5 conocida como *World Partition*.
- Los objetos posicionados en el mundo son heterogéneos, es decir, poseen distintos tamaños y se distribuyen de manera variada sobre el terreno. Por ejemplo, cierto tipo de objeto puede estar presente sólo en una zona específica del mapa, depender de la presencia de elementos como ríos y caminos, o bien, aparecer en función del relieve del terreno.

Para comprobar el cumplimiento de estos objetivos, la biblioteca contiene programas de ejemplo, tests unitarios y pruebas de rendimiento. Los tests unitarios sirven para comprobar la correctitud y el determinismo de la implementación del algoritmo de posicionamiento. Las pruebas de rendimiento miden el tiempo que tarda en ejecutarse el algoritmo y lo comparan contra otras posibles soluciones en CPU para verificar el aumento de rendimiento.

3.2. Algoritmo de posicionamiento

El algoritmo de posicionamiento procedimental está basado en el utilizado en Horizon: Zero Dawn [12]. En su forma más básica, este algoritmo consiste en elegir una posición sobre el terreno y evaluar una función de probabilidad definida sobre la superficie del terreno para

determinar si en el punto elegido corresponde crear un objeto. Sin embargo, lo que distingue a la versión utilizada en Horizon: Zero Down de algoritmos como el lanzamiento de dardos es la capacidad de generar y evaluar cada punto de forma independiente a los demás, facilitando la paralelización del algoritmo.

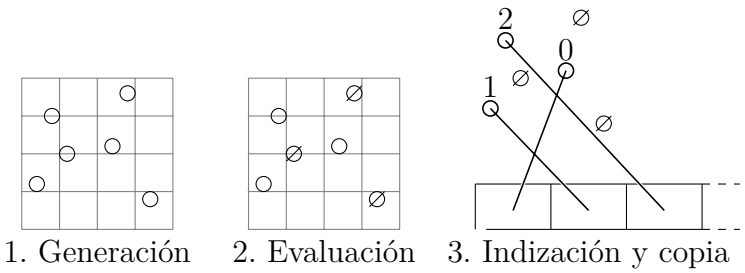


Figura 3.1: Etapas del algoritmo de posicionamiento.

En este trabajo el algoritmo está compuesto por cuatro etapas, de las cuales las últimas dos son opcionales. Las primeras dos etapas se denominan Generación y Evaluación, mientras que las etapas opcionales se conocen como Indización y Copia. La etapa de generación produce un conjunto de puntos sobre la superficie del terreno. La etapa de evaluación aplica la función de probabilidad a cada uno de estos puntos. Finalmente, las etapas de indización y copia generan un arreglo ordenado con los elementos elegidos en la etapa de evaluación.

Código 3.1: Estructura general del algoritmo de posicionamiento.

```
input: heightmap, densitymaps[]
begin
    candidates ← generate(heightmap)
    result ← {}

    for candidate in candidates
        for densitymap in densitymaps
            if evaluate(candidate, densitymap) == true
                result ← result ∪ {(candidate, densitymap)};
                break

    return result
end
```

En la sección siguiente se introducen algunos términos para poder, a continuación, exponer en más detalle las cuatro etapas mencionadas.

3.2.1. Conceptos previos

3.2.1.1. Mapa de altura y mapa de densidad

En el contexto de la computación gráfica, un "mapa" o "mapeo" es un campo vectorial o escalar definido sobre una superficie. Es decir, se define un sistema de coordenadas bidimensionales donde cada par (s, t) identifica un punto único sobre la superficie tridimensional, y en base a este se define el campo vectorial o escalar.

Éste término viene del concepto de "mapeo de texturas" (*texture mapping*), que permite proyectar una imagen sobre una malla poligonal. Esto se logra asignando a cada punto sobre la superficie un par de coordenadas (u, v) en el rango $[0, 1]$, conocidas como coordenadas de textura, las cuales se utilizan para acceder a una imagen bidimensional, lo que se conoce como muestreo. Para esto, la imagen se interpreta como una matriz de colores (píxeles, o más específicamente, texeles) de m filas y n columnas. Al muestrear una imagen usando coordenadas de textura (u, v) se obtiene el color del elemento en la fila $\lfloor vm \rfloor$ y la columna $\lfloor un \rfloor$.

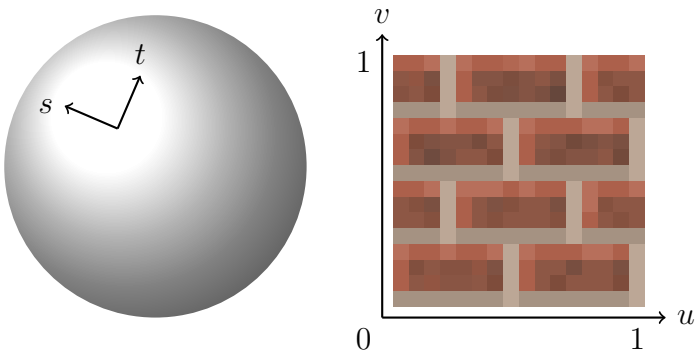


Figura 3.2: Coordenadas de textura (u, v) y coordenadas de superficie (s, t).

Es decir, para una imagen $I \in \mathcal{M}_{m \times n}$, se define la operación `sample` como

$$\texttt{sample}_I(u, v) = I_{\lfloor vm \rfloor, \lfloor un \rfloor}$$

En computación gráfica es común representar los colores como números reales en el rango $[0, 1]$. Para una imagen en el espectro de color RGB, el color de cada píxel se representa como

una tupla $c \in [0,1] \times [0,1] \times [0,1]$, donde cada elemento indica la intensidad de rojo, verde y azul, respectivamente. En el caso de una imagen en escala de grises se tiene, por tanto, un único valor en el rango $[0,1]$. Entonces, el resultado de la operación `sample` puede ser desde un valor escalar, para una imagen en escala de grises, hasta un vector de cuatro dimensiones, para una imagen con tres canales de color y uno de transparencia.

Por otra parte, las coordenadas de textura son, formalmente, una función de las coordenadas de la superficie: $(u(s,t), v(s,t))$. En la práctica, esto se logra asignando a cada vértice de la malla poligonal una coordenada de textura; así, la coordenada de textura para un punto sobre una cara de la malla poligonal se calcula interpolando las coordenadas para los vértices que componen dicha cara.

En los mapas de altura y densidad se asume que la malla poligonal representa el relieve de una sección rectangular de terreno. Así, las coordenadas (s,t) son simplemente coordenadas cartesianas dentro de este rectángulo. Por simplicidad, se asume de aquí en adelante que la sección de terreno está orientada y posicionada con respecto al sistema de coordenadas del mundo de modo que $s = x$ y $t = y$. Esto permite definir las coordenadas de textura como

$$u(x,y) = x/\bar{x}$$

$$v(x,y) = y/\bar{y}$$

donde $\bar{x}$ e $\bar{y}$ son las dimensiones de la sección de terreno en los ejes $\hat{x}$ e $\hat{y}$, respectivamente.

En el caso del mapa de altura, el color es interpretado como la elevación del terreno en cada punto. Así, la elevación z para un punto cualquiera del terreno está dada por $\texttt{sample}_H(x,y)$, donde H es una imagen en escala de grises. Una propiedad particular de este mapeo es que permite parametrizar la superficie en función de la imagen y el tamaño de la sección de terreno. Así, la superficie del terreno queda definida por la ecuación

$$z = z_{min} + (z_{max} - z_{min}) \cdot \texttt{sample}_H\left(\frac{x - x_{min}}{x_{max} - x_{min}}, \frac{y - y_{min}}{y_{max} - y_{min}}\right) \tag{3.1}$$

Esto hace posible construir una malla poligonal a partir un mapa de altura.

El mapa de densidad, de manera similar, mapea una imagen en escala de grises al terreno, pero esta se interpreta como una función de densidad por área. Para ello se define un valor de densidad mínimo y máximo, y se utiliza el valor obtenido al muestrear la imagen para

interpolar entre ellos. Por ejemplo, si el mínimo es $0\frac{u.}{m^2}$ y el máximo es $30\frac{u.}{m^2}$, un valor de 0.5 será interpretado como $15\frac{u.}{m^2}$.

3.2.1.2. Espacio de cómputo

Como se explicó en la sección 2.1, la arquitectura de una GPU es de tipo SIMD, lo que quiere decir que está optimizada para aplicar una misma operación (o serie de operaciones) a grandes cantidades de datos, lo que se conoce como "paralelismo de datos". Paralelizar un problema de esta manera requiere que la solución pueda ser encontrada subdividiendo el dominio del problema y resolviendo por separado el problema en cada uno de los subdominios.

Muchos problemas en el ámbito de la computación gráfica son de esta naturaleza. Por ejemplo, generar una imagen equivale a determinar separadamente el color de cada uno de los píxeles que la componen, y aplicar una transformación espacial a una malla de triángulos se logra multiplicando cada uno de sus vértices por una misma matriz. Por esta razón, los *shaders* tradicionales se definen sobre espacios de cómputo como los vértices de una malla (*vertex shader*) o los píxeles de una imagen (*pixel shader*, también llamado *fragment shader*).

Cuando se utiliza la GPU para realizar operaciones fuera del *pipeline* gráfico[3] no existe un espacio de cómputo predefinido. En su lugar existe un espacio de cómputo abstracto que refleja el modelo de ejecución de las operaciones en la GPU.

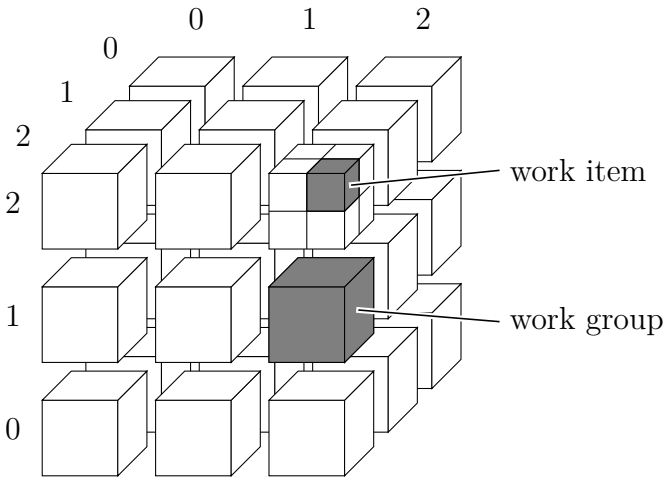


Figura 3.3: Visualización del espacio de cómputo.

[3] El pipeline gráfico es el conjunto de operaciones que se llevan a cabo en la GPU para producir una imagen a partir de la especificación geométrica de una escena.

Los distintos hilos de ejecución de un compute shader se agrupan en *work groups*[4], es decir, grillas de hasta tres dimensiones donde cada hilo tiene un identificador individual. Estos grupos están a su vez organizados en un espacio tridimensional, también con identificadores únicos.

Los work groups representan hilos de ejecución (también llamados *work items*) que se ejecutan en paralelo, potencialmente dentro de la misma unidad SIMD en el hardware de la GPU. Por esta razón, los hilos de un mismo grupo puede sincronizar su ejecución y, además, acceder a algunos recursos compartidos adicionales.

Todos los work groups tienen las mismas dimensiones, las cuales están especificadas en el código fuente del compute shader, mientras que la cantidad total de grupos es especificada por el usuario al despachar el compute shader. Si, por ejemplo, se despachan $4 \times 2 \times 4$ grupos de $8 \times 8 \times 1$ elementos cada uno, entonces el compute shader se ejecutará un total de 2048 veces. Además, los identificadores de los grupos serán números enteros entre 0 y 3, para los ejes x y z, y 0 a 1 para el eje y. Análogamente, los índices de los hilos dentro de un mismo grupo estarán entre 0 y 8 para los ejes x e y, mientras que en el eje z sólo podrán tomar el valor 0.

Al diseñar un algoritmo para ser ejecutado en GPU, es necesario establecer una correspondencia entre el espacio de cómputo y el dominio del problema. Generalmente esto se hace a través de una función inyectiva que transforma el índice de cada grupo e hilo en un valor apropiado al problema, como, por ejemplo, un índice para acceder a un arreglo.

3.2.2. Etapas

En esta sección se describen las cuatro etapas mencionadas al principio de la sección 3.2: generación, evaluación, indización y copia. En términos generales, la etapa de generación produce una cantidad determinada de puntos "candidatos". Luego, en la etapa de evaluación se selecciona un subconjunto de estos como puntos válidos de acuerdo a una variedad de criterios. Finalmente, las fases de indización y copia permiten enumerar puntos válidos para así copiarlos a un nuevo arreglo, donde puedan ser usados con mayor facilidad.

[4] Éste es el nombre que reciben en OpenGL y las otras APIs desarrolladas por Khronos Group. En CUDA, la API de cómputo de Nvidia, estos bloques reciben el nombre de *warp*.

3.2.2.1. Generación

El propósito esta etapa es generar una serie de posiciones que puedan dar lugar a objetos. Para cada posición, se inicializan también los atributos requeridos por las etapas posteriores del algoritmo de posicionamiento. El conjunto de una posición y sus atributos adicionales se denomina *candidato*.

Puesto que se espera que las posiciones generadas correspondan a objetos físicos en el mundo, éstas deben estar espaciadas de modo que los objetos no colisionen entre sí. Esto significa que cada punto tiene un radio dentro del cuál no puede generarse ningún otro punto. Para simplificar el problema, se asume que todos los objetos posicionados con una misma ejecución del algoritmo tienen el mismo radio de colisión y, adicionalmente, la distancia entre puntos se calcula sobre el plano, es decir, obviando la elevación del terreno. La separación mínima entre dos objetos se denominará huella (o *footprint*), y es igual a dos veces el radio de colisión. Esta restricción tiene la consecuencia de que, para generar objetos de distintos tamaños, se debe ejecutar el algoritmo varias veces y luego solucionar las colisiones entre ellos.

Una forma común de generar posiciones separadas por un distancia mínima es utilizando el algoritmo de "lanzamiento de dardos". Este consiste en, iterativamente, generar una posición al azar dentro de límites fijos, compararla con las posiciones anteriormente generadas, y añadirla al conjunto de posiciones si no se produjo una colisión. Esto se repite hasta tener la cantidad deseada de posiciones. Cabe mencionar que es posible construir una estructura para realizar el cálculo de colisiones en tiempo constante. La distribución generada por este algoritmo se conoce como distribución de Poisson de disco.

Código 3.2: El algoritmo de lanzamiento de dardos.

```
input: x_min, x_max, y_min, y_max, N
begin
    posiciones ← {}
    while |posiciones| < N
        x ← random(x_min, x_max)
        y ← random(y_min, y_max)
        if colisiones(posiciones, (x, y)) = 0
            posiciones ← {(x, y)} ∪ posiciones
    return posiciones
end
```

Éste método, sin embargo, es incompatible con los objetivos de diseño establecidos previamente. Es un algoritmo estrictamente secuencial, pues para calcular la n-ésima posición es

necesario conocer las $n-1$ posiciones anteriores para determinar si se producen colisiones. Esto no sólo hace imposible acelerarlo mediante el uso de la GPU, sino que además requiere que se calculen todas las posiciones para todo el mundo de una sola vez. Si se generan las posiciones para una región reducida del espacio de posicionamiento se corre el riesgo de que hayan colisiones con las regiones aledañas. Para resolver dichas colisiones, es necesario conocer las posiciones en las regiones aledañas. Para generar estas regiones, es necesario conocer las regiones aledañas a estas y así sucesivamente hasta llegar a los bordes del área de posicionamiento. Esto es un problema porque se asumió que el mundo es demasiado grande como para mantenerlo en memoria, y se busca que las posiciones puedan ser calculadas en tiempo de ejecución.

Para resolver este problema, se decidió sacrificar parcialmente la calidad de las posiciones generadas con tal de poder calcularlas sobre para un área reducida del mundo y obtener los mismos resultados cada vez. Para ello, se subdivide el espacio (no sólo el área de posicionamiento) en una grilla uniforme con celdas de tamaño arbitrario. Luego, se generan posiciones dentro de un espacio equivalente al tamaño de las celdas. Estas posiciones deben haber sido elegidas de modo que al desplazarlas en cualquier dirección a lo largo de los ejes x e y una distancia equivalente al tamaño de las celdas y compararlas contra las originales no se produzcan colisiones. Dicho de otro modo, se construye un patrón de posiciones que al repetirlo sobre la grilla no produce colisiones.

Existe más de un método para lograr esto. Una forma trivial es eligiendo posiciones en una grilla con celdas de tamaño igual a la huella de los objetos. También es posible modificar el cálculo de colisiones del algoritmo de lanzamiento de dardos para obtener este resultado.

En principio, el patrón de posicionamiento depende de la huella de los objetos a posicionar. Para permitir que el patrón sea reutilizado para objetos con distintos tamaños, se puede construir asumiendo una huella unitaria. Esto es, con una separación mínima igual a 1 y expresando el tamaño de las celdas en función a esta. Esto permite que tanto el patrón como la grilla sean multiplicados por el tamaño efectivo de la huella para obtener las posiciones adecuadas.

Para acomodar distintas formas de construir este patrón se añaden dos parámetros a esta etapa: el patrón de posiciones y el tamaño de la grilla. Ambos deben estar especificados en términos de una huella unitaria. Por defecto, el patrón se genera utilizando el método de lanzamiento de dardos modificado.

Este método de patrón y grilla da lugar a una correspondencia natural con los work groups y work items de un compute shader. Los grupos se corresponden con las celdas de la grilla,

y los hilos, con los elementos del patrón. Con esto, se debe especificar además que el patrón debe contener tantos elementos como hilos tiene un grupo del compute shader.

De este modo, es posible especificar sobre qué área del mundo generar posiciones mediante la cantidad de grupos despachados, en conjunto con un parámetro adicional denominado el índice base. Así, si se despachan $n \times m \times 1$ grupos de trabajo con un índice base igual a $(\bar{i}, \bar{j})$, el índice de la celda correspondiente al grupo de trabajo con índices $(i, j, 0)$ será, simplemente, $(\bar{i} + i, \bar{j} + j)$. A este índice se le denominará índice global, denotado por $g = (g_i, g_j)$.

Luego, la fórmula para generar posiciones es

$$p_{g_i,g_j} = (lg + P_{j,i})h \tag{3.2}$$

donde (i, j) es el índice local (work item), h la huella, l el tamaño de las celdas (asumiendo que son cuadradas) y $P \in \mathcal{M}_{m \times n}$ una matriz que contiene el patrón de posiciones [5]. La figura 3.4 ilustra el funcionamiento del patrón de posicionamiento con una matriz de 4×4.

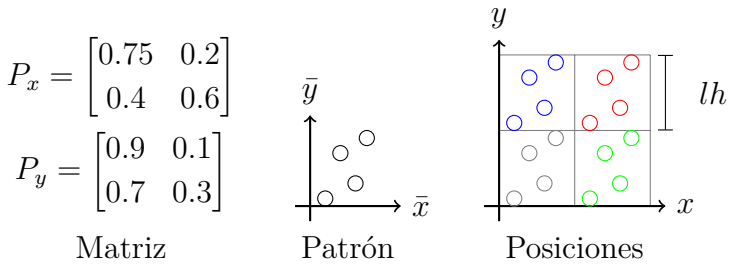


Figura 3.4: Utilización del patrón de posicionamiento.

Es fácil notar que el conjunto de puntos generados para una celda específica depende únicamente de estos parámetros. En particular, no depende de las celdas aledañas ni del orden en que se calculan las posiciones. Esto significa que esta forma de generar posiciones cumple con los requerimientos de determinismo establecidos previamente.

Conociendo la posición horizontal, se puede utilizar la fórmula 3.1 para determinar la altura del punto. El resultado de dividir la posición horizontal por el tamaño del mundo se conoce como posición normalizada, denotada $\bar{p}$. La posición normalizada es un atributo del

[5] Técnicamente, esta no es una matriz sino un tensor dimensiones $2 \times m \times n$, pues contiene vectores de dos elementos. Sin embargo, se le llama matriz pues en la práctica puede ser implementada como una matriz de vectores: `vec2[N][M]`

candidato, y puede ser utilizado en las etapas siguientes.

Recapitulando, la fase de generación toma como entrada el número de work groups, el índice base, el patrón de posicionamiento, el tamaño de las celdas, y la huella. La salida de la etapa es un conjunto de candidatos. Cada candidato tiene una posición tridimensional acorde con el mapa de altura, y una posición normalizada que puede ser usada para muestrear el mapa de altura o de densidad. En la práctica, estos candidatos se almacenan en un arreglo.

3.2.2.2. Evaluación

Esta etapa se encarga de determinar la validez de los candidatos generados en la etapa anterior. Un candidato válido es aquel que forma parte del resultado final y, por tanto, da lugar a un objeto en el mundo. Para esto se utiliza un mapa de densidad; las zonas con valores más altos tendrán más candidatos válidos que las zonas con valores más bajos. En el caso extremo, un mapa de densidad constante igual a 1 (es decir, una imagen en blanco) validará todos los candidatos, mientras que uno con densidad constante igual a 0 no validará ninguno.

La forma más simple de lograr este efecto es, para cada candidato, generar un número aleatorio en el rango $[0, 1]$, muestrear el mapa de densidad de la forma explicada en la sección 3.2.1.1 y comparar ambos valores. Si el número generado excede el valor muestreado, entonces el candidato es válido.

Sin embargo, esta técnica es compleja de paralelizar, sobre todo si se busca que los resultados sean deterministas. Si bien la generación de números pseudo-aleatorios es un proceso donde el orden en que los números son generados es determinista, el orden en que se evalúan los candidatos en la GPU no lo es. Esto causaría que al evaluar el mismo conjunto de candidatos dos veces, con los mismos parámetros cada vez, se obtengan valores distintos. Una forma de resolver este problema es utilizar un generador de números aleatorios por candidato en lugar de uno global, y elegir una semilla para el generador usando una función de la posición.

Por simplicidad, y al igual que en el método utilizado en Horizon: Zero Dawn, se ha optado por un sistema que no requiera la generación de números pseudo-aleatorios. Usando el mismo esquema de ordenamiento en celdas de la sección anterior, se reemplaza el patrón de posicionamiento por una matriz de números en el rango $[0, 1]$. En particular, se utiliza una matriz de *dithering*, la cual proviene de la técnica de computación gráfica del mismo nombre (a veces también llamada "tramado"). El dithering permite reducir la profundidad de bits del

color de una imagen, es decir, la cantidad de colores distintos que contiene. Por ejemplo, una imagen con 256 niveles discretos (8 bits) por canal puede ser reducida a 64 niveles (6 bits), o incluso a 16 (4 bits). Si bien existen distintos algoritmos de dithering, las matrices que se mencionan en este caso constan de valores uniformemente distribuidos y sin repetición, como se ejemplifica en las ecuaciones 3.3 y 3.4. Esta propiedad permite utilizarlas como sustituto de una variable aleatoria uniformemente distribuida en el rango $[0, 1]$.

$$M_{2\times 2} = \frac{1}{4}\begin{bmatrix} 0 & 2 \\ 3 & 1 \end{bmatrix} \tag{3.3}$$

$$M_{4\times 4} = \frac{1}{16}\begin{bmatrix} 0 & 8 & 2 & 10 \\ 12 & 4 & 14 & 6 \\ 3 & 11 & 1 & 9 \\ 15 & 7 & 13 & 5 \end{bmatrix} \tag{3.4}$$

La matriz de dithering escogida debe tener las mismas dimensiones que los work groups. Así, se puede determinar si un candidato será válido de manera similar a como se escoge su posición, evaluando la siguiente expresión:

$$\texttt{sample}_D(\bar{p}_x, \bar{p}_y) > M_{j,i} \tag{3.5}$$

donde D es el mapa de densidad, $\bar{p}$ es la posición normalizada del candidato, (i, j) es el índice local y M es la matriz de dithering.

Como se señala en el código 3.1, esta etapa puede ser repetida varias veces para un mismo conjunto de candidatos. Si se utiliza una variable auxiliar para acumular la densidad de cada candidato, se puede lograr que cada nuevo mapa evaluado añada candidatos válidos al conjunto, como se muestra en el código 3.3. El índice de la capa que validó el candidato puede ser añadido al resultado final.

Código 3.3: Evaluación de varios mapas de densidad con un mismo candidato.

```
1 input: densitymaps, posicion_norm, matriz_dithering
2 begin
3     indice_capa ← 0
4     densidad_acumulada ← 0
```

```
    k ← 1
    while k ≤ |densitymaps|
        // ecuación 3.5 :
        densidad_acumulada ← densidad_acumulada + sample(densitymaps[k - 1],
     ↪ posicion_norm)
        if indice_capa = 0 and densidad_acumulada > matriz_dithering[i][j]
            indice_capa ← k
        k ← k + 1
    return k
end
```

En resumen, esta etapa toma como argumentos un conjunto de candidatos, una lista de mapas de densidad y una matriz de dithering. Además, se deben despachar la misma cantidad de work groups que en la fase de generación, con el mismo tamaño e índice base. El valor de retorno es es el mismo conjunto de candidatos, donde a cada candidato se le ha asignado el índice de uno de los mapas de densidad, o el valor 0 si es un candidato inválido.

Si las posiciones van a ser usadas en CPU, es posible terminar el algoritmo de posicionamiento en esta etapa. Simplemente es necesario copiar el arreglo a la memoria principal e iterar sobre él eliminando los candidatos con un índice de capa nulo (marcado con $\emptyset$).

3.2.2.3. Indización y copia

Estas dos etapas existen para optimizar el caso en que los resultados son utilizados en la misma GPU. Al final de la etapa anterior se obtiene un arreglo de candidatos, algunos de los cuales son inválidos. La presencia de estos elementos en el arreglo dificulta su utilización para operaciones comunes, como el dibujo instanciado[6], pues éstas suelen esperar que todos los elementos en un arreglo sean válidos. El hecho de que los objetos estén desordenados también hace imposible seleccionar sólo aquellos que tengan un índice de capa específico.

Los algoritmos de indización y copia, en su conjunto, tienen la función de eliminar los elementos inválidos del arreglo de candidatos. La forma tradicional de lograr esto es iterando secuencialmente sobre el arreglo, pero realizar esta operación en GPU es ineficiente. Copiar los resultados a la CPU, reordenarlos y luego copiarlos de vuelta a la GPU puede ser más rápido, pero también es altamente ineficiente. En lo posible, es preferible evitar este tipo de

[6] El dibujo instanciado es una técnica para dibujar repetidamente objetos similares, donde al menos uno de los parámetros toma su valor desde un arreglo en la memoria de la GPU en lugar de ser especificados cada vez desde la CPU.

"vueltas en círculo".

El algoritmo de indización asigna índices secuenciales a los elementos del arreglo de candidatos de modo que cada candidato válido tiene un índice único y todos los índices entre 1 y la cantidad de candidatos válidos tienen un candidato asignado. Es decir, el algoritmo enumera los candidatos válidos. Con esto es posible crear un nuevo arreglo y copiar cada candidato válido a la posición en el arreglo correspondiente al índice que le fue asignado. Esto último es la función del algoritmo de copia.

En el diseño del algoritmo de indización, la observación clave es que el índice de un candidato válido es igual a la cantidad de elementos válidos que le preceden en el arreglo. Esta cantidad puede ser computada en paralelo dividiendo los elementos que le preceden en N grupos, contando la cantidad de elementos válidos en cada uno de ellos, y sumando los resultados para cada uno de estos grupos. Como la cantidad de elementos válidos en un arreglo es igual al índice de su último elemento, contar los elementos válidos en cada uno de los subgrupos se puede realizar aplicando el algoritmo recursivamente hasta llegar a un caso trivial (un arreglo de un solo elemento). Así, el índice del último elemento del grupo n es igual a la cantidad de elementos en ese grupo en sumada con las cantidades los $n - 1$ anteriores.

La figura 3.5 ilustra un método paralelizable para calcular la suma de cada elemento del arreglo con todos los elementos que le preceden. En la primera iteración ($t = 0$), a los elementos impares del arreglo (x_1, x_3, x_5 y x_7) se les suma el valor del elemento que les precede. En $t = 1$, el arreglo se subidivide en subarreglos de largo 2, de modo que los índices 0 y 1 conforman el primer subarreglo, 2 y 3 el segundo, y así sucesivamente. Luego, de manera similar a en la primera iteración, a cada subarreglo impar se le suma el valor del último elemento del subarreglo que les precede. Por ejemplo, a los elementos en las posiciones 2 y 3 se les suma el valor del elemento en la posición 1. Finalmente, en $t = 3$ se repite el paso anterior subdividiendo en arreglos de largo 4 para obtener el estado observado en $t = 3$.

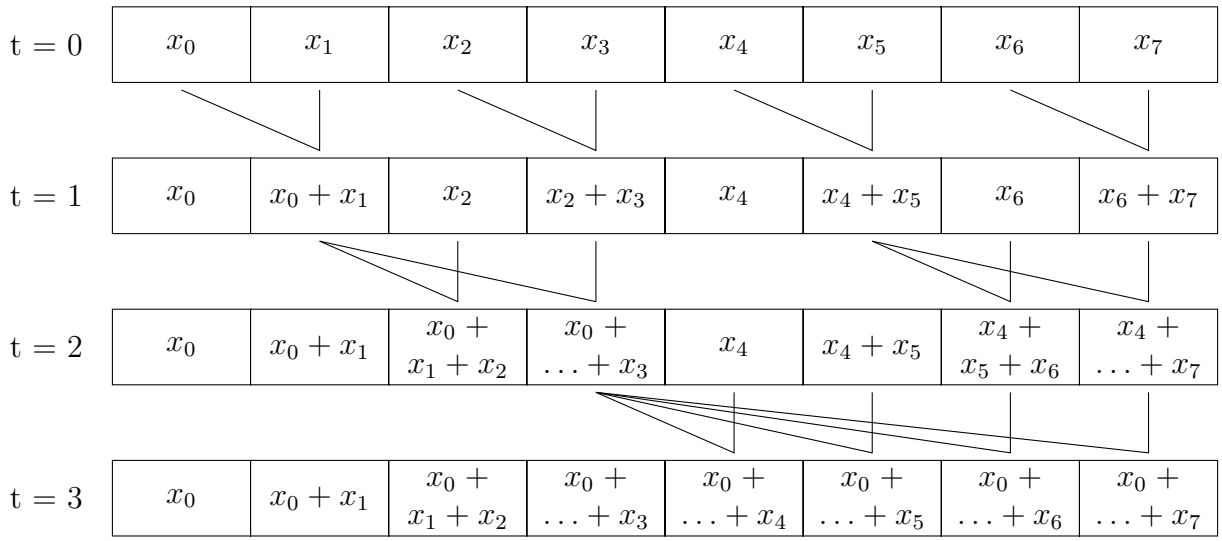


Figura 3.5: Algoritmo de suma paralelizable.

Nótese que, en cada una de estas iteraciones, los primeros 2^t elementos del arreglo cumplen con la condición buscada, esto es, se les han sumado todos los elementos que le preceden. Además, la primera iteración puede entenderse como una subdivisión del arreglo en subarreglos de largo 1, con lo que se hace evidente que el tamaño de estos subarreglos es, de hecho, 2^t. Por otro lado se puede observar que, si bien este algoritmo funciona con arreglos de largo arbitrario, cuando el largo es una potencia de dos ($N = 2^k$) la cantidad de sumas realizadas en cada iteración permanece constante y es exactamente la mitad del largo del arreglo ($N/2$). Por último, es simple deducir que el algoritmo toma $log_2(N)$ iteraciones y realiza un total de $(N/2)log_2(N)$ sumas.

Este algoritmo puede adaptarse al contexto de ejecución en GPU fácilmente, despachando un ítem por cada suma. Por ejemplo, un work group de 32 hilos ($32 \times 1 \times 1$) puede calcular la suma de un arreglo de 64 elementos en 6 iteraciones. En la práctica, este se utiliza para sumar la cantidad de elementos válidos generados al interior de un work group, la cual luego es añadida al total global utilizando operaciones atómicas [7]. Con ello, es posible asignar un índice único a cada elemento válido.

El arreglo se construye de modo que $x_i = 1$ si el i-ésimo elemento es válido, y $x_i = 0$ si no. De este modo, luego de ejecutar el algoritmo, cada posición en el arreglo contendrá la cantidad de elementos válidos hasta esa posición. Así, los elementos válidos pueden ser copiados a otro arreglo de forma directa. Es decir, si `a` es el arreglo de origen, `b` el arreglo de destino, y x_i la cantidad de elementos válidos hasta la posición i en el arreglo de origen, cada elemento

[7] Es decir, que se comportan como si fueran una sola instrucción del procesador y, por tanto, son seguras de utilizar en un contexto concurrente.

válido se copia según $\texttt{b}[x_i - 1] \leftarrow \texttt{a}[i]$.

El código 3.4 muestra como realizar este proceso en paralelo, de acuerdo con el modelo de ejecución de un compute shader. En la práctica, este algoritmo se ejecuta dentro de un mismo work group, pues no es posible sincronizar la ejecución entre work groups distintos. Esta sincronización es necesaria para asegurar que las sumas de una iteración se hayan completado antes de la siguiente. Para obtener el índice final de cada elemento, se utiliza un contador global donde cada work group añade la cantidad de elementos válidos que contabilizó utilizando operaciones atómicas.

Código 3.4: Asignación de índices en paralelo.

```
input: candidatos
output: indices

// Se asume tamaño y cantidad de grupos unidimensionales (n × 1 × 1).
// La cantidad total de hilos es igual a |candidatos| / 2
// indice_global_hilo es el índice global del hilo de ejecución.

begin
    indices[indice_global_hilo] ← 1 if candidatos.indice_capa > 0 else 0
    indices[|candidatos| + indice_global_hilo] ← 1 if candidatos.indice_capa > 0 else 0

    tamaño_subgrupo ← 1
    while tamaño_subgrupo < |candidatos|
        indice_base = (2 × (indice_global_hilo / tamaño_subgrupo) + 1) × tamañ
     ↪ o_subgrupo
        indices[indice_base + indice_global_hilo % tamaño_grupo] += indices[indice_base -
     ↪ 1]
        barrier() // sincroniza la ejecución de los hilos
end
```

3.3. Arquitectura de software

3.3.1. Tecnologías y dependencias

La biblioteca está escrita en C++ y utiliza OpenGL para acceder a los recursos de la GPU. C++ es un lenguaje de bajo nivel con soporte para paradigmas de más alto nivel, como la

programación orientada a objetos. Es ampliamente usado en la programación de software de sistemas pues, al igual que C, permite un acceso casi directo a los recursos del computador. Esto lo hace también un lenguaje idóneo para aplicaciones de alto rendimiento, entre las cuales se cuentan los videojuegos. Por ejemplo, Unreal Engine, uno de los motores de videojuegos más usados en la actualidad, está escrito en C++. Por esta razón es que es una elección natural para un proyecto en el ámbito de los videojuegos.

Para poder utilizar la GPU con fines de cómputo existen dos opciones: acceder a través de una API de cómputo (CUDA, OpenCL), o de una API gráfica. Puesto que la amplia mayoría de los videojuegos modernos ya utilizan una API gráfica, lo ideal es hacer uso de ésta en lugar de una API de cómputo. Idealmente, la biblioteca sería compatible con las APIs utilizadas en la industria, sin embargo, el costo de desarrollo de replicar el trabajo a través de varias APIs y sistemas operativos escapa al alcance de este trabajo.

Direct3D, OpenGl, Vulkan y Metal son las APIs más utilizadas en el desarrollo de videojuegos. Direct3D y Metal son exclusivas a Windows y MacOS, respectivamente, mientras que OpenGL y Vulkan son un estándar abierto y están disponibles tanto en Windows como en Linux. Todas éstas poseen las capacidades necesarias para desarrollar este proyecto, por lo que el criterio principal de elección es la compatibilidad. En ese sentido, Vulkan y OpenGL son los mejores candidatos y, entre éstos dos, OpenGL es preferible por ser más simple de usar que Vulkan.

Cabe mencionar que también existe la posibilidad de implementar el sistema usando un motor gráfico o de videojuegos como Unreal Engine. La ventaja de esta aproximación es que permiten abstraer la API gráfica, facilitando mayor compatibilidad. Sin embargo, una extensión desarrollada para un motor específico sólo puede ser utilizada en éste, además de tener un costo de desarrollo adicional debido a la necesidad de aprender a usar y extender el motor en cuestión.

3.3.2. Interfaz y modo de uso

A continuación se encuentra una lista de los elementos más importantes de la API de la biblioteca. Más adelante (secciones 3.3.2.1 a 3.3.2.5) se incluye una descripción detallada de cada elemento de ésta.

- **El contexto de posicionamiento**, denominado *PlacementPipeline*. Encapsula todos los recursos utilizados por el algoritmo de posicionamiento y le permite al usuario eje-

cutarlo.

- **La descripción del terreno**, denominada *WorldData*. Consiste de un mapa de altura, que describe el relieve del terreno, junto a un factor de escala que especifica las dimensiones del mundo.
- **El conjunto de objetos a posicionar**, denominado *LayerData*. Especifica la separación mínima entre objetos, denominada *footprint*, y una lista de capas de posicionamiento. Cada capa corresponde a un mapa de densidad, es decir, una textura en escala de grises.
- **Los resultados del posicionamiento**, *FutureResult* y *Result*. Describen los resultados del proceso de posicionamiento y permiten acceder a ellos.

El fragmento de pseudo-código 3.5 muestra, de forma simple, cómo ejecutar el algoritmo de posicionamiento. Los nombres precedidos por `Placement::` pertenecen a la biblioteca. Todos los demás corresponden a código del usuario.

Código 3.5: Ejemplo de uso en pseudo-código

```
// Las clases que pertenecen a la biblioteca de posicionamiento están precedidas
// por Placement::, el resto corresponde a clases y funciones de ejemplo.

// 1. Configurar el contexto de OpenGL.
window ← Window(...)
window.makeGLContextCurrent()

Placement::loadGLContext()

// 2. Crear un contexto de posicionamiento.
pipeline ← Placement::PlacementPipeline()

// 3. Crear el mundo de juego y un WorldData para describirlo.
game_world ← GameWorld( ... ) // esta es un clase definida por el usuario

world_data ← Placement::WorldData(game_world.size, game_world.heightmap_texture)

// 4. Construir uno (o más) LayerData para los objetos a posicionar.
a_texture ← SomeFunction( ... )
another_texture ← AnotherFunction( ... )
footprint ← 1.0f
layer_data ← Placement::LayerData(footprint, { DensityMap(a_texture), DensityMap(
    ↪ another_texture) })

// 5. Ejecutar el algoritmo, especificando el área sobre la que computar posiciones.
```

```
placement_start ← vec2(12, 34)
placement_end ← vec2(56, 78)
future_result ← pipeline.computePlacement(world_data, layer_data, placement_start,
    ↪ placement_end)

// 6. Esperar y leer los resultados.
result ← future_result.readResult()

for element in result.copyLayerToHost(0):
   game_world.addObject(AnObject(), element.position)

for element in result.copyLayerToHost(1):
   game_world.addObject(AnotherObject(), element.position)
```

3.3.2.1. Acceso al contexto de OpenGL

El "contexto" es un objeto específico de OpenGL que abstrae información sobre el entorno de ejecución, como el hardware y la versión específica de la API que se encuentran en uso. Antes de poder realizar cualquier tipo de operación con la GPU, es necesario haber creado un contexto. El contexto también contiene todos los objetos de OpenGL, como buffers y shaders, por lo que la biblioteca necesita acceder al contexto creado por la aplicación para poder transferir los datos con más facilidad. La creación del contexto es dependiente del sistema de ventanas del sistema operativo, por lo que es usual utilizar una biblioteca externa (como SDL y GLFW) para este proceso. De manera similar, la obtención de punteros a funciones de OpenGL también es dependiente del sistema operativo, por lo que existe otro conjunto de bibliotecas que se encargan de ello. Ambos procesos son responsabilidad del usuario. La función `loadGLContext()` le permite a la biblioteca acceder a las funciones del contexto de OpenGL. Su único argumento es una función de carga especificada por el usuario, usualmente provista por la biblioteca de manejo de ventanas en uso por la aplicación.

3.3.2.2. PlacementPipeline

Cómo se explica en la sección 3.2, el algoritmo de posicionamiento consta de varias fases. Esta clase abstrae los detalles de implementación del algoritmo a través de una interfaz más funcional[8]. La clase posee una única función de importancia, `computePlacement`, la cual ejecuta el cada una de las fases del algoritmo de posicionamiento y cuyos argumentos definen

[8] En el sentido de "programación funcional", es decir, sin efectos secundarios. El resultado de la función depende únicamente de sus argumentos.

completamente el resultado de la operación[9]. Si estos argumentos permanecen constantes, los resultados serán los mismos (salvo por el ordenamiento de los elementos individuales dentro del arreglo de resultados).

La función `computePlacement` tiene cuatro parámetros. En primer lugar, `world_data` especifica las características del mundo en el que se realiza el posicionamiento. Luego, `layer_data` define el tamaño de los objetos a posicionar y los mapas de densidad a utilizar. Se permite especificar múltiples mapas de densidad porque no es necesario repetir la fase de generación cuando el tamaño de los objetos es el mismo, pues se obtendría el mismo conjunto de candidatos cada vez. Más importante aún, si se conservan los valores de densidad obtenidos en cada iteración de la etapa de evaluación, entonces las capas posteriores pueden determinar fácilmente si ya existe un objeto en su posición y así garantizar la ausencia de colisiones.

Finalmente, `lower_bound` y `upper_bound` definen un área rectangular, con lados de largo mayor a 0 y menor a las dimensiones del mundo especificadas en `world_data`, dentro de la cual están contenidos todas las posiciones generadas. Es decir, estas posiciones $p = (p_x, p_y)$ están acotadas por `lower_bound` y `upper_bound` o, más explícitamente, cumplen que `lower_bound.x` $\leq p_x <$ `upper_bound.x` y `lower_bound.y` $\leq p_y <$ `upper_bound.y`.

PlacementPipeline
- generate : GenerationKernel - evaluate : EvaluationKernel - index : IndexationKernel - copy : CopyKernel
+ computePlacement(world_data : WorldData, layer_data : LayerData, lower_bound : vec2, upper_bound : vec2) : FutureResult

Figura 3.6: Resumen de la clase PlacementPipeline

Cómo se mencionó anteriormente, esta clase encapsula todos los recursos utilizados por el algoritmo de posicionamiento. En particular, establece una separación entre los objetos de OpenGL utilizados por la biblioteca de posicionamiento y aquellos creados por la aplicación. Únicamente las texturas utilizadas como mapas de densidad y los buffers que contienen los resultados pueden cruzar la barrera, lo que sólo ocurre cuando se llama a la función `computePlacement`. `world_data` y `layer_data` contienen las texturas de altura y densidad, respectivamente, mientras que el valor de retorno incorpora el buffer que contiene los resultados.

[9] En la práctica es deseable utilizar también un generador de números pseudoaleatorios. En tal caso, el resultado será también dependiente del estado de dicho generador.

Es importante señalar que la amplia mayoría de las funciones de OpenGL acceden a algún tipo de estado global, por lo que las operaciones realizadas en la biblioteca de posicionamiento pueden alterar el funcionamiento de la aplicación que la invoca de maneras imprevistas. Por esta razón la documentación de la API indica que operaciones alteran el estado global y qué aspectos de éste son modificados.

Reducir los puntos de la interfaz que involucran a la API gráfica directamente también facilita la tarea de adaptar la biblioteca a una API gráfica distinta.

3.3.2.3. WorldData

Esta estructura agrupa toda la información propia del terreno sobre el que se han de posicionar los objetos. En caso de querer extender el sistema para incorporar más parámetros de la superficie (como, por ejemplo, humedad) se debe modificar esta estructura.

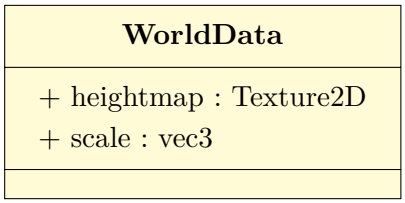


Figura 3.7: Resumen de la clase WorldData.

`heightmap` es el identificador de una textura de OpenGL que contiene el mapa de altura para el mundo, mientras que `scale` indica las dimensiones de dicho mapa de altura. Para encontrar el punto en el mapa de altura que se corresponde con un punto sobre la superficie del mundo, se deben dividir las componentes x e y de dicho punto por las del vector `scale`. Para transformar la elevación normalizada del mapa de altura a coordendas del mundo se debe multiplicar por la componente z del vector `scale`. En otras palabras, si $p = (x, y, z)$ es un punto sobre la superficie del terreno expresado en el espacio de coordenadas del mundo, $h(u, v)$ es la función definida por el mapa de altura (con $u, v \in [0, 1]$) y s es el vector de escala, entonces

$$z = h(\frac{x}{s_x}, \frac{y}{s_y})$$

3.3.2.4. LayerData

Esta clase agrupa varios mapas de densidad para objetos con un mismo *footprint*. Los mapas de densidad incluidos en un mismo LayerData son procesados juntos en una misma ejecución del algoritmo, y por esa razón se encuentran separados del `WorldData`. Está garantizado que la distancia entre cualquier par de objetos resultantes de una misma operación de posicionamiento será mayor a `footprint`, independientemente de qué mapa de densidad haya dado origen a cada uno. Los mapas de densidad son evaluados en el orden que se especifica en `densitymaps`. Esto significa que si los primeros mapas en ser evaluados llenan todo el espacio disponible, los mapas restantes no generarán ningún objeto debido al método de prevención de colisiones mencionado en la sección 3.3.2.2.

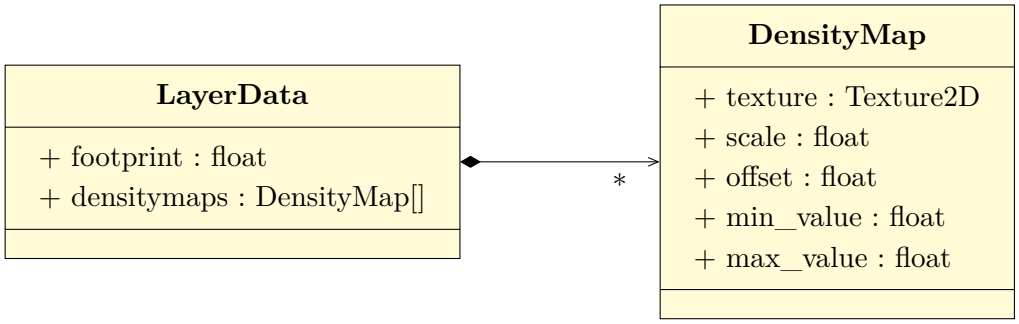


Figura 3.8: Resumen de las clases LayerData y DensityMap.

La estructura `DensityMap` especifica los parámetros de muestreo de un mapa de densidad específico. Si d es el valor obtenido al muestrear el mapa de densidad en un punto específico, el valor de densidad final $\bar{d}$ para el objeto es

$$\bar{d} = \max(\min(d \times \texttt{scale} + \texttt{offset}, \texttt{min\_value}), \texttt{max\_value})$$

La correspondencia entre el espacio de coordenadas del mundo y el del mapa de densidad se realiza de la misma forma que para el mapa de altura, utilizando la escala especificada en `WorldData`.

3.3.2.5. FutureResult y Result

Las operaciones en GPU son fundamentalmente concurrentes, por lo que los resultados de la operación de posicionamiento no están disponibles de forma inmediata. Puesto que no

es razonable suspender la ejecución del programa hasta que estén listos los resultados, la función `computePlacement` retorna un objeto de tipo `FutureResult` que contiene el buffer de OpenGL donde serán escritos los resultados en el futuro. Esto permite a la aplicación realizar otras tareas en el intertanto.

La función `isReady()` permite a la aplicación consultar si los resultados están listos para ser leídos. Por su parte, `wait()` suspende la ejecución del programa hasta que lo anterior suceda. Una vez que los resultados están disponibles, se puede utilizar la función `getResult()` para obtener un objeto de tipo `Result` que toma control del buffer de resultados, invalidando en el proceso el `FutureResult`. En este sentido, es un objeto de un solo uso.

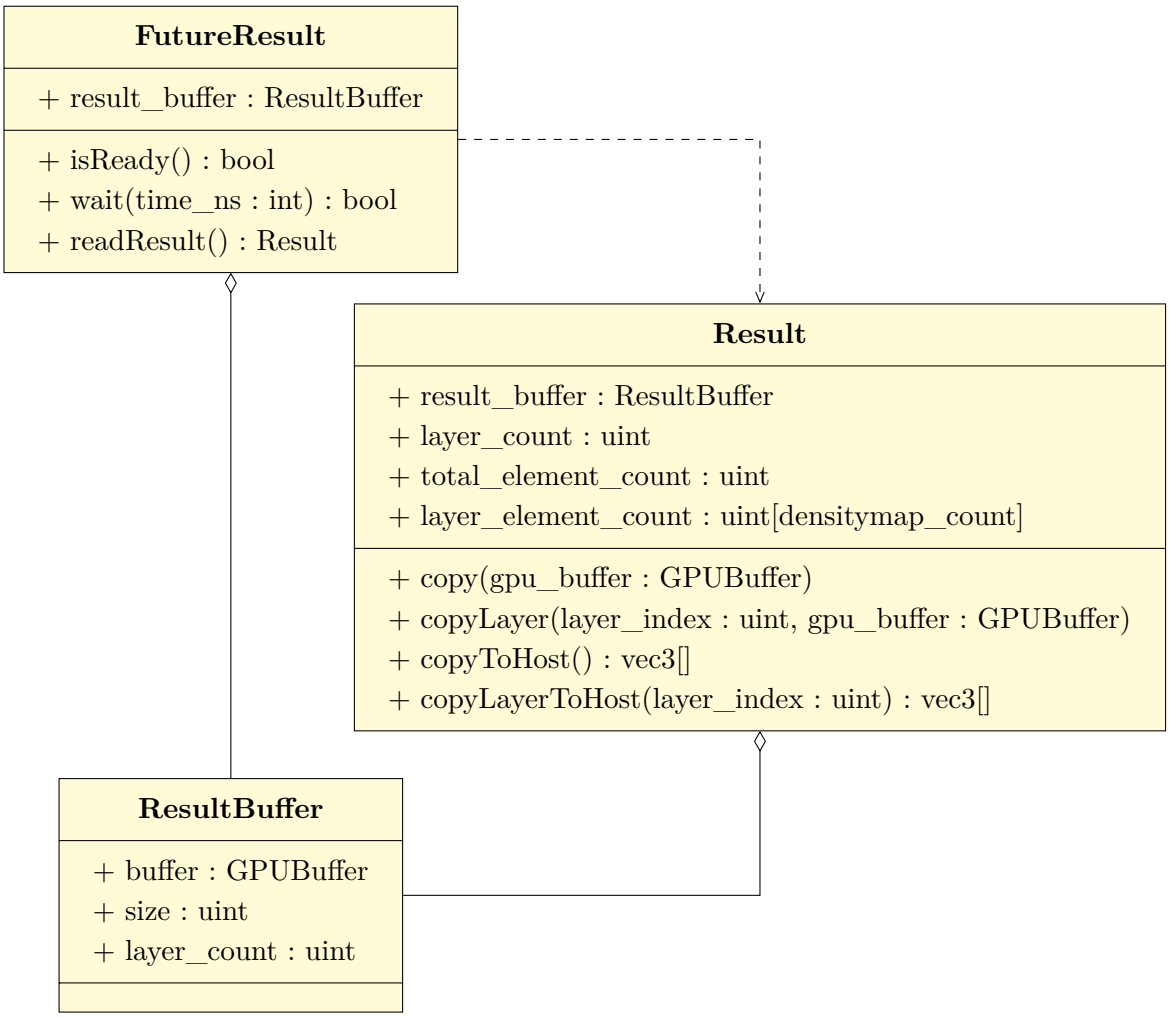


Figura 3.9: Resumen de las clases FutureResult y Result.

El objeto `Result` permite acceder a los contenidos del buffer de resultados. Provee varias funciones de utilidad, principalmente para consultar la cantidad de elementos generados y copiarlos a otro buffer o la memoria principal. También es válido acceder al buffer de OpenGL

directamente (usando `getBuffer()`), pues la distribución de los datos dentro de este está bien definida y especificada en la documentación.

3.3.3. Estructura interna

Como se mencionó en la sección anterior, PlacementPipeline encapsula los recursos utilizados por el algoritmo de posicionamiento. Los recursos en cuestión son de dos tipos: *compute shaders* y *buffers*. OpenGL ordena estos recursos en objetos que encapsulan su estado. A su vez, la biblioteca encapsula estos objetos para abstraer los detalles de OpenGL y facilitar la posibilidad de añadir soporte para otras APIs gráficas.

Los compute shaders están envueltos en clases denominadas kernels[10]. Estas clases se encargan de compilar, enlazar, configurar los argumentos y ejecutar el compute shader. Existe una clase kernel por cada etapa descrita en la sección 3.2: `GenerationKernel`, `EvaluationKernel`, `IndexationKernel` y `CopyKernel`. Cada una de estas clases contiene un `ComputeShaderProgram`, que abstrae a su vez la funcionalidad del Shader Object y el Program Object de OpenGL. Así, la estructura básica de un kernel es la descrita en la figura 3.10. El código fuente en GLSL del compute shader que cada una de estas clases utiliza se encuentra disponible en el anexo 5.1.

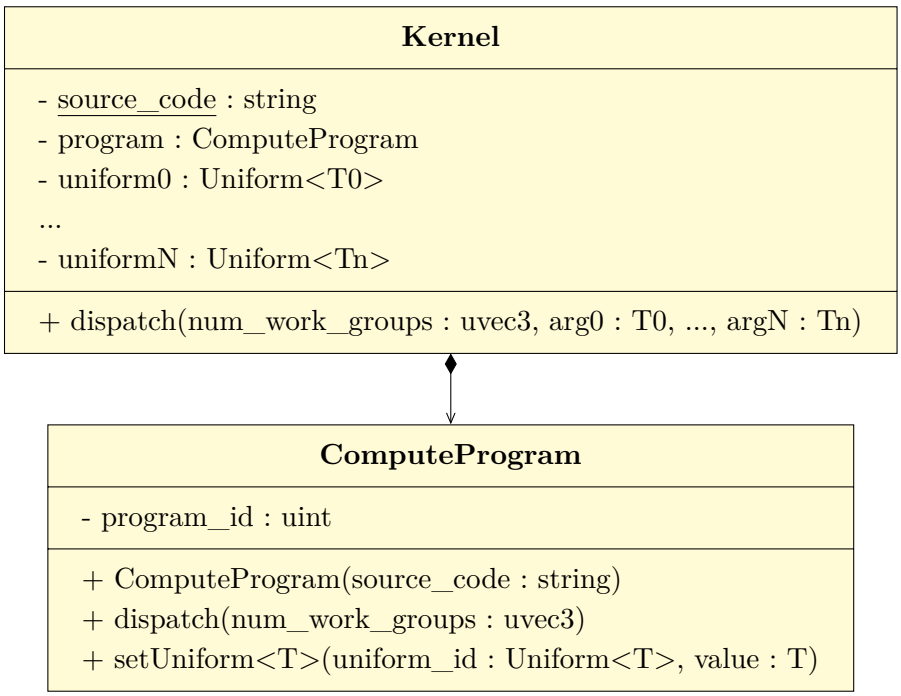


Figura 3.10: Estructura general de una clase Kernel.

[10] Este es el nombre que reciben los programas de cómputo en GPU en CUDA y OpenCL

Del manera similar, los buffers utilizados en el algoritmo de posicionamiento están encapsulados en dos clases: **TransientBuffer** y **ResultBuffer**. La primera de éstas se utiliza para almacenar datos intermedios, mientras que la segunda es donde se almacenan los resultados finales. La estructura general del **PlacementPipeline** es la indicada en la figura 3.6.

Código 3.6: Implementación del algoritmo de posicionamiento, en pseudocódigo.

```
input world_data : WorldData, layer_data : LayerData, lower_bound : vec2, upper_bound :
    ↪ vec2

begin
    generation_kernel ← Placement::GenerationKernel()
    evaluation_kernal ← Placement::EvaluationKernel()
    indexation_kernel ← Placement::IndexationKernel()
    copy_kernel ← Placement::CopyKernel()

    work_group_size : uvec2 ←  generation_kernel.work_group_size
    work_group_bounds : vec2 ← work_group_scale * layer_data.footprint
    work_group_offset : uvec2 ← lower_bound / work_group_bounds
    num_work_groups : uvec3 ← {1u + uvec2((upper_bound - lower_bound) /
     ↪ work_group_bounds), 1u}

    candidate_count : uint ← num_work_groups.x * num_work_groups.y *
     ↪ work_group_size.x * work_group_size.y

    transient_buffer ← GPUBuffer()
    transient_buffer.resize(candidate_count)

    // 1. Generación de candidatos
    generation_kernel(num_work_groups, work_group_offset, layer_data.footprint,
     ↪ world_data.scale, world_data.heightmap, transient_buffer)

    glMemoryBarrier( ... ) // función de OpenGL. Asegura que los cambios hechos por la
     ↪ invocación de un kernel sean visibles para los siguientes.

    // 2. Evaluación de candidatos.
    for layer_index in {0, ..., layer_data.densitymaps.size()}
        evaluation_kernel(num_work_groups, work_group_offset, layer_index, layer_data.
     ↪ densitymaps[layer_index], transient_buffer)
        glMemoryBarrier( ... )

    // 3. Asignación de indices.
    result_buffer ← Placement::ResultBuffer()
    result_buffer.resize(candidate_count)

```

```
33    indexation_kernel(/*num_work_groups=*/1u + candidate_count / (2 *
       ↪ indexation_kernelwork_group_size), transient_buffer, result_buffer)
34    glMemoryBarrier( ... )
35
36    // 4. Copia de resultados
37    copy_kernel(/*num_work_groups=*/ 1u + candidate_count / copy_kernel.
       ↪ work_group_size, transient_buffer, result_buffer)
38    glMemoryBarrier( ... )
39
40    return FutureResult(result_buffer)
41 end
```

Finalmente, cabe mencionar que para asegurar la correctitud de la implementación se utilizó un conjunto de pruebas automatizadas. Estas consisten en *tests* unitarios para el `PlacementPipeline` y para cada kernel que comparan el resultado contra una implementación de referencia en CPU. Además, se verifica que el resultado de cada invocación sea el mismo si los argumentos son los mismos.

Capítulo 4

Resultados

El código del proyecto se encuentra disponible en https://github.com/zerujio/procedural-placement-lib, bajo licencia MIT de código abierto. El repositorio contiene demos y ejemplos de uso, así como pruebas de correctitud y rendimiento. La sección próxima (4.1) expone el funcionamiento de la demo principal incluida en dicho repositorio, mientras que la sección de validación (4.2) muestra los resultados obtenidos al ejecutar las pruebas de rendimiento.

4.1. Demo

La demo consiste en una escena de aproximadamente 30 km^2, en la cual geometría del terreno es generada a partir de un mapa de altura. La librería de posicionamiento procedimental se utiliza para calcular las posiciones de árboles y rocas en tiempo de ejecución. Por defecto, se calcula un área de casi $1km^2$ en torno a un punto de referencia. Al desplazar la cámara por el terreno este punto se desplaza en conjunto, y las posiciones se vuelven a calcular dinámicamente cuando ésta se ha alejado lo suficiente del centro de las posiciones actuales.

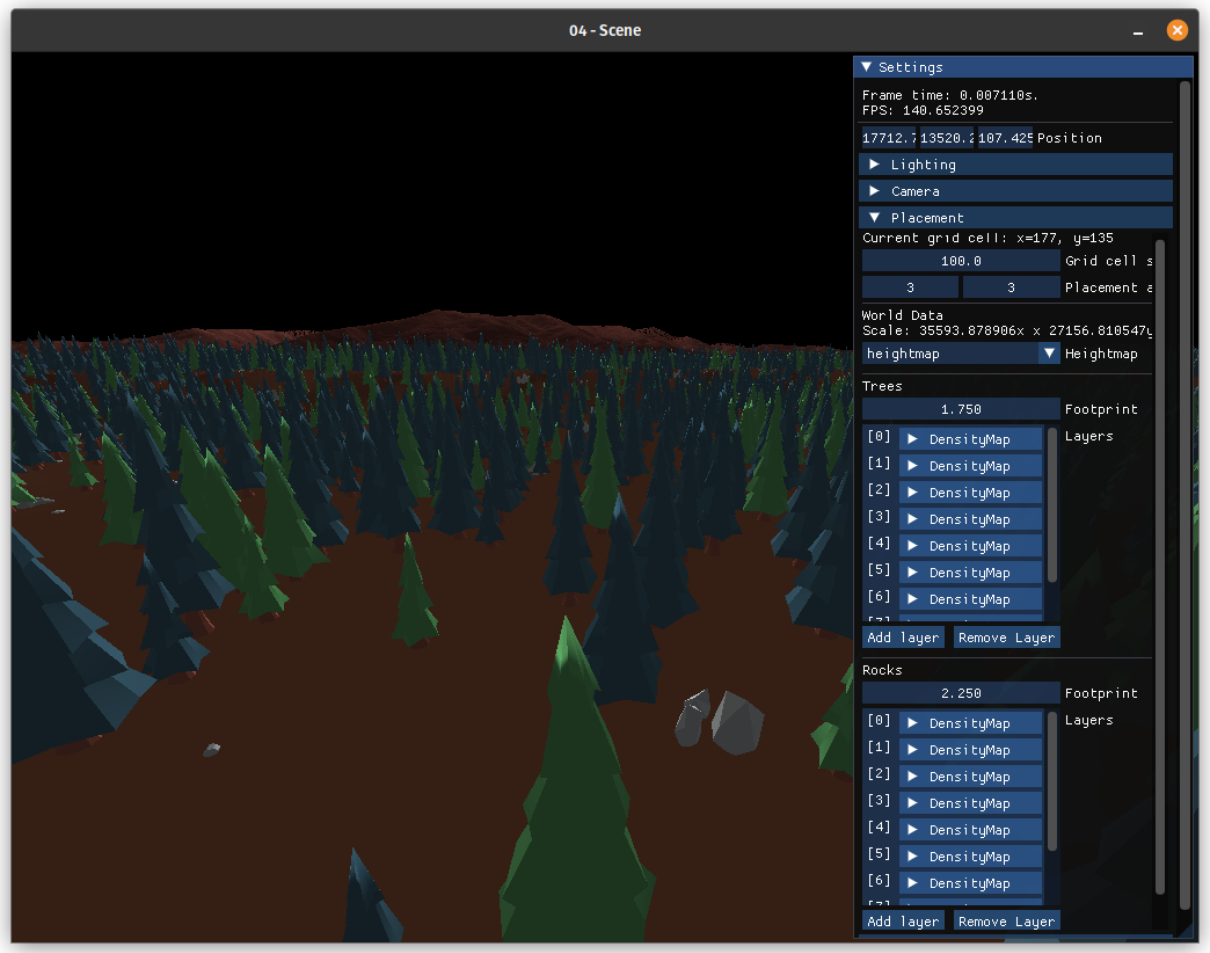
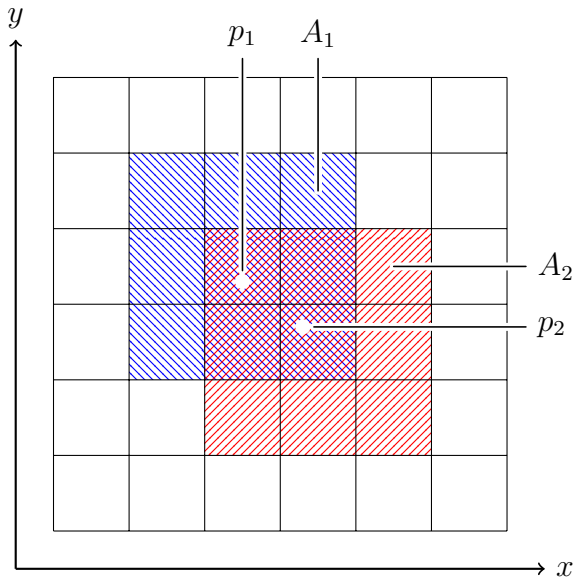


Figura 4.1: Captura de pantalla de la demo en ejecución.

En la imagen de la figura 4.1 es visible la interfaz que permite ajustar los parámetros de posicionamiento, así como algunos otros datos adicionales. En el extremo superior derecho se encuentran los indicadores de *frame time* y FPS (*frames per second*, o cuadros por segundo). El frame time se refiere a la cantidad de tiempo que transcurre entre el inicio de un cuadro y el siguiente, mientras que la tasa de cuadros es simplemente el inverso de este valor. Inmediatamente debajo de esta información se encuentra la posición actual de la cámara y luego, las entradas del menú de configuración de iluminación y cámara.

La sección etiquetada como "Placement" permite ajustar los parámetros del algoritmo de posicionamiento. Los campos "grid cell size" y "placement area" determinan el tamaño del área de posicionamiento y, por tanto, los valores para los argumentos `lower_bound` y `upper_bound` de la función `computePlacement`. La sección "world data" corresponde, naturalmente, a la estructura `WorldData` y permite cambiar el mapa de altura del terreno. Por su parte, las secciones "Trees" y "Rocks" corresponden, cada una, a una estructura `LayerData`; una para los árboles y la otra, para las rocas. Para cada una de estas es posible ajustar la

longitud de la huella (*footprint*) y la cantidad de capas que la componen, cada una de las cuales es una estructura `DensityMap`, como se describe en la sección 3.3.2.4.



p_i: posición de la cámara.

A_i: área de posicionamiento resultante.

Figura 4.2: Determinación del área de generación en la demo.

La figura 4.2 ilustra el método mediante el cuál se determina el área a generar. El espacio se divide en una grilla con celdas cuadradas de tamaño uniforme. Así, cuando el punto de foco de la cámara corresponde a la posición p_1, el área generada será A_1. En el ejemplo de la figura, ello corresponde a la celda que contiene a p_1 y sus 8 celdas vecinas. Luego, si el punto de referencia se desplaza hasta el punto p_2, saliendo de la celda en la que se encontraba anteriormente, los datos previamente generados son descartados y el posicionamiento se realiza nuevamente. Los resultados para el área de traslape entre A_1 y A_2 son idénticos en cada ejecución del algoritmo. También es posible ajustar en tiempo de ejecución tanto el tamaño de las celdas como la cantidad de celdas vecinas a considerar sin alterar esta propiedad. La imagen de la figura 4.3 muestra cómo se ve esto en la demo.

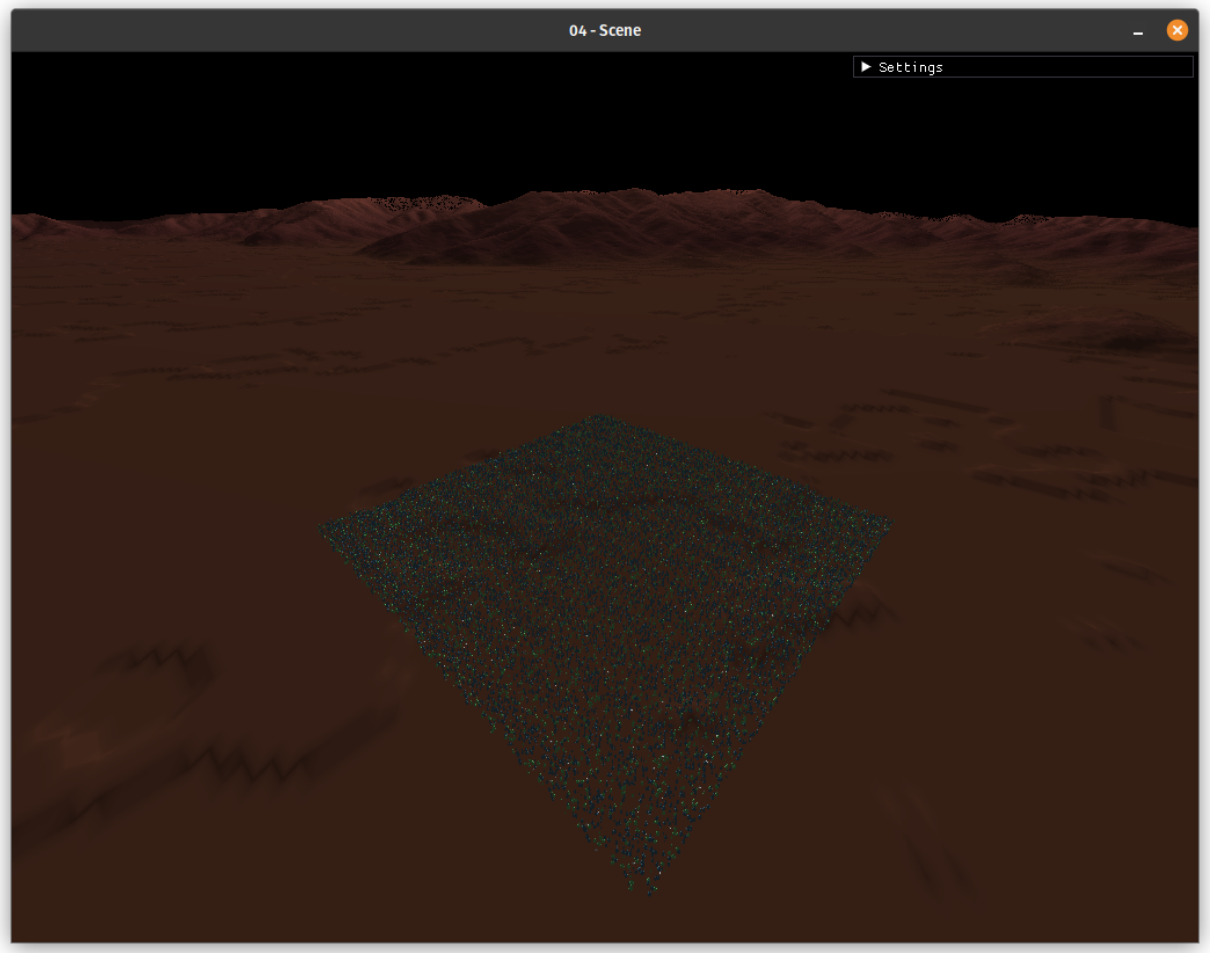
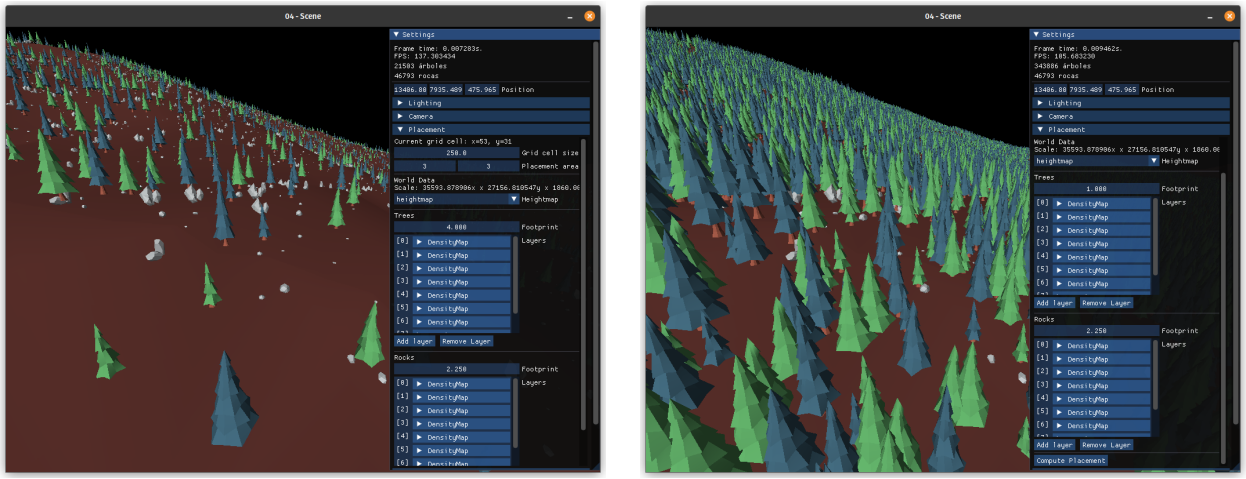


Figura 4.3: Vista general del área de posicionamiento.

Los objetos en escena se posicionan en dos grupos, es decir, ejecutando dos veces el algoritmo de posicionamiento. Uno de estos grupos posiciona los árboles y el otro, las rocas. Cada una des estas ejecuciones utiliza una de las estructuras `LayerData` mencionada anteriormente, a través de la cual se puede configurar el tamaño de la huella y la lista de mapas de densidad. También es posible cambiar el modelo utilizado para visualizar los resultados de cada capa. Por defecto, el grupo de los árboles tiene una huella de 1,75 mts. y 8 capas, mientras que para las rocas la huella es de 2,25 mts. y se utilizan 10 capas. En las imágenes adjuntas, cada modelo de árbol y roca proviene de una capa distinta. Con la configuración por defecto, se calculan posiciones para más de 20000 objetos cada vez.



(a) $h = 4$ (b) $h = 1$

Figura 4.4: Efecto de distintos diámetros de huella.

Por último, el *rendering* de la demo se realiza utilizando una biblioteca simple[11] desarrollada con anterioridad a este proyecto. El propósito de esto es verificar que la biblioteca de posicionamiento puede ser utilizada en conjunto con un motor gráfico preexistente.

4.2. Validación

En esta sección se presentan métricas de rendimiento tomadas de la demo, así como de las pruebas automatizadas de la biblioteca. Al momento de escribir este informe, todas las pruebas se completan exitosamente en la última versión de la biblioteca. Las pruebas de validación fueron ejecutadas en un computador de escritorio con un procesador AMD Ryzen 7 5700x de 8 núcleos y 16 hilos, 32GB de memoria RAM DDR4 y una GPU AMD Radeon RX 6700 XT con 12 GB de VRAM, bajo el sistema operativo Pop!_OS 22.04.[12] Las pruebas fueron realizadas compilando la biblioteca y los programas de prueba en modo *release*,[13] es decir, con la mayoría de las optimizaciones del compilador activas.

[11] Disponible en el repositorio https://github.com/zerujio/simple-renderer
[12] Una distribución de Linux derivada de Ubuntu, la cual a su vez desciende de Debian.
[13] Esta es una de las configuraciones predeterminadas de CMake, el sistema de build utilizado en el proyecto. Esta configuración está diseñada para ser lo más rápida posible.

4.2.1. Métricas de rendimiento generales

La tabla 4.1 corresponde a un registro de los tiempos por cuadro de la demo para un periodo de aproximadamente 30 segundos. El tiempo entre cuadros (denotado ΔT) es el tiempo transcurrido desde el comienzo de un cuadro hasta el comienzo del siguiente. El inverso de este valor es la tasa de cuadros del programa. El tiempo en CPU, por otro lado, es la parte del tiempo entre cuadros ocupado por tareas que se ejecutan exclusivamente en ésta. Específicamente, es el tiempo que transcurre entre el inicio del procesamiento de cada cuadro y el final de este, justo antes de llamar la función de intercambio de framebuffer.[14] Luego, la diferencia entre el tiempo total entre cuadros y el tiempo en CPU corresponde, aproximadamente, al tiempo ocupado por la GPU para terminar de realizar todas las operaciones necesarias para dibujar el cuadro. Estos tiempos fueron medidos con la sincronización vertical desactivada, lo que significa que el programa no espera a que el monitor se actualice para dibujar un nuevo cuadro, si no que se ejecuta tan rápido como sea posible.

Tabla 4.1: Resumen del tiempo por cuadro en la demo.

Métrica	ΔT [ms]	Sólo CPU [ms]
Promedio	1.16	0.13
Desviación estándar	1.08	0.31
Percentil 90	1.18	0.15
Percentil 99	1.47	0.21
Percentil 99.9	5.49	2.55

La figura 4.5 grafica estas métricas para una serie de 100 cuadros consecutivos tomados del mismo conjunto de datos. Las líneas verticales continuas corresponden al momento en que se despachan los compute shaders, mientras que las líneas verticales punteadas indican el momento en que se terminan de leer los resultados del posicionamiento. Es decir, estas líneas marcan los instantes en que se vuelven a generar las posiciones para todos los árboles y rocas en escena. De forma congruente con lo visto en la tabla 4.1, el ΔT para la mayoría de los cuadros se encuentra entre 1 y 1,5 milisegundos. Cuando se calculan las posiciones, este valor se eleva hasta los 2,5 milisegundos. La mayor parte de este tiempo proviene del tiempo ocupado en CPU para leer los resultados.

[14] El framebuffer es, en términos simples, una imagen que representa el contenido de una ventana del sistema operativo, o de la pantalla completa. Intercambiar el framebuffer signifca cambiar la imagen actual por la siguiente, es decir, avanzar un cuadro.

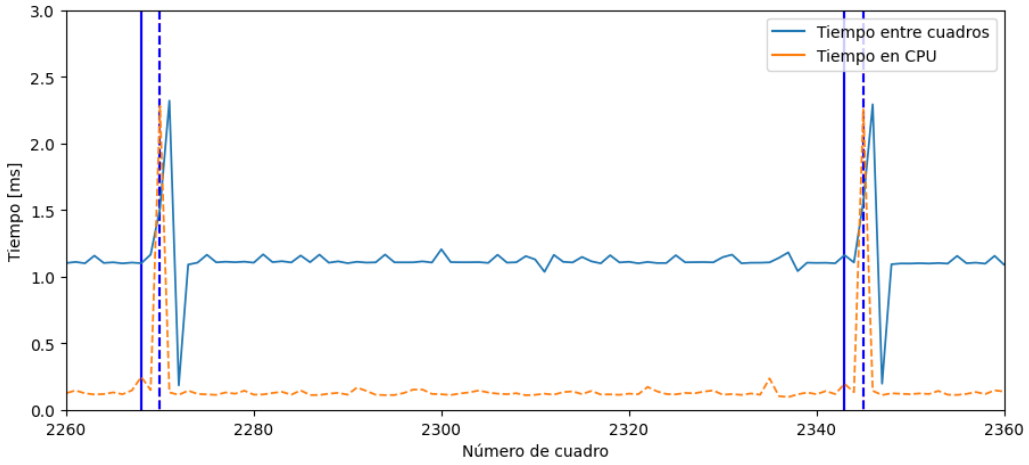


Figura 4.5: Gráfico del tiempo por cuadro, para 100 cuadros.

Tabla 4.2: Sumario del tiempo de ejecución de las pruebas de rendimiento automatizadas.

Área [m^2]	Algoritmo	Promedio [μs]	σ [μs]
100	Lanzamiento de dardos	1339.1	3.9
	Posicionamiento en GPU	237.1	211.4
	CPU, referencia secuencial	77.2	0.8
	CPU, referencia en paralelo	59.2	3.7
500	Lanzamiento de dardos	25800.4	33.9
	Posicionamiento en GPU	3403.5	244.7
	CPU, referencia secuencial	2193.8	6.3
	CPU, referencia en paralelo	1164.7	86.8
1000	Lanzamiento de dardos	92305.5	732.7
	Posicionamiento en GPU	14935.6	196.3
	CPU, referencia secuencial	9239.4	16.2
	CPU, referencia en paralelo	4166.6	308.3

La tabla 4.2 presenta los resultados de las pruebas automatizadas de la biblioteca. Estas pruebas miden el tiempo que toma en completarse el con un área dada y huella dada para distintos algoritmos. En el caso del área más grande, se calculan las posiciones para casi 50000 elementos. El tiempo mostrado es el promedio de varias ejecuciones de cada algoritmo.

El algoritmo etiquetado “posicionamiento en GPU” corresponde a la implementación de la biblioteca. Por otra parte, el algoritmo de lanzamiento de dardos es el descrito en el fragmento de código 3.2, incluyendo la optimización necesaria para comprobar colisiones en tiempo constante. Este es el principal punto de comparación en términos de rendimiento, aunque no

sea equivalente en funcionalidad[15]. Los otros dos métodos son implementaciones en CPU del algoritmo de posicionamiento, basadas en las funciones de referencia usadas para validar los resultados de la biblioteca. Su propósito principal es proveer un punto de referencia con el que estimar el posible sobrecosto del algoritmo en GPU. La versión secuencial genera y evalúa los puntos uno por uno, para luego ordenarlos utilizando la función `sort` de la biblioteca estándar de C++. La versión paralela es exactamente lo mismo, pero utilizando la política de ejecución[16] paralela en las funciones `foreach` y `sort`.

En todos los casos el algoritmo de posicionamiento en GPU obtuvo mejores métricas que el algoritmo de lanzamiento de dardos, siendo entre 5 y 7 veces más rápido. Sin embargo, la implementación de referencia en CPU aparenta ser más rápida. El posicionamiento en GPU es aproximadamente 50 % más lenta que la versión en CPU secuencial, y 3 veces más lenta que la versión paralelizada en CPU.

4.2.2. Análisis de los resultados

Para poner en perspectiva los valores vistos anteriormente, se debe considerar que la tasa de refresco típica para un monitor de oficina o televisor es de 60 Hz, lo que equivale a un tiempo entre cuadros de 16,6 ms. Por contraparte, los monitores para juegos, así como algunos televisores, tienen tasas de refresco entre 120 y 165 Hz y, por tanto, un ΔT en torno a los 7 ms.

En comparación con estos valores, el incremento de 1 ms en el ΔT observado en la figura 4.5 tiene una magnitud lo suficientemente pequeña como para no tener un impacto perceptible en la fluidez de la aplicación. Si bien esto es suficiente para considerar que la implementación del algoritmo de posicionamiento cumple con los objetivos de rendimiento, lo ideal sería que esta anomalía no estuviera presente. Así mismo, también se debe señalar que los resultados de las pruebas automatizadas no son tan positivos como se esperaba. En particular, lo ideal habría sido que el algoritmo de posicionamiento sobrepasara en velocidad a la implementación de referencia secuencial. En vista de esto, se han identificado tres puntos en los que la implementación del algoritmo podría tener problemas.

En primer lugar está la presencia de puntos de sincronización CPU-GPU. En términos sim-

[15] El algoritmo de lanzamiento de dardos genera una distribución pseudo-aleatoria mejor, pues no depende de la repetición de un patrón.

[16] Las "políticas de ejecución" son argumentos que permiten ajustar la forma en que se ejecutan algunos algoritmos de librería estándar. En la práctica, existen dos políticas: secuencial y paralela. La secuencial es la predeterminada, mientras que la paralela debe ser especificada explícitamente y causa se utilicen todos los núcleos del procesador de forma automática.

ples, estos puntos de sincronización se producen porque los comandos enviados a la GPU desde la CPU se almacenenan en una cola, para luego ser ejecutados de forma asíncrona. Es esperable que la arquitectura de OpenGL (y, en general, de cualquier API gráfica de bajo nivel) esté optimizada para transferencias de información desde la CPU hacia la GPU, pues este es el flujo de información usual en una aplicación gráfica. Realizar una operación en el sentido contrario requiere, en el peor de los casos, dejar de añadir nuevos comandos, esperar a que se completen aquellos actualmente en ejecución, y sólo entonces acceder a los datos. La mayoría de las operaciones de transferencia de datos desde la GPU a la CPU requiere de una sincronización de este tipo [20].

De acuerdo con lo observado en la figura 4.5, la mayor elongación del tiempo entre cuadros se produce en el cuadro en que se accede a los resultados del posicionamiento desde la CPU, evento que está marcado con una línea punteada. El hecho de que todo ese tiempo adicional esté contabilizado dentro del "Tiempo en CPU" sugiere la presencia de un punto de sincronización. Si se considera que la cantidad de información leída desde la GPU está en el orden de los cientos de bytes, y que la magnitud de la disrupción es similar a la del ΔT promedio, es razonable suponer que esta operación está causando que la CPU espere a que se completen todos (o gran parte de) los comandos en cola.

Una segunda posibilidad es que el volumen de datos siendo procesados no sea el necesario para compensar el costo asociado a OpenGL. Si bien la GPU escala mejor que la CPU a medida que el tamaño de la carga de trabajo aumenta, el costo fijo es bastante elevado. Si la cantidad de elementos evaluados es muy pequeña, el sobrecosto de OpenGL puede acabar por ser mayor al tiempo ganado gracias al paralelismo. Es por esto que en la tabla 4.2, en la prueba de 100 m^2, el algoritmo de referencia secuencial es 3 veces más rápido que en GPU, mientras que con áreas de 500 y 1000 m^2 la diferencia se reduce a un 50 %. Podría ser, entonces, que se requiera de un área aún mayor para obtener un resultado positivo.

Finalmente, también hay que considerar que algunos de los otros detalles de implementación de la biblioteca pueden estar afectando negativamente el rendimiento, tanto en el código de los compute shaders como a nivel de manejo de recursos de OpenGL. En esta categoría se incluye, por ejemplo, las múltiples posibles formas de asignar y utilizar buffers, así como los métodos que utilizan compute shaders para acceder a éstos. En particular, hay un número de puntos conocidos que podrían afectar negativamente el rendimiento, pero cuyo impacto no ha sido evaluado de forma individual. Algunos de estos son, por ejemplo, que las primeras dos etapas del algoritmo reutilizan un mismo buffer entre distintas invocaciones (de compute shaders), pero podría ser más eficiente crear un buffer nuevo para cada invocación, o utilizar parámetros distintos en la creación del buffer. Así mismo, es posible modificar el compute shader de la fase de evaluación para procesar todos los mapas de densidad en un loop en lugar de requerir múltiples invocaciones. En cualquier caso, estas conjeturas deben ser puestas a

prueba antes de concluir que efectivamente son la causa de los resultados observados.

4.2.3. Métricas de rendimiento mejoradas

Para determinar cuáles de las explicaciones anteriormente propuestas tienen un impacto efectivo en el rendimiento de la biblioteca, fue necesario obtener mejor información de rendimiento. Así, la demo se modificó para entregar información más detallada acerca de las tareas realizadas. Los gráficos de esta sección muestran esta información sobre una línea temporal, donde cada uno de los períodos de tiempo medidos aparece como una banda de color. En primer lugar, aquellas etiquetadas como "generación" corresponden al período entre que se da inicio al algoritmo de posicionamiento y que se obtiene la notificación de que éste ha terminado. Así mismo, la "lectura de resultados" indica el tiempo requerido para obtener las posiciones generadas, mientras que la "actualización de geometría" es la actualización de los buffers necesarios para visualizar los resultados. El "total por cuadro" es, simplemente, el tiempo total de trabajo en CPU para cada cuadro. Los espacios en blanco que le siguen a cada línea anaranjada corresponden, por tanto, al tiempo que tarda la GPU en completar y mostrar el cuadro actual. Los gráficos tienen, además dos "carriles",

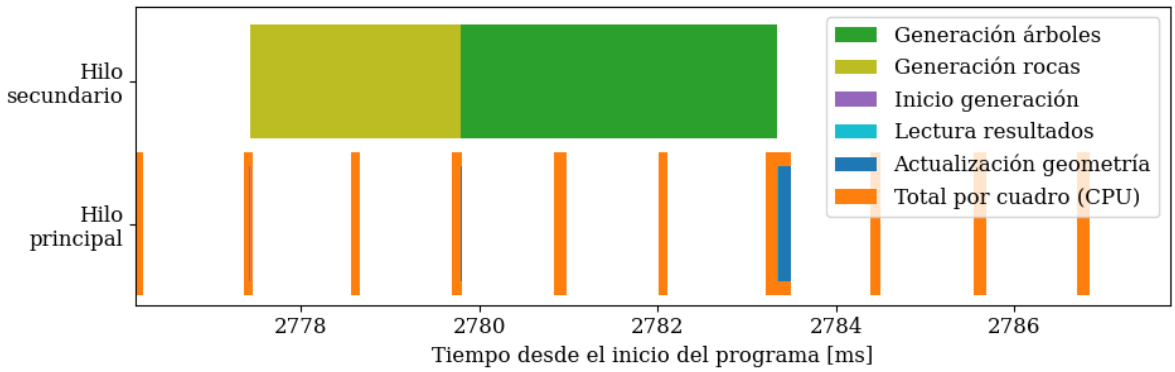


Figura 4.6: Captura del rendimiento de la demo (posicionamiento en CPU, monohilo).

La figura 4.6 muestra el detalle de una operación de posicionamiento, tomada de una ejecución de la versión en CPU de la demo. Esta versión utiliza la implementación en CPU del algoritmo de posicionamiento, lo permite obtener un punto de referencia más preciso y descartar posibles errores de implementación de la demo. Como se puede observar, la operación de posicionamiento no causa una alteración significativa del tiempo que toma cada cuadro en CPU. La única variación significativa se produce al actualizar la geometría, donde la información debe ser transferida a la GPU.

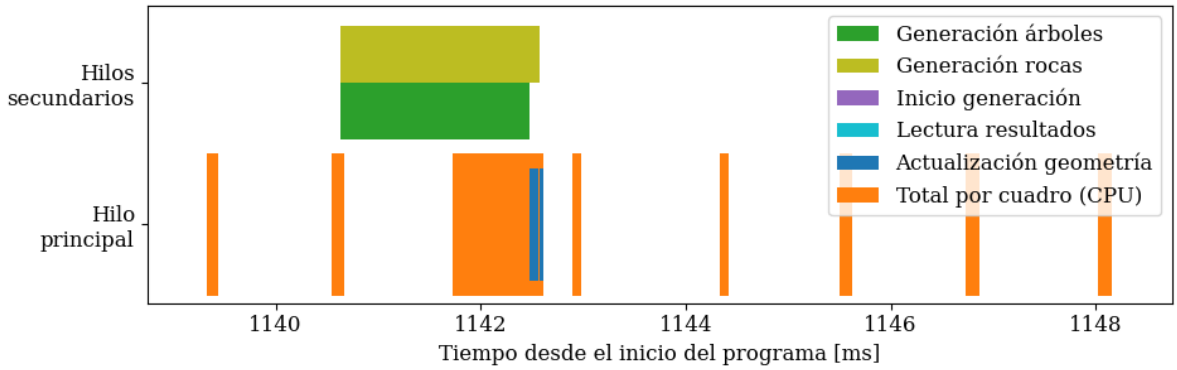


Figura 4.7: Captura del rendimiento de la demo (posicionamiento en CPU, multihilo).

De manera análoga, la figura 4.7 muestra el rendimiento de la versión en CPU del algoritmo, pero permitiendo que se ejecute en múltiples hilos. Como es de esperar, el tiempo de ejecución es mucho menor que en el caso monohilo. Se puede observar también que, en el caso específico mostrado, el cuadro siguiente a aquel en que comienza la operación de posicionamiento toma significativamente más de lo normal, y que ni la lectura de los resultados ni la actualización de la geometría son la causa de ello. Este comportamiento se debe, con seguridad, a que el algoritmo de posicionamiento está saturando la capacidad de cómputo de la CPU y, por tanto, las operaciones en el hilo principal se ejecutan mas lentamente. Si bien este problema no se presenta siempre y puede ser solucionado fácilmente reduciendo la cantidad de hilos que se utilizan para el posicionamiento, también hace notorio el hecho de que el algoritmo ejecutado en CPU hace uso intensivo de un recurso limitado. Esto significa que en una aplicación más compleja, que haga uso mas intenso de la CPU, ejecutar el posicionamiento de este modo tendrá un impacto más significativo en el rendimiento, tanto del algoritmo como de la aplicación en su totalidad.

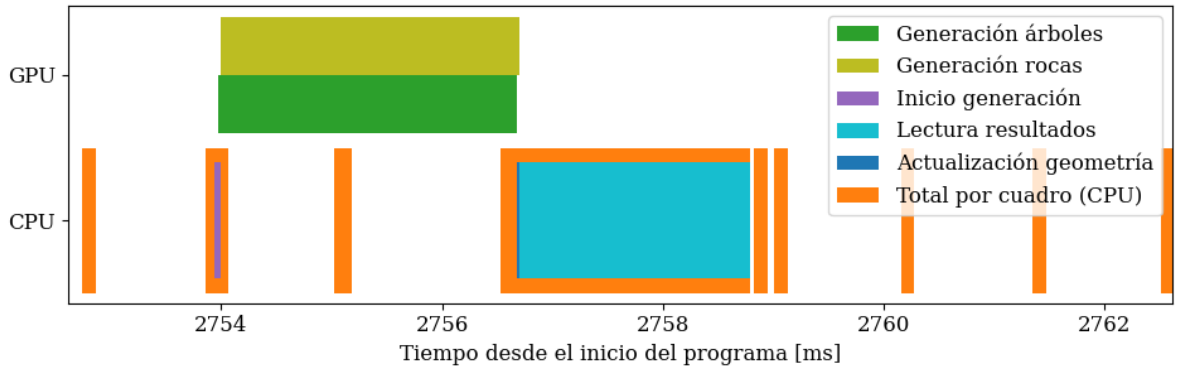


Figura 4.8: Captura del rendimiento de la demo (posicionamiento en GPU).

Lo anterior contrasta con lo que se observa en la figura 4.5, donde al momento de realizar el posicionamiento se presenta un gran aumento en el tiempo que toma cada cuadro en CPU, específicamente en la fase de lectura de resultados. Esto confirma la sospecha acerca de la presencia de una sincronización CPU-GPU, y el hecho de que las versiones en CPU no exhiban este comportamiento también lo corrobora, pues indica que el problema es específico al uso de la GPU. En resumen, la causa más probable es que la operación de lectura causa que OpenGL detenga el programa hasta que todas las operaciones en GPU se hayan completado, con tal de asegurar la correctitud de los datos leidos desde el buffer de resultados.

4.2.4. Optimización de rendimiento

Teniendo en cuenta las observaciones anteriores, modificó la forma en que se leen los resultados. El método inicial consistía en utilizar la función `glBufferSubData` para copiar los contenidos del buffer al espacio de memoria de la CPU. Esta función es simple de usar pero, si el buffer al que se está accediendo será modificado por alguna operación en cola, debe esperar a que se complete antes de retornar. Si bien la biblioteca utiliza una cerca de sincronización[17] para determinar cuándo las operaciones de posicionamiento se han completado, es probable que la implementación de `glBufferSubData` no tome en cuenta esta información. La especificación de esta función no impone requerimientos en ese sentido, por lo que puede causar una sincronización CPU-GPU incluso cuando no es estrictamente necesaria.

El problema puede ser solucionado accediendo al buffer mediante un mapeo persistente. En OpenGL, mapear un buffer es una operación que mapea total o parcialmente el rango de memoria asignado al buffer en la GPU a un rango equivalente en el espacio de memoria de la CPU, lo que permite a la aplicación acceder a los contenidos del buffer a través de un puntero, como si se tratara de memoria normal. Normalmente, no es posible utilizar un buffer para otras operaciones mientras está mapeado, lo que asegura la coherencia de los datos y evita *data races*[18]. A partir de la versión 4.4 OpenGL es posible crear buffers especiales que pueden ser mapeados de manera persistente, lo que significa que pueden ser utilizados para otras operaciones mientras se encuentran mapeados. Esto permite evitar multiples puntos de sincronización CPU-GPU, pues ya no es necesario realizar un operación de mapeo (con su consiguiente sincronización) cada vez que se quiera acceder a los datos, sino que se puede realizar una única operación de mapeo después de la creación del buffer y utilizar siempre el mismo puntero. Sin embargo, al evitar los puntos de sincronización también se hacen posibles las data races, por lo que es necesario utilizar mecanismos de sincronización explícitos, como

[17] *Fence sync* en inglés. Objeto de OpenGL análogo a un semáforo o mutex con condición.

[18] Una data race ("carrera de datos") es una situación en que dos o más hilos de ejecución modifican el mismo sector de memoria de manera concurrente, generando resultados posiblemente incorrectos o incoherentes. En este caso, ocurre cuando la GPU lee de un sector que la CPU está escribiendo y vice versa.

las cercas de sincronización anteriormente mencionados.

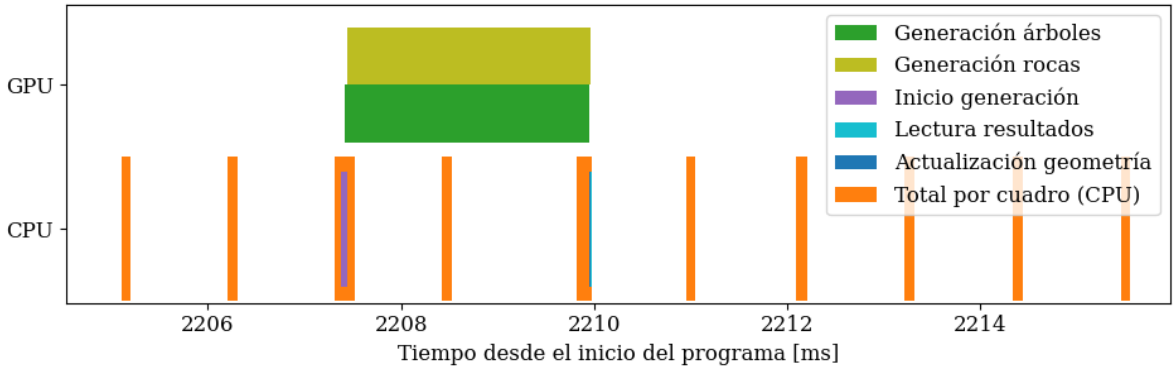


Figura 4.9: Captura del rendimiento de la demo optimizada (posicionamiento en GPU).

Al convertir el buffer de resultados en un buffer mapeado persistentemente se obtiene un rendimiento muy superior al visto anteriormente. Además, al realizar las modificaciones se identificó que uno de los detalles de implementación mencionados en la sección 4.2.2 tiene un impacto en el rendimiento. Específicamente, al utilizar el mismo buffer para distantas invocaciones del algoritmo se previene que estas se ejecuten en paralelo, pues cada una `computePlacement()` debe esperar a que la anterior termine antes de poder ocupar el buffer. Este problema se resolvió creando un buffer nuevo para cada operación de posicionamiento.

Con estas modificaciones, la implementación en GPU pasa a ser considerablemente más rápida que la versión en CPU, aún cuando esta se ejecuta en múltiples hilos. La tabla 4.3 muestra el resultado de las pruebas de rendimiento para la implementación optimizada y los compara con los resultados previamente presentados en la tabla 4.2. La columna derecha muestra la razón entre el tiempo promedio de cada método y el tiempo promedio de la versión en GPU optimizada. Este valor, denominado *speedup* o aceleración, indica cuánto más rápido es este último método en relación a los demás. Por ejemplo, cuando el área de posicionamiento es de 1000 m^2, la versión en GPU es 4 veces más rápida que la versión paralela en CPU, 90 veces más rápida que el algoritmo de lanzamiento de dardos, y 15 veces más rápida que la antigua implementación.

Finalmente, nótese que el speedup aumenta a medida que el tamaño de la carga de trabajo crece, como habría de esperar de acuerdo a los discutido en la sección 4.2.2. Así, los datos de la tabla permiten determinar que el área para la cual la aceleración producto del paralelismo compensa el costo fijo del algoritmo se encuentra entre los 100 y 500 m^2, y que, por tanto, la explicación que aludía al sobrecosto de OpenGL era incorrecta.

Tabla 4.3: Resultados de las pruebas de rendimiento, con optimizaciones de rendimiento aplicadas.

Área $[m^2]$	Algoritmo	Promedio $[\mu s]$	σ $[\mu s]$	T/T_{GPU}
100	Lanzamiento de dardos	1339.1	3.9	6.5
	Posicionamiento en GPU	237.1	211.4	1.2
	CPU, referencia secuencial	77.2	0.8	0.4
	CPU, referencia en paralelo	59.2	3.7	0.3
	GPU, optimizado	204.8	166.8	
500	Lanzamiento de dardos	25800.4	33.9	64.1
	Posicionamiento en GPU	3403.5	244.7	8.5
	CPU, referencia secuencial	2193.8	6.3	5.5
	CPU, referencia en paralelo	1164.7	86.8	2.9
	GPU, optimizado	402.8	243.2	
1000	Lanzamiento de dardos	92305.5	732.7	93.3
	Posicionamiento en GPU	14935.6	196.3	15.1
	CPU, referencia secuencial	9239.4	16.2	9.33
	CPU, referencia en paralelo	4166.6	308.3	4.2
	GPU, optimizado	989.3	352.9	

Capítulo 5

Conclusiones

El presente trabajo está motivado por el deseo de poner a disposición de desarrolladores de videojuegos de toda índole una nueva herramienta para la creación de mundos virtuales extensos y variados. Tomando inspiración en el trabajo de Guerrilla Games para Horizon: Zero Dawn, se diseñó e implementó un algoritmo capaz de hacer uso de las capacidades de cómputo masivamente paralelo de las GPUs modernas. Esta técnica es capaz de poblar el terreno de un mundo virtual con objetos decorativos de toda índole, y hacerlo lo suficientemente rápido como para ejecutarse en el contexto de una aplicación en tiempo real. Adicionalmente, los resultados son consistentes, fácilmente reproducibles y pueden ser calculados de forma local. Esto, en conjunto con la velocidad del algoritmo, hace innecesario precalcular posiciones para una amplia variedad de elementos y, por tanto, permite reducir el tamaño de los archivos de datos necesarios en un videojuego de mundo abierto. Estas mismas características también reducen significativamente la cantidad de trabajo humano requerido para decorar un paisaje virtual de gran extensión.

El trabajo de desarrollo realizado dio lugar a una biblioteca de código abierto que permite utilizar este algoritmo. Esta implementación cumple con los requisitos de velocidad y determinismo necesarios para ser utilizada del modo descrito anteriormente. Está desarrollada con tecnologías ampliamente utilizadas en la industria de los videojuegos, C++ y OpenGL, con tal de facilitar su uso en proyectos en dicho ámbito. Incluso en proyectos basados en otras tecnologías la biblioteca puede servir de modelo para el desarrollo de una solución *ad hoc*, o bien, puede ser adaptada (e.g. para usar Vulkan en lugar de OpenGL).

Si bien la biblioteca está inspirada por las tendencias de diseño en el mundo de los videojuegos, es adaptable a otros ámbitos de la computación gráfica. De hecho, el relieve del

terreno utilizado para los programas de ejemplo de la biblioteca corresponde a una ubicación real en Chile. Esto sugiere la posibilidad de utilizar el algoritmo para visualizar o modelar vastas extensiones terreno en aplicaciones de distinta índole como, por ejemplo, científica o educativa.

5.1. Trabajo futuro

En primer lugar, existen algunos puntos acerca de la implementación actual que podrían ser mejorados, así como características que habría sido deseable incluir en la biblioteca, pero que excedían el alcance de este trabajo. Algunos de estos son:

- Compatibilidad con más APIs gráficas. En particular, Vulkan es más moderna, naturalmente asíncrona y multihilo, manteniendo con un nivel comparable de compatibilidad multiplataforma.
- Atributos adicionales para los elementos generados como, por ejemplo, rotación e inclinación.
- Parámetros adicionales para el mundo, tales que puedan ser utilizados para cambiar la forma en que los objetos se distribuyen, como la pendiente del terreno o la presencia de cauces de agua y caminos.
- El sistema de *ecotopos* utilizado en Horizon: Zero Dawn. Esto es, una estructura de datos jerárquica que permite componer distintos mapas de densidad para generar distribuciones complejas con menos esfuerzo.
- Aceleración en GPU para el cálculo de colisiones entre objetos con distinto tamaño de huella. En la forma actual del algoritmo sólo se puede garantizar la ausencia de colisiones entre objetos con la misma huella, por lo que la resolución de conflictos entre elementos de distinto tamaño queda a cargo del usuario de la biblioteca.

Finalmente, utilizando la biblioteca, es posible construir un *plugin* para un motor gráfico o de videojuegos existente, para permitir su uso con una interfaz más propias de éste. En el caso de motores que no sean compatibles con OpenGL, o que no brinden acceso fácil al contexto, también es posible desarrollar una adaptación de la biblioteca utilizando las herramientas del motor. Por ejemplo, la recientemente estrenada versión 4 del motor Godot utiliza Vulkan en lugar de OpenGL, pero también posee herramientas para trabajar con compute shaders en GLSL, lo que permitiría fácilmente transferir los compute shaders desarrollados.

Bibliografía

[1] Togelius, J., Kastbjerg, E., Schedl, D., y Yannakakis, G. N., "What is procedural content generation? Mario on the borderline," en Proceedings of the 2nd International Workshop on Procedural Content Generation in Games, PCGames '11, (New York, NY, USA), Association for Computing Machinery, 2011, doi:10.1145/2000919.2000922.

[2] Togelius, J., Shaker, N., y Nelson, M. J., Introduction, pp. 1–15. Cham: Springer International Publishing, 2016, doi:10.1007/978-3-319-42716-4_1.

[3] Corbyn, Z. y Morris, B., "Nvidia: the chipmaker that became an AI superpower," BBC News, 2023, https://www.bbc.com/news/business-65675027.

[4] Navarro, C., Hitschfeld, N., y Mateu, L., "A survey on parallel computing and its applications in data-parallel problems using GPU architectures," Communications in Computational Physics, vol. 15, pp. 285–329, 2013, doi:10.4208/cicp.110113.010813a.

[5] Freiknecht, J. y Effelsberg, W., "A survey on the procedural generation of virtual worlds," Multimodal Technologies and Interaction, vol. 1, no. 4, 2017, doi:10.3390/mti1040027.

[6] Perlin, K., "Improving noise," ACM Trans. Graph., vol. 21, p. 681–682, 2002, doi:10.1145/566654.566636.

[7] Smelik, R. M., Tutenel, T., Bidarra, R., y Benes, B., "A survey on procedural modelling for virtual worlds," Computer Graphics Forum, vol. 33, no. 6, pp. 31–50, 2014, doi:https://doi.org/10.1111/cgf.12276.

[8] Berger, U., Hildenbrandt, H., y Grimm, V., "Towards a standard for the individual-based modeling of plant populations: Self-thinning and the field-of-neigborhood approach," Natural Resource Modeling, vol. 15, no. 1, pp. 39–54, 2002, doi:https://doi.org/10.1111/j.1939-7445.2002.tb00079.x.

[9] Deussen, O., Hanrahan, P., Lintermann, B., Měch, R., Pharr, M., y Prusinkiewicz, P., "Realistic modeling and rendering of plant ecosystems," en Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '98, (New York, NY, USA), p. 275–286, Association for Computing Machinery, 1998, doi:10.1145/280814.280898.

[10] do Nascimento, B. T., Franzin, F. P., y Pozzer, C. T., "GPU-based real-time procedural distribution of vegetation on large-scale virtual terrains," en 2018 17th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames), pp. 157–15709, IEEE, 2018.

[11] Henríquez, H. y Akleman, E., "Web-based and interactive 3D pictorial maps with data-driven procedural placement." Disponible en línea: https://hernaldo.me/assets/files/pictorial_map.pdf, 2021.

[12] van Muijden, J., "GPU-based run-time procedural placement in 'Horizon: Zero Dawn'." GDC Vault, 2017, https://www.gdcvault.com/play/1024120/GPU-Based-Run-Time-Procedural.

[13] "What are the best game engines?." Perforce Blog, 2023, https://www.perforce.com/blog/vcs/most-popular-game-engines.

[14] "Procedural foliage tool." Documentación de Unreal Engine 5.0, 2022, https://docs.unrealengine.com/5.0/en-US/procedural-foliage-tool-in-unreal-engine/.

[15] "¡Unreal Engine 5.2 ya está disponible!," 2023, https://www.unrealengine.com/es-ES/blog/unreal-engine-5-2-is-now-available.

[16] "Procedural geometry." Documentación de Godot Engine 4.0, 2023, https://docs.godotengine.org/en/4.0/tutorials/3d/procedural_geometry/index.html.

[17] "ProtonGraph." Repositorio Git, 2023, https://github.com/protongraph/protongraph.

[18] "PCG secrets: The art of procedural generation in Godot." Curso en línea, https://gdquest.mavenseed.com/courses/godot-pcg-secrets.

[19] Cox, E., "Procedural world map generator and viewer component." Repositorio de Git, https://github.com/edwin-cox/godot-infinite-worldmap.

[20] "Synchronization." OpenGL Wiki, 2021, http://www.khronos.org/opengl/wiki_opengl/index.php?title=Synchronization&oldid=14848.

Anexo

En este anexo se encuentra el código en GLSL utilizado para implementar las cuatro etapas descritas en la sección 3.2.

Código .1: Kernel de generación.

```
#version 450 core

#define INVALID_INDEX 0xFFffFFff

// Esta declaración define el tamaño de los work groups. En este caso se tiene una matriz
// cuadrada 8 elementos por eje.
layout(local_size_x = 8, local_size_y = 8) in;

// La huella de los objetos, que corresponde a la separación mínima.
uniform float u_footprint;

// Las dimensiones totales del mundo.
uniform vec3 u_world_scale;

// El espacio que cada celda de posicionamiento ocupa, especificado en el mismo
// espacio de coordenadas que u_world_scale. Cada celda corresponde a un work group.
uniform vec2 u_work_group_scale;

// El índice de la celda en la esquina inferior del área de posicionamiento.
uniform uvec2 u_work_group_offset;

// El patrón de posicionamiento.
uniform vec2 u_work_group_pattern[gl_WorkGroupSize.x][gl_WorkGroupSize.y];

// La textura utilizada como mapa de altura para el terreno.
uniform sampler2D u_heightmap;

// Las propiedades principales de una posición candidata.
```

```
struct Candidate
{
    // La posición del candidato, en el mismo espacio de coordenadas que u_world_scale.
    vec3 position;

    // Indica si es un punto válido, y a qué mapa de densidad pertenece.
    uint class_index;
};

// Arreglo que almacenará todos los candidatos generados.
layout(std430) restrict writeonly
buffer CandidateBuffer
{
    Candidate[gl_WorkGroupSize.x][gl_WorkGroupSize.y] candidate_array[];
};

// Arreglo con las coordenadas globales normalizadas de cada candidato.
layout(std430) restrict writeonly
buffer WorldUVBuffer
{
    vec2[gl_WorkGroupSize.x][gl_WorkGroupSize.y] world_uv_array[];
};

// Arreglo con la densidad acumulada de cada candidato (utilizada en la próxima etapa).
layout(std430) restrict writeonly
buffer DensityBuffer
{
    float[gl_WorkGroupSize.x][gl_WorkGroupSize.y] density_array[];
};

void main()
{
    const uint array_index = gl_WorkGroupID.y * gl_NumWorkGroups.x + ↩
     ↪ gl_WorkGroupID.x;

    // El índice global de la celda que corresponde al work group actual.
    const uvec2 grid_index = gl_WorkGroupID.xy + u_work_group_offset;

    // Determinación de la posición para el punto actual, en el plano horizontal.
    const vec2 h_position =
          u_footprint *
          (u_work_group_pattern[gl_LocalInvocationID.x][gl_LocalInvocationID.y]
           + grid_index * u_work_group_scale);

    // Cálculo de las coordenadas globales normalizadas
```

```
    const vec2 world_uv = h_position / u_world_scale.xy;
    world_uv_array[array_index][gl_LocalInvocationID.x][gl_LocalInvocationID.y] = world_uv;

    // Muestreo del mapa de altura para determinar la altura del punto generado.
    const float height = texture(u_heightmap, world_uv).x * u_world_scale.z;

    // Se asigna la posición del candidato. Se inicializa como un candidato inválido.
    candidate_array[array_index][gl_LocalInvocationID.x][gl_LocalInvocationID.y] = Candidate(vec3(h_position, height), INVALID_INDEX);

    // La densidad acumulada se inicializa en 0.
    density_array[array_index][gl_LocalInvocationID.x][gl_LocalInvocationID.y] = 0.0f;
}

```

Código .2: Kernel de evaluación.

```
#version 450 core

layout(local_size_x = 8, local_size_y = 8) in;

// El mapa de densidad a utilizar en esta iteración.
uniform sampler2D u_density_map;

// Parámetros de ajuste para el muestreo del mapa de densidad.
uniform vec4 u_density_map_params;

// El índice del mapa de densidad siendo evaluado.
uniform uint u_class_index;

// La matriz de dithering.
uniform float u_dithering_matrix [gl_WorkGroupSize.x][gl_WorkGroupSize.y];

// Límites del área de posicionamiento.
uniform vec2 u_lower_bound;
uniform vec2 u_upper_bound;

uniform uvec2 u_work_group_index_offset;

struct Candidate {
    vec3 position;
    uint class_index;
};

```

```
layout(std430) restrict
buffer CandidateBuffer
{
    Candidate[gl_WorkGroupSize.x][gl_WorkGroupSize.y] candidate_array[];
};

layout(std430) restrict readonly
buffer WorldUVBuffer
{
    vec2[gl_WorkGroupSize.x][gl_WorkGroupSize.y] world_uv_array[];
};

layout(std430) restrict
buffer DensityBuffer
{
    float[gl_WorkGroupSize.x][gl_WorkGroupSize.y] density_array[];
};

float sampleDensityMap(vec2 world_uv)
{
    const float scale = u_density_map_params.x;
    const float offset = u_density_map_params.y;
    const float min_value = u_density_map_params.z;
    const float max_value = u_density_map_params.w;

    const float density = texture(u_density_map, world_uv).x;

    return clamp(density * scale + offset, min_value, max_value);
}

void main()
{
    const uint array_index = gl_WorkGroupID.y * gl_NumWorkGroups.x + ↩
     ↪ gl_WorkGroupID.x;

    const vec2 world_uv = world_uv_array[array_index][gl_LocalInvocationID.x][↩
     ↪ gl_LocalInvocationID.y];

    // El índice de la matriz de dithering que le corresponde a este candidato.
    // Se aplica un desplazamiento para evitar la repetición del mismo patrón en
    // celdas adyacentes.
    const uvec2 threshold_matrix_index =
        (gl_LocalInvocationID.xy + u_work_group_index_offset + gl_WorkGroupID.xy)
        % gl_WorkGroupSize.xy;

```

```
    // El valor de densidad necesario para validar el candidato actual.
    const float threshold = u_dithering_matrix[threshold_matrix_index.x][threshold_matrix_index.y];

    // El valor de densidad, muestreado desde el mapa de densidad, y
    // sumado con el valor total acumulado hasta el momento.
    const float density = density_array[array_index][gl_LocalInvocationID.x][gl_LocalInvocationID.y]
                        + sampleDensityMap(world_uv);

    density_array[array_index][gl_LocalInvocationID.x][gl_LocalInvocationID.y] = density;

    const vec2 position2d = candidate_array[array_index][gl_LocalInvocationID.x][gl_LocalInvocationID.y].position.xy;

    // Se determina si el punto está dentro del área de posicionamiento.
    const bool above_lower_bound = all(greaterThanEqual(position2d, u_lower_bound));
    const bool below_upper_bound = all(lessThan(position2d, u_upper_bound));

    const uint current_layer_index = candidate_array[array_index][gl_LocalInvocationID.x][gl_LocalInvocationID.y].class_index;

    // Se evalúa la validez del punto.
    // Si no había sido validado antes, se le asigna el índice del mapa de densidad actual.
    if (u_class_index < current_layer_index
            && density > threshold && above_lower_bound
            && below_upper_bound)
        candidate_array[array_index][gl_LocalInvocationID.x][gl_LocalInvocationID.y].class_index = u_class_index;
}

```

Código .3: Kernel de indización.

```
#version 450 core

#define INVALID_INDEX 0xFFffFFff

// En este kernel los work groups son lineales y tienen la mitad de hilos que los anteriores,
// pues cada hilo opera sobre dos candidatos.
layout(local_size_x = 32) in;

struct Candidate
{
    vec3 position;
```

```
12      uint class_index;
13  };
14
15  layout(std430) restrict readonly
16  buffer CandidateBuffer
17  {
18      Candidate array[];
19  } b_candidate;
20
21  uint readClassIndex(uint index)
22  {
23      return index < b_candidate.array.length() ? b_candidate.array[index].class_index : ↩
        ↪ INVALID_INDEX;
24  }
25
26  // Arreglo que contiene el total de candidatos válidos por capa.
27  // Cada capa viene de un mapa de densidad.
28  layout(std430) restrict
29  buffer CountBuffer
30  {
31      uint array[];
32  } b_count;
33
34  // Arreglo en que se escribirán los índices de copia.
35  layout(std430) restrict writeonly
36  buffer IndexBuffer
37  {
38      uint array[];
39  } b_index;
40
41  void writeIndex(uint array_index, uint value)
42  {
43      if (array_index < b_index.array.length())
44          b_index.array[array_index] = value;
45  }
46
47  shared uint s_index_array[2 * gl_WorkGroupSize.x];
48  shared uint s_index_offset;
49
50  void initLocalIndexArray(uvec2 array_index, uvec2 value)
51  {
52      s_index_array[array_index.x] = value.x;
53      s_index_array[array_index.y] = value.y;
54  }
55
```

```
// Ejecuta el algoritmo de indización en paralelo para todos
// los elementos del work group.
void addUpLocalIndexArray()
{
   for (uint group_size = 1; group_size < 2 * gl_WorkGroupSize.x; group_size <<= 1)
   {
      const uint group_index = (gl_LocalInvocationID.x / group_size) * 2 + 1;
      const uint base_index = group_index * group_size;
      const uint write_index = base_index + gl_LocalInvocationID.x % group_size;
      const uint read_index = base_index - 1;

      s_index_array[write_index] += s_index_array[read_index];

      barrier();
      memoryBarrierShared();
   }
}

uint atomicAddToClassCount(uint class_index)
{
   const uint local_sum = s_index_array[2 * gl_WorkGroupSize.x - 1];
   return atomicAdd(b_count.array[class_index], local_sum);
}

void main()
{
   const uvec2 local_index = {gl_LocalInvocationID.x, gl_LocalInvocationID.x + gl_WorkGroupSize.x};
   const uvec2 global_index = uvec2(gl_WorkGroupID.x * 2 * gl_WorkGroupSize.x) + local_index;
   const uvec2 class_index = {readClassIndex(global_index.x), readClassIndex(global_index.y)};

   uvec2 result_value = uvec2(INVALID_INDEX);

   for (uint i = 0; i < b_count.array.length(); i++)
   {
      initLocalIndexArray(local_index, uvec2(equal(class_index, uvec2(i))));

      barrier();
      memoryBarrierShared();

      // Se enumeran los candidatos válidos en este work group.
      addUpLocalIndexArray();

```

```
98        // Se suma (atómicamente) el número de candidatos válidos del work group
99        // al contador global.
100       // El valor previo del contador se utiliza como el offset del work group.
101       if (gl_LocalInvocationIndex == 0)
102          s_index_offset = atomicAddToClassCount(i);
103
104       barrier();
105       memoryBarrierShared();
106
107       // Se calcula el índice global de cada candidato válido en base a su
108       // índice local y el offset del work group.
109       if (class_index.x == i)
110          result_value.x = s_index_array[local_index.x] + s_index_offset - 1;
111       if (class_index.y == i )
112          result_value.y = s_index_array[local_index.y] + s_index_offset - 1;
113    }
114
115    writeIndex(global_index.x, result_value.x);
116    writeIndex(global_index.y, result_value.y);
117 }
118
```

Código .4: Kernel de copia.

```
1  #version 430 core
2
3  #define INVALID_INDEX 0xFFffFFff
4
5  // Este kernel el doble de items por grupo que el anterior
6  // porque cada hilo copia un elemento.
7  layout(local_size_x = 64) in;
8
9  struct Candidate
10 {
11    vec3 position;
12    uint class_index;
13 };
14
15 // Arreglo que contiene todos los candidatos,
16 // el mismo utilizado en las etapas anteriores.
17 layout(std430) restrict readonly
18 buffer CandidateBuffer
19 {
20       Candidate array[];
21 } b_candidate;
```

```
22
23 layout(std430) restrict readonly
24 buffer IndexBuffer
25 {
26       uint array[];
27 } b_index;
28
29 // Arreglo al que se copiarán sólo los candidatos válidos.
30 layout(std430) restrict writeonly
31 buffer OutputBuffer
32 {
33       Candidate array[];
34 } b_output;
35
36 layout(std430) restrict readonly
37 buffer CountBuffer
38 {
39    uint array[];
40 } b_count;
41
42 void main()
43 {
44    const uint candidate_index = gl_GlobalInvocationID.x;
45    if (candidate_index >= b_candidate.array.length())
46       return;
47
48    // los candidatos inválidos no se copian
49    const Candidate candidate = b_candidate.array[candidate_index];
50    if (candidate.class_index == INVALID_INDEX)
51       return;
52
53    // se obtiene el índice de copia del mapa de densidad al que
54    // pertenece el candidato.
55    const uint copy_index = b_index.array[candidate_index];
56
57    // el índice de copia global se obtiene sumando la cantidad de candidatos
58    // válidos en todas los mapas de densidad previos.
59    uint index_offset = 0;
60    for (uint class_index = 0; class_index < candidate.class_index; class_index++)
61       index_offset += b_count.array[class_index];
62
63    // copia del elemento al arreglo final
64    b_output.array[copy_index + index_offset] = candidate;
65 }
66
```