Escuela Politécnica Superior de Jaén

Grado en Ingeniería Informática

UNIVERSIDAD DE JAÉN
Escuela Politécnica Superior de Jaén

Trabajo Fin de Grado

SIMULADOR DE UNA FEMTO-CÁMARA

Alumno: Alejandro Jiménez Pereira

Tutor: Prof. D. Juan Roberto Jiménez Pérez
Dpto: Informática

Junio, 2015

Universidad de Jaén
Escuela Politécnica Superior de Jaén
Departamento de Informática

Don JUAN ROBERTO JIMÉNEZ PÉREZ, tutor del Proyecto Fin de Carrera titulado: SIMULADOR DE UNA FEMTO-CÁMARA, que presenta ALEJANDRO JIMÉNEZ PEREIRA, autoriza su presentación para defensa y evaluación en la Escuela Politécnica Superior de Jaén.

Jaén, junio de 2015

El alumno: Los tutores:

ALEJANDRO JIMÉNEZ PEREIRA JUAN ROBERTO JIMÉNEZ PÉREZ

Índice

1. INTRODUCCIÓN 5
1.1. Motivación 6
1.2. Experimento del MIT 9
1.2.1. Resultados 10
1.3. Ámbito del TFG 14
2. GESTIÓN Y PLANIFICACIÓN 15
2.1. Planificación temporal 15
Diagrama PERT 17
Diagrama de Gantt 19
2.2. Análisis de costes 20
Material 20
Cálculo de luz 21
COCOMO 22
3. ANÁLISIS 26
4. DISEÑO 30
4.1. Escena 31
4.2. Diagramas UML 33
Diagrama de clases 33
Diagrama de secuencia 37
Diagrama de tiempos 37
Diagrama de máquina de estados 39
Diagrama de actividades 40
5. IMPLEMENTACIÓN 40
Manual 61
6. RESULTADOS 70
7. CONCLUSIONES 77
8. TRABAJOS FUTUROS 79

Aplicaciones.. 80
Bibliografía.. 82

1. INTRODUCCIÓN

La femto-cámara es un dispositivo capaz de capturar un trillón de imágenes por segundo, grabando así un vídeo a cámara súper lenta. Este dispositivo ha sido desarrollado por el **MIT Media Lab's Camera Culture group** (*Massachusetts Institute of Technology*) en colaboración con **Bawendi Lab** en el departamento de química del MIT.

Gracias a este dispositivo se puede observar la propagación de la luz, de cómo los fotones iluminan una escena. Esto tiene muchas aplicaciones directas e indirectas en el conocimiento de la luz y de cómo se propaga en el ambiente. En la ilustración 1.1 vemos una visualización time-lapse de los frentes esféricos de la luz reflejada avanzando por las superficies de la escena.

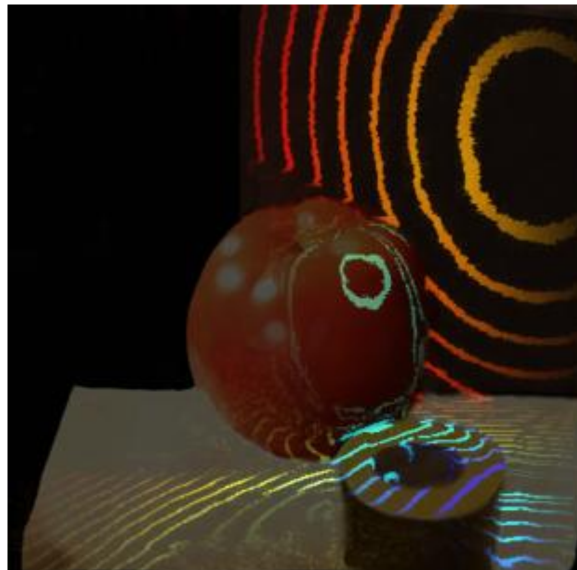

Ilustración 1.1. Time-Lapse. http://web.media.mit.edu/~raskar/trillionfps/

Además de las aplicaciones obvias de la femto-cámara tiene otras relacionadas con la recuperación de información tridimensional de objetos ocultos. Este fué el objetivo de un experimento realizado en el MIT que explicaré en el punto 1.2 y del cual trata este trabajo.

1.1. Motivación

La motivación del experimento era poder percibir un objeto tridimensional que a priori era imposible de ver ya que se encontraba oculto por otros objetos de la escena. Esto sería a *grosso modo* como una visión de rayos X a través del obstáculo que nos impide ver el objeto. Esta idea que suena a ciencia ficción tiene muchas aplicaciones muy útiles en campos muy diversos como seguridad vial, medicina, información de una escena en caso de incendio, y muchas más.

Para poder entender el funcionamiento del experimento es necesario repasar el comportamiento de la luz. La luz es una dualidad onda-partícula, en este trabajo nos vamos a centrar en la componente partícula de la luz, obviando una serie de fenómenos, que se explican por su comportamiento ondulatorio: difracción, … . Como partícula lo que hace es interactuar con los objetos de la escena que absorberán, reflejarán, refractarán o transmitirán la misma. La interacción de la luz con la escena, es decir, con cada objeto de la escena depende de las propiedades ópticas del material del objeto.

Para empezar hay que presentar *el espectro de luz visible*. El espectro de luz visible está compuesto por diferentes longitudes de onda que corresponde a los colores que percibimos. Podemos verlo en la ilustración 1.2. Como podremos ver más abajo en el experimento se utilizó un láser con una longitud de onda de 795 nm que está por los niveles infrarrojos.

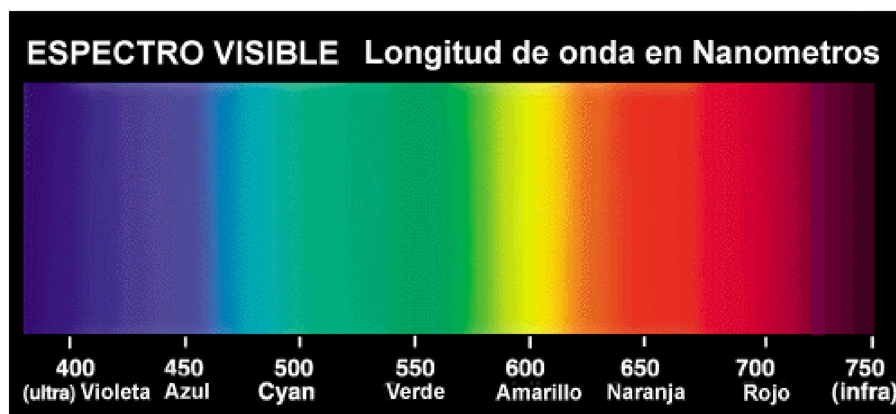

Ilustración 1.2. Espectro visible. https://teoriadelcolorymas.wordpress.com/2013/11/02/una-mas/

Absorción

Cuando la luz llega a un objeto éste puede absorber parte de la luz, toda la luz o nada de luz. Si un objeto absorbe todas las longitudes de onda de este espectro, no reflejará ninguna luz visible y por lo tanto veremos un objeto negro. Si no es capaz de absorber ninguna longitud de onda del espectro visible, reflejará toda la luz y veremos un objeto blanco. Es decir que el color del objeto que vemos va a ser la suma de las luces con longitudes de onda que no es capaz de absorber el objeto. Poniendo un ejemplo simplificado en la ilustración 1.3, si trabajamos con un representación RGB y tenemos un pantalón de color rojo, otro negro y otro blanco, el primero absorbería las componentes G y B, el negro absorbería R,G y B y el blanco no absorbería ninguna.

Ilustración 1.3. Ejemplo absorción de la luz. [Propiedades de la luz]

Reflexión

Cuando la luz llega al objeto y se refleja o rebota total o parcialmente. Hay dos tipos de reflexiones:

Reflexión especular: cuando la luz rebota con el mismo ángulo de incidencia. En las superficies brillantes, la reflexión especular se corresponde con la zona de la superficie que brilla, por reflejarse total o casi totalmente la luz

Reflexión difusa: la luz se refleja en la superficie con la misma intensidad en todas las direcciones, independientemente del punto de visión.

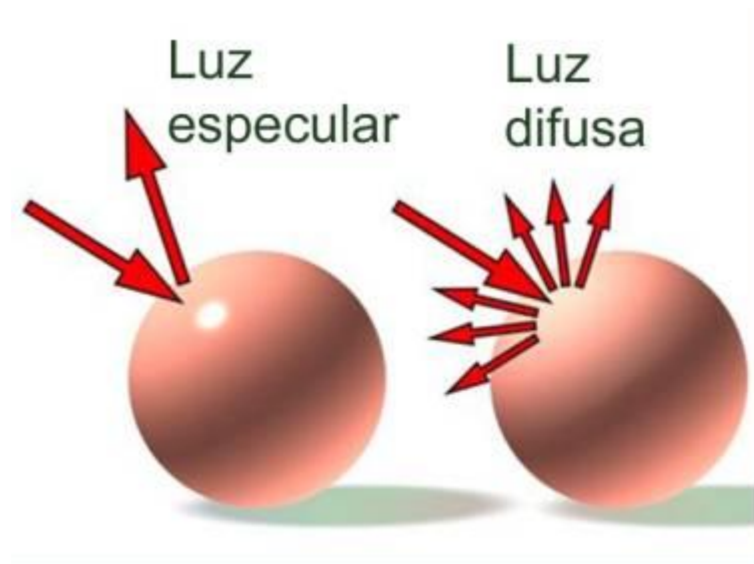

Ilustración 1.4. Tipos de reflexiones. http://web.salleurl.edu/~jherrero/Autodesk%20VIZ%204.htm

Normalmente cada material de cada objeto tiene una combinación de ambas reflexiones pero para el trabajo nos interesa la reflexión difusa por lo tanto necesitamos materiales con una alta componente difusa. Lo más cercano a superficies *lambertianas* como se muestra en la ilustración 1.5.

Ilustración 1.5. Comportamiento de la luz en las reflexiones. [Propiedades de la luz]

Transmisión

La transmisión ocurre cuando la luz atraviesa el objeto o una superficie. Se da en objetos no opacos. Esta propiedad queda fuera del alcance de este trabajo.

1.2. Experimento del MIT

El experimento que se realizó en el MIT consistió en lanzar un haz de luz de rayo láser ultrarrápido contra una pared la cual tiene un alto componente difuso en la que al chocar propaga la luz reflejada de forma difusa, esta luz choca con el objeto oculto y se refleja por múltiples superficies del objeto, y vuelve a la pared difusa, donde es reflejado de nuevo y capturado por la cámara, como en la ilustración 1.6, extraída de *Kirmani2011_LAC*. Haciendo varias veces este proceso, cambiando el ángulo de dirección del láser y guardando el *time-of-flight*, tiempo de vuelo, recuperamos una pila de imágenes que serán mapas de calor, *heatmap*. Estos mapas de calor representarán la intensidad de cada fotón.

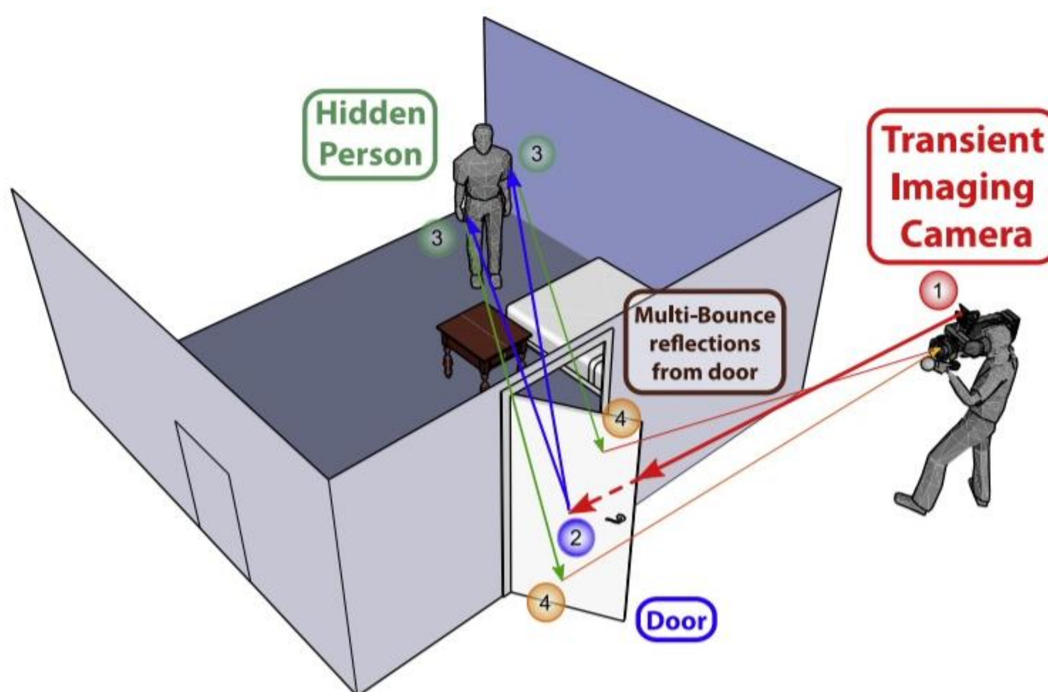

Ilustración 1.6. Escena hipotética del experimento. Kirmani2011_LAC

Una vez con la pila de *heatmaps* estas se procesan con técnicas de procesamiento de imágenes y un algoritmo de reconstrucción para obtener la información tridimensional del objeto aproximadamente.

1.2.1. Resultados

A continuación, veréis los resultados del experimento. Empezaremos por el setup, como están posicionados y distribuidos todos los elementos que intervienen en el proceso. El esquema del setup se puede ver en la ilustración 1.7 a, que es una imagen extraída del artículo *Velten2012_RTD* et al. [Velten2012]. Está el objeto oculto, en este caso un maniquí de 20 cm de alto, escondido de la cámara por el occluder, el obstáculo. La pared difusora, Wall, con una componente difusa muy alta para la dispersión difusa de la luz, como comenté antes. Y en el otro lado tenemos la cámara de barrido, Streak camera, la femto-cámara que apunta a la pared difusora, el láser ultrarrápido con un sistema de divisores del rayo que manda una señal al photodetector para tener sincronizado el láser con la cámara. El Galvo (conjunto de de espejos de galvanómetro accionado) sirve para que el haz del láser modifique su dirección y poder apuntar a distintos puntos de la pared difusora. La línea discontinua es lo que la femto-cámara ve.

Los fotones de luz que salen del láser siguen la trayectoria r_1 hasta que choca en el punto *L*, en la pared difusora. Al chocar, la luz se propaga de forma esférica (difusamente) por toda la escena; alguna luz difusa, hace la trayectoria r_2 hasta que golpea el objeto oculto *s*, luego hace el camino r_3 hasta impactar de nuevo en *Wall* en el punto *W* y finalmente recorriendo r_4 es capturado por la cámara de barrido.

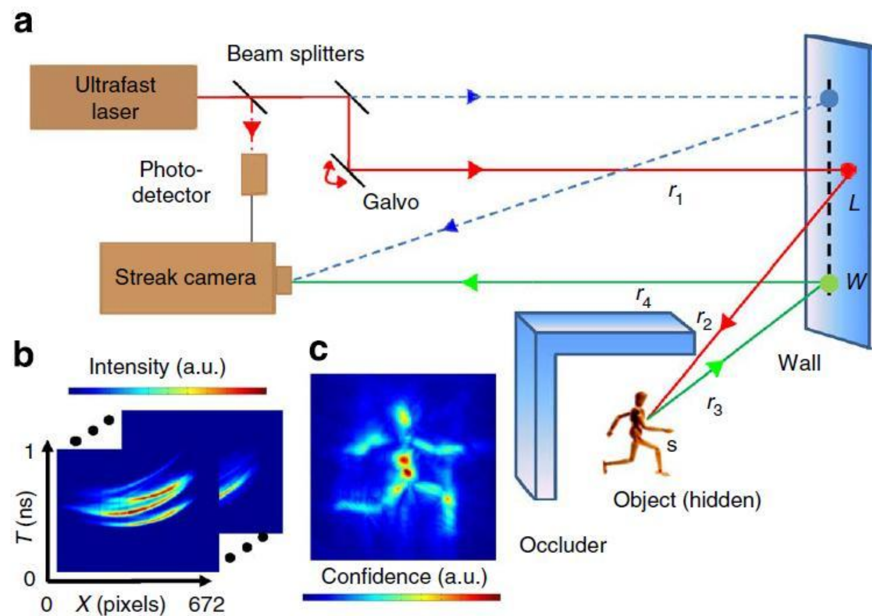

Ilustración 1.7. Velten2012_RTD (Setup).

Por cada lanzamiento de pulso de luz del láser la femto-cámara digitaliza la información captada en intervalos de 2 ps (picosegundos, 10^{-12}) generando una colección de heatmaps, cada uno asociado a una posición distinta del láser, ilustración 1.7 b y ilustración 1.9 b. Como podemos ver en la ilustración 1.8, los mapas de calor tienen una dimensión espacial, el eje de abscisas, que representa la luz capturada por la cámara y una dimensión temporal, el eje de ordenadas, que es el tiempo que la cámara ha estado capturando información. La componente X va de 0 a 672 es la precisión de la cámara y el eje Y va de 0 a 1 ns (nanosegundo, 10^{-9}) es cuanto dura la cámara abierta capturando, si captura cada 2 ps la imagen tendrá 500 píxeles de altura. El color representa la intensidad con la que el fotón llega a la cámara; este color está normalizado siendo rojo la máxima intensidad y azul la mínima.

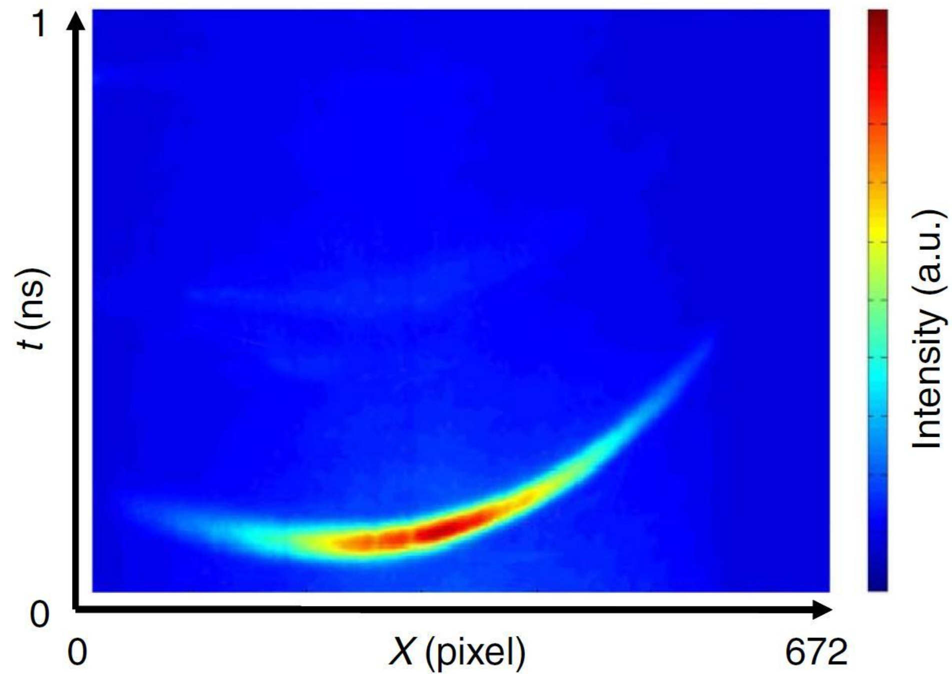

Ilustración 1.8. Heatmap. Velten2012_RTD.

El siguiente paso es usar un algoritmo de retroproyección, backprojection, para los heatmap que genera un único heatmap ilustración 1.9 c. Después va el filtrado, que consiste en hacer la segunda derivada respecto a la profundidad (z) dejando así una mapa de calor en el que ya se puede observar el contorno de la forma del objeto, ilustración 1.9 d. La imagen e es un mapa de profundidad de la figura, la f es un mapa de confianza. Y el resto es el objeto que reconstruye el algoritmo.

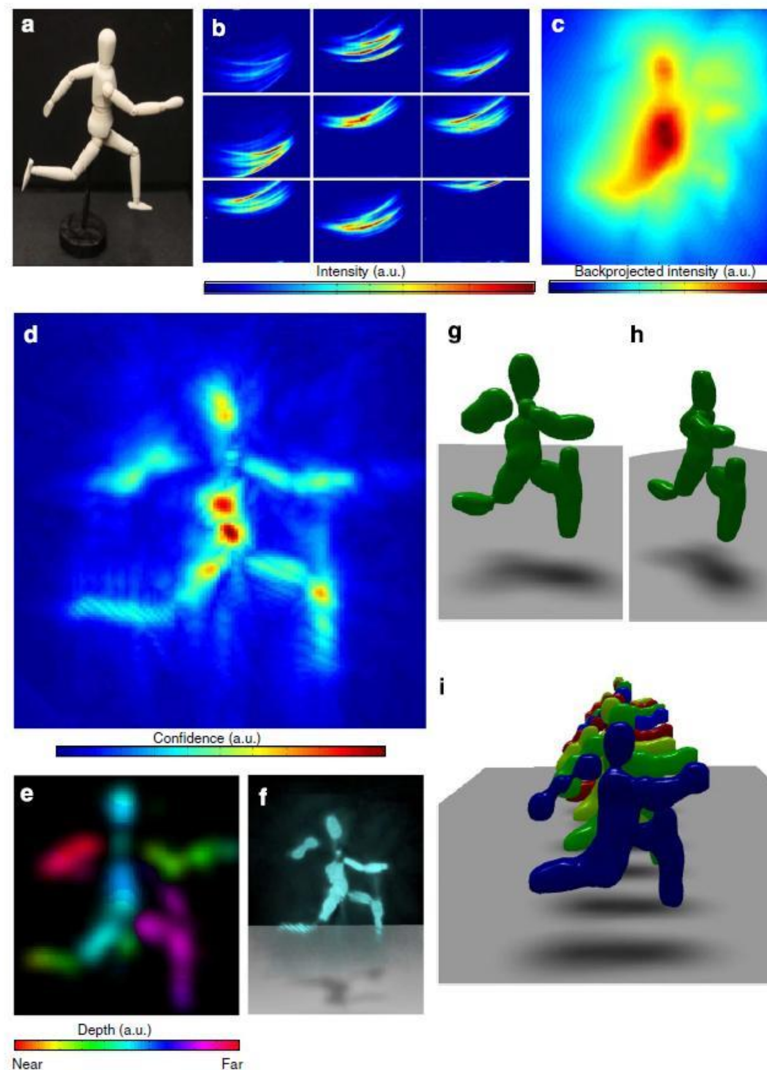

Ilustración 1.9. Velten2012_RTD (Resultados experimento).

En la siguiente tabla están recopilados algunos datos importantes sobre el experimento que serán necesarios tenerlos en cuenta a la hora de diseñar e implementar el simulador.

Dato	Valor
Velocidad de la luz	$3x10^8$ m/s
Pulsos largos del láser	50 fs
Frecuencia de emisión láser	75 MHz (13.3 ns)
Longitud de onda del láser	795 nm
Intervalo en el que la cámara digitaliza la información (cada	2 ps

pixel en Y)	
Tiempo que la cámara permanece abierta	1 ns
Distancia de la cámara a la pared	62 cm
Altura de Wall	40 cm
Anchura de Wall	25 cm
Altura maniquí	20 cm
Distancia maniquí pared	25 cm

Tabla 1.1. Valores del experimento.

1.3. Ámbito del TFG

El simulador de la femto-cámara va a simular el comportamiento de la luz como partícula y de cómo interacciona con una escena con objetos que tendrán las propiedades de absorción y reflexión difusa y especular. La escena será un representación del esquema del experimento explicado por Velten et al. [Velten2012], con sus elementos básicos: el láser, la femto-cámara, el occluder, la pared difusa y el objeto oculto. El objetivo del simulador será recoger información de las partículas que son captadas por la cámara la cual se traduce como poder generar lo mapas de calor necesarios para la reconstrucción del objeto oculto.

Sin embargo, la reconstrucción del modelo y el procesamiento de las imágenes se queda fuera del ámbito de este TFG. No se tendrá en cuenta la longitud de onda de la luz empleada. El material de la pared difusa se supone totalmente difusa para facilitar la recogida de información. No se contemplan varias fuentes emisoras de luz, varios objetos ocultos ni varias paredes difusas.

Como herramienta se utilizará el motor gráfico **Unity 3d** con su potente interfaz de usuario para modelar la escena y su sistema de scripts para crear la lógica del simulador. Este entorno está pensado para desarrollar videojuegos multiplataforma en 2D y 3D pero yo lo he utilizaré como herramienta para crear la escena de la simulación. He elegido Unity porque ya tenía experiencia y conocimientos previos en este motor al haber cursado la asignatura “Desarrollo de videojuegos”. Entre otras herramientas del motor usaré su sistema de partículas *(Shuriken Particle System)*, el

sistema de detección de colisiones y su interfaz para crear los elementos gráficos de la escena. El motor funciona con objetos compuestos por componentes(transform, colliders, mesh, …) y la forma con la que el código se implementa en la escena y objetos es mediantes **scripts**. En estos scripts está toda la lógica del simulador. Los scripts de Unity se pueden escribir en C#, JavaScript y Boo. Yo he elegido el lenguaje **JavaScript** debido a su flexibilidad, facilidad de uso y porque tiene más soporte en la guía de codificación (más documentación). Para decantarme por un lenguaje u otro me he informado en este blog.

2. GESTIÓN Y PLANIFICACIÓN

Para la planificación de este TFG se ha llevado a cabo las actividades previas necesarias en cualquier proyecto informático un análisis, un cálculo de costes y una planificación temporal. Utilizando modelos aprendidos durante la carrera como diagramas Gantt y Pert, Cocomo, etcétera.

2.1. Planificación temporal

Para el correcto desarrollo del TFG he llevado a cabo una gestión del tiempo. Para ello he seguido las siguientes acciones:

- Definición de las tareas
- Estimación de la duración de las mismas
- Secuencia de actividades
- Establecimiento del calendario
- Control del calendario

Las tareas que he definido son:

- Búsqueda Bibliográfica sobre el experimento desarrollado en el MIT
- Revisión Bibliográfica
- Estudio de los elementos del simulador (setup, variables)
- Diseño de la escena (setup)
- Diseño de la Interfaz Gráfica de Usuario

- Implementación del sistema de partículas y el manejo de sus colisiones
- Implementación del algoritmo principal
- Implementación de las propiedades de los materiales
- Implementación de la función de generación de los heatmap
- Implementación de la GUI
- Pruebas y Tests
- Redactar memoria

Una vez definidas las actividades he estimado el tiempo que pienso que pueden durar. Este tiempo es aproximado y puede que no sea real ya que la actividad se puede complicar o facilitar. El tiempo se mide en días y cada día supone 3 horas de trabajo. También he diseñado la secuencia de las actividades oportuna. El orden de estas actividades es importante a la hora de respetar el calendario para que no haya retrasos.

Como el plan bolonia especifica que un crédito equivale de 25 a 30 horas de trabajo, ver bibliografía. Como el TFG son 12 créditos el supuesto total de horas serían entre 300 y 360 horas. Si hay que trabajar 360 horas echando 3 horas al día habría que trabajar 120 días por 5 días a la semana saldrían 24 semanas o lo que es equivalente a 5 meses y una semana aproximadamente.

Tabla de actividades:

Tarea	Descripción	Tiempo	Predecesoras
A	Búsqueda Bibliográfica	5	-
B	Revisión Bibliográfica	10	A
C	Estudio de los elementos	4	B
D	Diseño de la escena	2	C
E	Diseño de la Interfaz Gráfica de Usuario	10	C
F	Implementación Sistemas de partículas y colisiones	15	D
G	Implementación del algoritmo principal	15	C
H	Implementación de las propiedades de los materiales	5	D
I	Implementación de la función de generación de los heatmap	8	H, G, F

J	Implementación de la GUI	15	E
K	Pruebas y Tests	16	I
L	Redactar memoria	15	J, K

Tabla 2.1. Tabla de actividades.

Diagrama PERT

El diagrama PERT ha sido diseñado con la herramienta easyPERT desarrollada por mi compañero Cristóbal Mata de GuGames DEV. Este software me ha permitido dibujar el diagrama PERT con las actividades y su duración, ilustración 2.1.

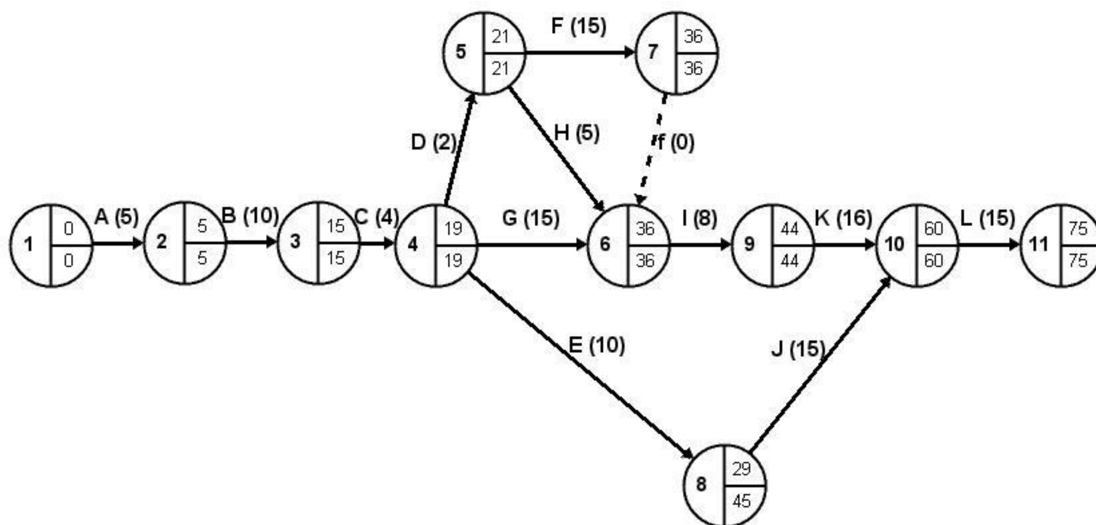

Ilustración 2.1. Diagrama PERT.

El mismo software calcula la tabla de actividades con sus tiempos de inicio y de fin y las holguras, indicando las actividades que pertenecen al camino crítico en rojo. El resultado es la tabla de abajo, ilustración 2.2.

Tarea	Precedentes	Duración	I_m	F_M	I_M	F_m	H_I	H_T
A	-	5	0	5	0	5	0	0
B	A	10	5	15	5	15	0	0
C	B	4	15	19	15	19	0	0
D	C	2	19	21	19	21	0	0
E	C	10	19	45	19	29	0	16
G	C	15	19	36	19	34	2	2
H	D	5	21	36	21	26	10	10
F	D	15	21	36	21	36	0	0
I	G,H,F	8	36	44	36	44	0	0
K	I	16	44	60	44	60	0	0
J	E	15	29	60	45	44	0	16
L	K,J	15	60	75	60	75	0	0

** Las filas en rojo indican el camino crítico.*

Ilustración 2.2. Tabla de tiempos y holguras.

También genera el diagrama con su camino crítico dibujado en rojo, ilustración 2.3. Las actividades que pertenecen al camino crítico son: A,B,C,D,F,I,K,L.

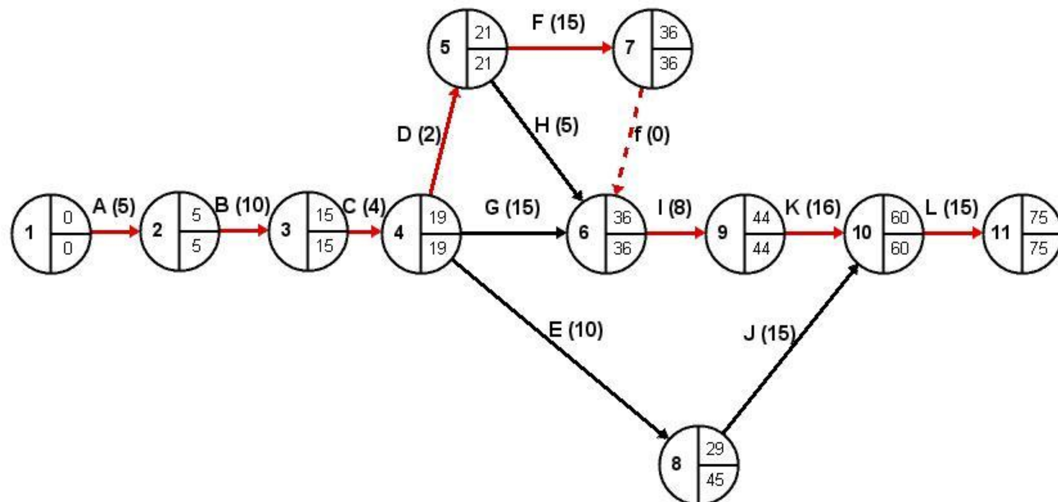

Ilustración 2.3. Diagrama PERT con camino crítico.

Diagrama de Gantt

Para hacer el diagrama de Gantt he usado la herramienta online Wrike, es similar a **Redmine**, con una interfaz de usuario muy amigable y permite crear un flujo de trabajo insertando actividades y generando el correspondiente diagrama de Gantt. El diagrama de Gantt generado es el de la ilustración 2.4. A día 28 de mayo el diagrama de Gantt se encuentra como en la ilustración 2.5.

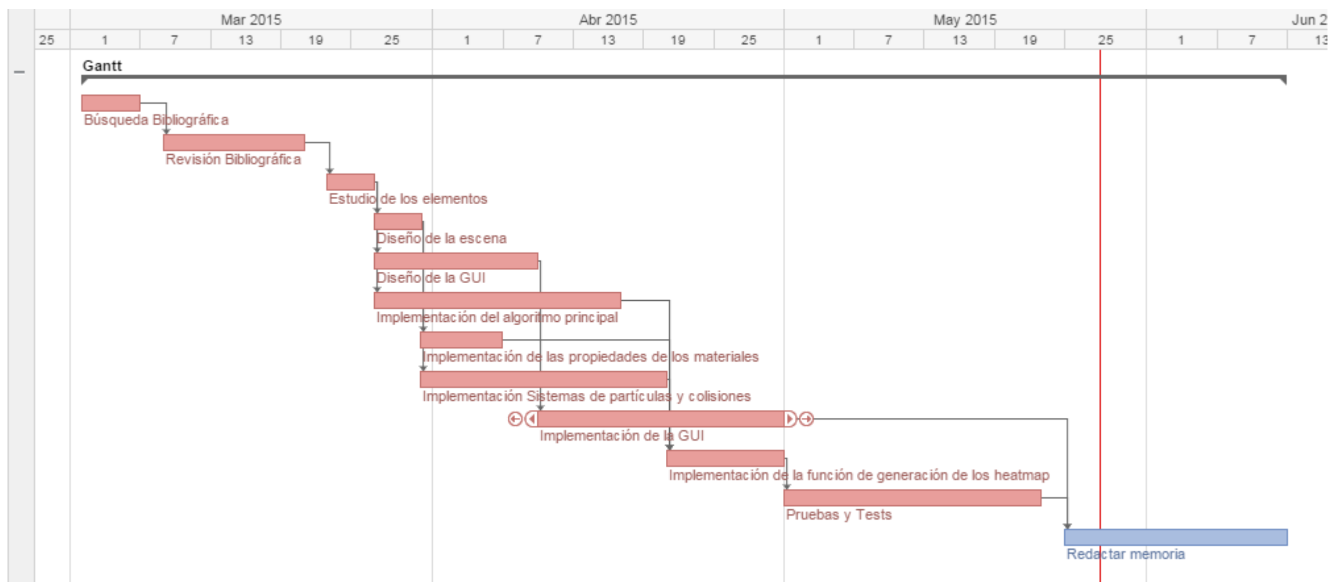

Ilustración 2.4. Diagrama de Gantt.

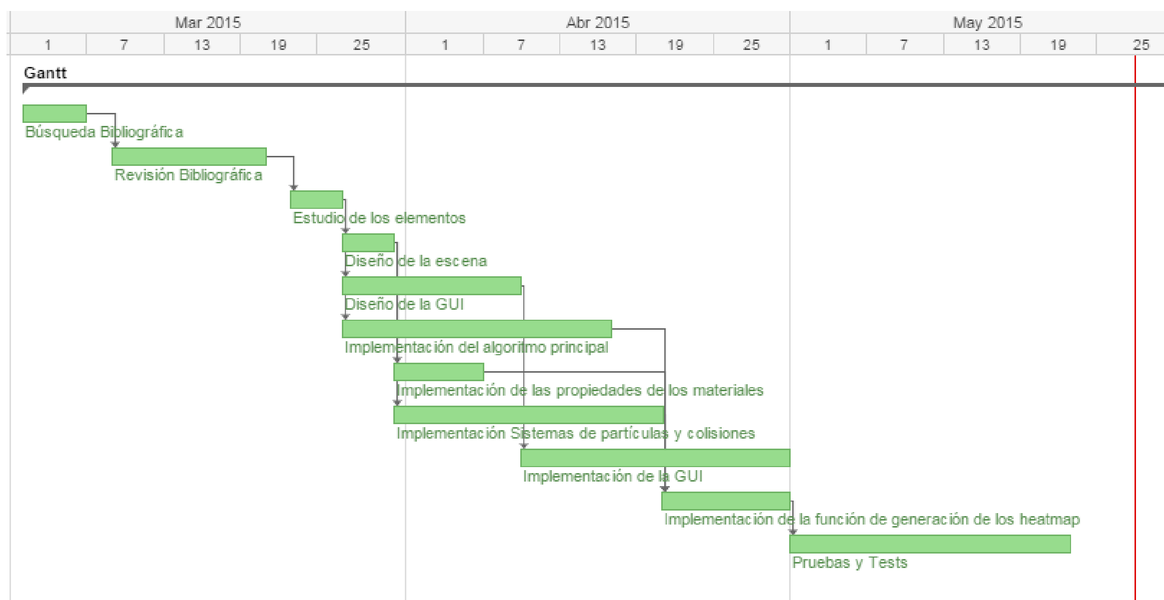

Ilustración 2.5. Diagrama de Gantt a fecha 28 de mayo de 2015.

Para el control del calendario, mi tutor y yo hemos tenido reuniones semanales en las que revisamos el trabajo realizado durante la semana. Y cada semana me dedicaba a un tema concreto del proyecto. Es un método que emula la metodología ágil de *SCRUM* en la que cada semana es una revisión.

2.2. Análisis de costes

Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Proin pharetra nonummy pede.

- Planificación de recursos: Determinando qué recursos (personal, equipamiento, materiales…) y en qué cantidad son necesarios para desarrollar las tareas del proyecto.
- Estimación de costes: calculando aproximadamente los costes de los recursos consumidos
- Presupuestos y Financiación de los costes: Estableciendo en qué categoría se deben asignar estos costes dentro del presupuesto
- Control de costes: Controlando y supervisando los cambios de costes dentro del proyecto.

Los recursos usados en el desarrollo de este software de simulación han sido los siguientes:

Material

La **licencia Unity** debe estar dentro de los costes del desarrollo de este trabajo. Hay tres tipos de licencias en Unity: la versión de prueba, que es gratuita pero que caduca a los 30 días. La edición “Professional”, anterior “Unity Pro”, de pago que tiene un precio de 1500$ o de 75$/mes para la reciente versión 5. Y la versión gratuita “Unity Free” o “Personal” que tiene menos funcionalidades que la versión de pago. Yo he utilizado la licencia “Personal” por lo tanto el gasto por el motor gráfico es nulo.

Para el diseño de mi proyecto, diagramas UML, he usado la herramienta **Visual Paradigm**. Esta es una herramienta Software para el diseño de UML2. Este software lo utilicé en la asignatura de *Fundamentos de Ingeniería del Software* con una

licencia de estudiante ya que por aquel momento era de pago. Sin embargo, actualmente y para mi proyecto ha salido la versión 12.1 community edition que es una versión gratuita para uso no comercial.

La **bibliografía** referente al experimento del MIT al que hace referencia este proyecto, ver introducción, me fue suministrado por el tutor del TFG, Roberto Jiménez, en formato electrónico (PDF). Por lo tanto no he tenido ningún gasto sobre material bibliográfico. El material suministrado, apuntes de la carrera e Internet han sido todas las fuentes que he necesitado.

El **equipo informático** usado ha sido un ordenador personal (PC) montado a mano por componentes comprados en la tienda online http://www.pccomponentes.com/ y compuesto principalmente por: CPU Intel Core i5-4430 de cuatro núcleos a 3GHz, 8GB de memoria RAM, disco duro de estado sólido de 128GB (SSD), tarjeta gráfica AMD Radeon R9 200 Series GDDR5 de 2GB y fuente de alimentación NOX Urano VX 750 KW. Estos y demás componentes del equipo hacen un precio de 1050€.

Cálculo de luz

Luz que gasta el equipo según las horas de trabajo. Sumando los días de la tabla de actividades hacen 120 días, un total de 360 horas.

$$T = 120\, días * 3\, Horas = 360\, Horas$$

Para la pantalla, según la ficha consume 20 W (vatios) la hora. http://www.pccomponentes.com/lg_22mp55hq_p_22__led_ips.html

$$W_{pantalla} = 20W / 1000 = 0.02\, KW$$

Para calcular el consumo de mi PC he usado una calculadora online que da una cifra más o menos aproximada. He rellenado el formulario de la calculadora con los datos de mi PC y ha calculado una media de 445 W.

$$W_{PC} = 445\, W / 1000 = 0.445\, KW$$

El precio de KW por hora es 0,126122 €/KWh. Este dato lo he consultado en la web de Endesa con la configuración contratada en mi casa.

Esto hace un total de:

$$Consumo_{pantalla} = 0.02\,KW * 360\,Horas * 0.126122\,€/KWH = 0.9080784\,€$$

$$Consumo_{pantallla} = 0{,}445\,KW * 360\,Horas * 0.126122\,€/KWH = 20.2047444\,€$$

$$Consumo_{pantalla} = 0.9080784\,€ + 20.2047444\,€ = 21.1128228\,€ \simeq 21.11\,€$$

COCOMO

De entre los tres tipos de COCOMO el que mejor representa el tipo de proyecto es el modo semi-acoplado. He elegido este modo ya que el problema a resolver es único y no hay experiencia previa sobre el tema. Sin embargo, no hay muchas líneas de código por eso he elegido el modo semi-acoplado.

He optado por el modelo intermedio para tener un cálculo más preciso.

$$E = a \cdot KLDC^{b} \cdot FAE$$

$$T = c \cdot E^{d}$$

$$P = E/T$$

$$Coste = T \cdot P \cdot Salario\ Medio$$

Donde:

E, es el esfuerzo, en persona-mes

KLDC, es el número de líneas de código

FAE, es una variable que depende de 15 atributos

T, es el tiempo requerido, en meses

P, es el número de personas necesarias

Coste, es el coste total del proyecto

(Fórmula 2.1)

Calcular KLDC con Puntos de Función(PF) y Líneas de Código(LDC). El PF depende del lenguaje usado. Buscando un poco por la red he encontrado que el PF de JavaScript es 54. Las líneas de código estimadas por PF son 90.

$$KLDC = PF \cdot LDC/1000 = 54 \cdot 90/1000 = 4{,}86$$

	a	b	c	d
Semi-acoplado	3	1,12	2,5	0,35

Tabla 2.2. Coeficientes del modo-semiacoplado.

Para el cálculo de la variable FAE marcamos los coeficientes oportunos en la siguiente tabla.

CONDUCTORES DE COSTE	VALORACIÓN					
	Muy bajo	Bajo	Nominal	Alto	Muy alto	Extr. alto
Fiabilidad requerida del software	0,75	0,88	1.00	1,15	1,40	-
Tamaño de la base de datos	-	0,94	1.00	1,08	1,16	-
Complejidad del producto	0,70	0,85	1.00	1,15	1,30	1,65
Restricciones del tiempo de ejecución	-	-	1.00	1,11	1,30	1,66
Restricciones del almacenamiento principal	-	-	1.00	1,06	1,21	1,56
Volatilidad de la máquina virtual	-	0,87	1.00	1,15	1,30	-
Tiempo de respuesta del ordenador	-	0,87	1.00	1,07	1,15	-
Capacidad del analista	1,46	1,19	1.00	0,86	0,71	-
Experiencia en la aplicación	1,29	1,13	1.00	0,91	0,82	-

Capacidad de los programadores	1,42	1,17	1.00	0,86	0,70	-
Experiencia en S.O. utilizado	1,21	1,10	1.00	0,90	-	-
Experiencia en el lenguaje de programación	1,14	1,07	1.00	0,95	-	-
Prácticas de programación modernas	1,24	1,10	1.00	0,91	0,82	-
Utilización de herramientas software	1,24	1,10	1.00	0,91	0,83	-
Limitaciones de planificación del proyecto	1,23	1,08	1.00	1,04	1,10	-

Tabla 2.3. Coeficientes FAE.

$$FAE = 1 \cdot 0{,}94 \cdot 1{,}30 \cdot 1{,}30 \cdot 1 \cdot 1 \cdot 1{,}15 \cdot 1{,}19 \cdot 0{,}82 \cdot 0{,}70 \cdot 1 \cdot 1{,}14 \cdot 1 \cdot 1 \cdot 1{,}08 = 1{,}53638429516208$$

$$E = 3 \cdot 4{,}86^{1,12} \cdot 1{,}53638429516208 = 27{,}08\, personas\ mes$$

Cálculo del tiempo

$$T = 2{,}5 \cdot 27{,}08^{0,35} = 7{,}9317\, meses$$

Cálculo de las personas necesarias

$$P = 27{,}08/7{,}9317 = 3{,}4141\, personas$$

Según los resultados necesitaría un grupo de 4 personas trabajando durantes 8 meses.

Para estimar el sueldo promedio de cada trabajador busqué en la web acerca del tema. Tenía entendido que esto estaba regulado según los estatutos pero tras indagar no encontré información al respecto. Por lo tanto recurrí a mandar un email al “Colegio Profesional de Ingenieros en Informática de Andalucía” preguntando sobre el tema. Me respondieron que no existe un sueldo preestablecido y que

realmente las empresas y el mundo laboral son quienes establecen los salarios. No hay ningún organismo que regule qué trabajo puede hacer o no un ingeniero informático ni cuánto debe cobrar. Me recomendaron buscar ofertas de trabajo en portales webs como Infojobs y similares para orientarme sobre la cuantía de los salarios. Como detalle también me aclararon que ellos abarcan los estudios de Máster y que existe otra institución "Colegio Profesional de Ingenieros Técnicos en Informática de Andalucía" que son los equivalentes al Grado. Finalmente para determinar un salario busqué en *Infojobs* y *Tecnoempleo* unas seis ofertas de trabajo correspondientes a la palabra clave "*ingenieros técnicos informáticos"* y calcule el promedio del salario de estos anuncios. Normalmente las ofertas tienen un rango de salario, por ejemplo desde 900 a 1500 € al mes, así que hice la media de todos estos números y el resultado fué 1624 €, aproximadamente.

Así que, con un sueldo medio de 1624 € por persona al mes el coste del proyecto sería:

$$Coste = 4 \cdot 8 \cdot 1624 = 51968\,€$$

Calculando el total de todos los costes:

Concepto	**Precio**
Licencia Unity	0 €
Licencia Visual Paradigm	0 €
Consumo de luz	21.11 €
Estimación de costes COCOMO	51 968 €
Coste PC	1 050 €
Total	53 039.11 €

Tabla 2.4. Resultado de la estimación de costes.

3. ANÁLISIS

Lo primera tarea que hice para el proyecto, incluso antes de empezarlo, fué leer el artículo de Velten [Velten2012] donde se explicaba el experimento. Posteriormente mi tutor me proporcionó más artículos relacionados con la investigación, después de leerlos los descarté ya que unos estaban poco detallados y otros eran un versión más complicada del experimento. A pesar de la complejidad del experimento, me centré en el artículo Velten [Velten2012] por la simplicidad de la escena y porque estaba explicada con un alto nivel de detalle, dando detalles de valores y variables.

Para el simulador era necesario implementar la propiedad de reflexión de la luz para que las partículas de luz se propaguen difusamente. Con planes de futuro también he decidido añadir un campo de absorción y la longitud de onda asociada a la luz que refleja la superficie para poder ampliar la complejidad de la luz en la simulación.

Se espera del simulador que sea genérico, es decir, que funcione independientemente de los cambios en la escena. Excepto, lógicamente, que falte la femto-cámara, el láser o la pared difusora. Que tenga unos resultados realistas. También, dotar al simulador la capacidad de que el usuario pueda modificar las variables que intervienen en la simulación para que pueda experimentar todo lo que se le ocurra con él.

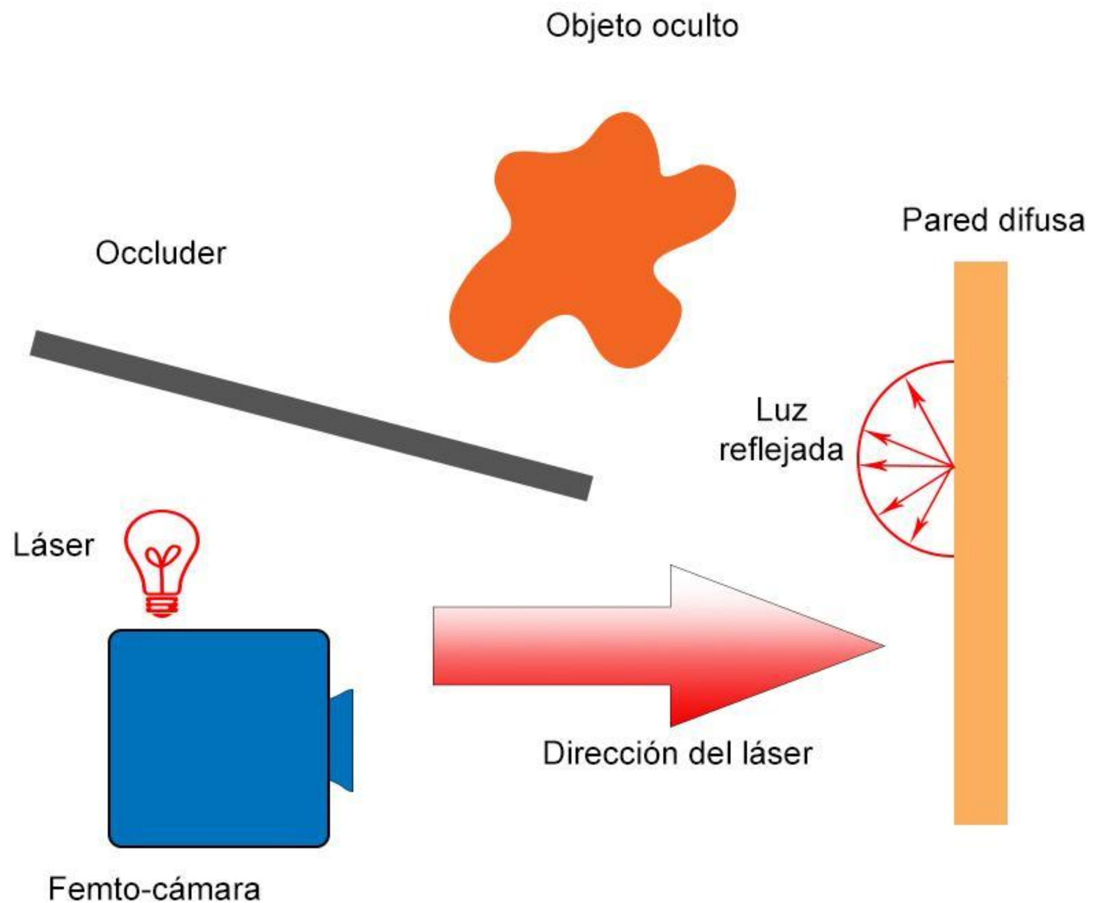

Ilustración 3.1. Esquema de la escena.

El proceso del simulador consiste en un bucle que se repite un número igual al número de imágenes que se vayan a generar. En la ilustración 3.2 se puede ver un esquema del bucle.

Ilustración 3.2. Esquema del bucle.

Prototipo 1.

Lo primero será montar la escena y que esta sea fiel a la escena del experimento. El láser enviará un pulso de luz hacia un punto en la pared difusora que al chocar se emitirán, por ejemplo, 500 partículas en todas direcciones. De estas partículas, las que choquen con el objeto oculto generará otras tantas de nuevo en todas las direcciones y de estas algunas serán las capturadas. La cámara permanecerá detectando partículas desde que el láser golpea la pared hasta que la partícula chocara con el objeto a una distancia igual a la de la cámara desde la pared y volviera a esta.

Prototipo 2.

Para esta revisión el cambio a hacer será modificar el tiempo que la cámara permanecerá abierta para que esté acorde con el tiempo real del experimento.

Prototipo 3.

Para este nuevo prototipo cambiaremos el modelo de la luz al modelo de Pattanaik. La diferencia respecto al anterior es que ahora hay un número constante de partículas y no se crean nuevas. Desde la pared difusa se lanzan todas estas partículas en todas direcciones y cuando colisionan con el objeto simplemente rebotan sin generar nuevas.

Prototipo 4.

Este prototipo es una optimización del anterior. Hacemos que la luz difusa enfoque al objeto oculto en un ángulo mucho más reducido de la semiesfera que se generaba. De esta forma las partículas de luz se emite en un cono con un ángulo específico. Se espera que más cantidad de partículas colisionen con el objeto oculto.

Prototipo 5.

En este prototipo quiero coger la idea del anterior y hacerlo genérico para cada escena posible. El problema del anterior estaba en que la luz tenía que saber donde se encontraba el objeto oculto para dirigir la luz a ese lugar. La manera de aprovechar la optimización anterior y hacerlo genérico es digamos parar el tiempo cuando la luz se envía desde la pared y procesar en diferentes ramas las distintas porciones del espacio. Esto se haría en un procesamiento individual y al final se juntarían todos los datos de colisiones uniendolas en una única imagen.

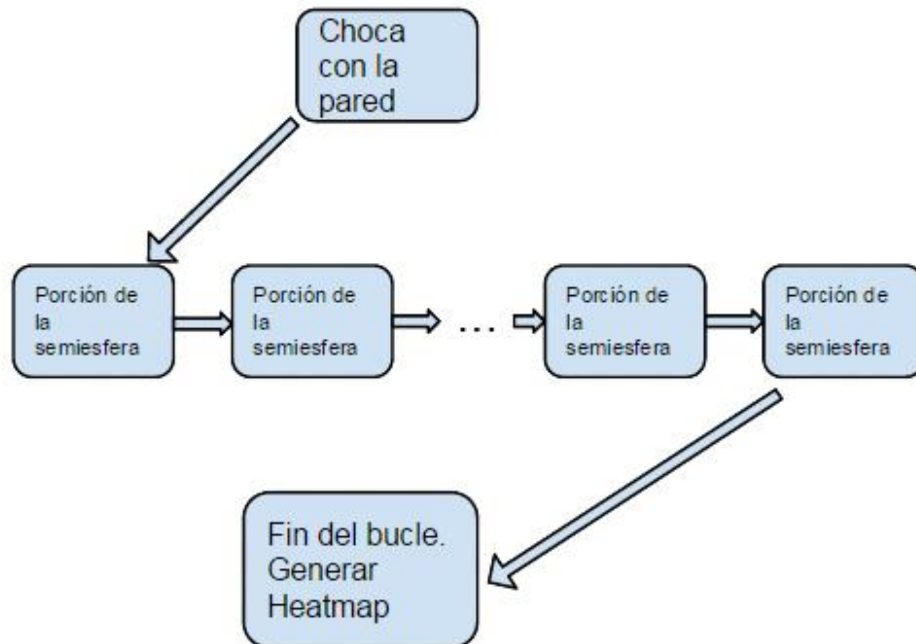

Ilustración 3.3. Esquema de ejecución del prototipo 5.

4. DISEÑO

En el diseño de mi TFG, he usado la herramienta Visual Paradigm 12.1 Community Edition para hacer los diagramas UML. Esta es una versión gratuita del software.

Para la modificación de valores por parte del usuario, he pensado agrupar los valores modificables en categorías para hacer más intuitivo el significado de cada variable y que el usuario pueda encontrar con facilidad la variable.

Físicas, agrupa las variables como la velocidad e intensidad de la luz, cantidad de partículas que se emiten, umbral de generación de nuevas partículas.

Cámara, esta categoría engloba tiempos de la cámara, tamaño o resolución de las imágenes, número de imágenes que se van a generar.

Escena, aquí enlazaremos los objetos importantes de la escena para que el simulador sepa donde está el láser y donde está la pared a la que tiene que apuntar.

Otros valores, en esta sección especificamos otros tipos de variables que no pertenecen a las otras categorías.

Estos valores se almacenarán de forma **static** en el simulador para que cualquier script pueda consultar y usar estos valores. Esta decisión hará que el simulador sea menos sensible a modificaciones en el futuro.

Dotar la escena con un botón de control para empezar y pausar la simulación. Añadir un botón para cambiar de punto de vista de la escena para que el usuario vea desde diferentes perspectivas la escena.

Mostrar la información en tiempo real la cantidad de partículas que están colisionando con la pared difusora, con el objeto oculto y las que son captadas por la cámara.

4.1. Escena

La escena ha sido diseñada a semejanza de la del experimento del artículo [Velten2012]. La mayoría de los elementos han sido modelados con cubos escalados en Unity, ilustración 4.1 y 4.2.

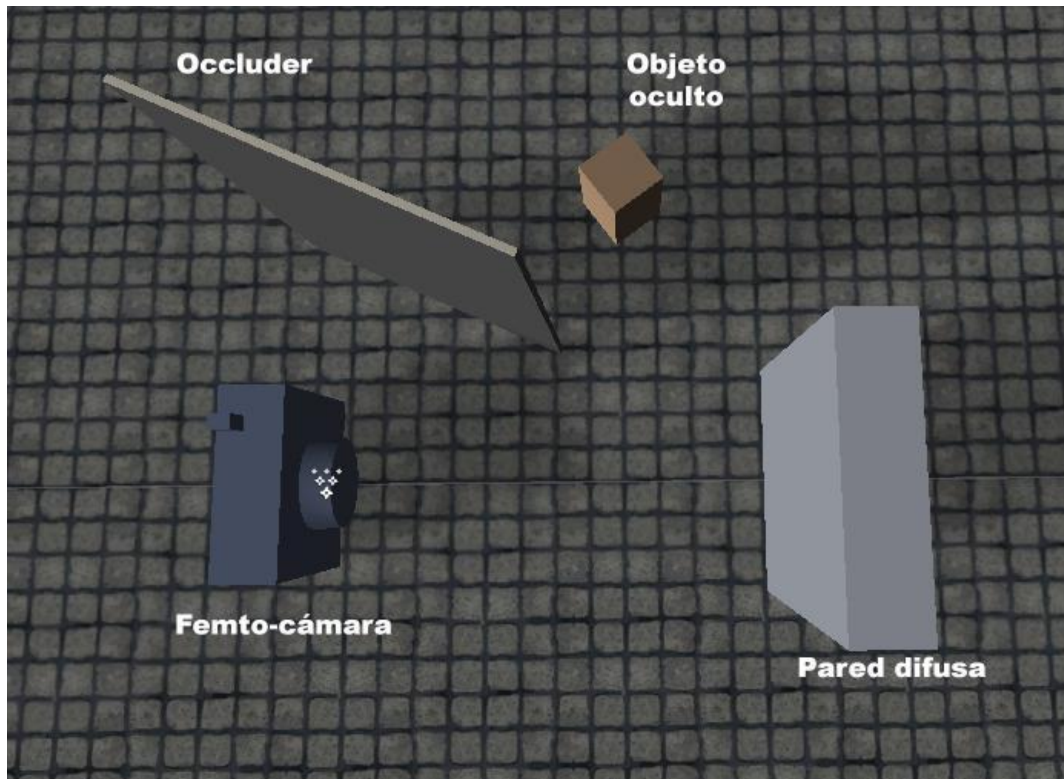

Ilustración 4.1. Escena.

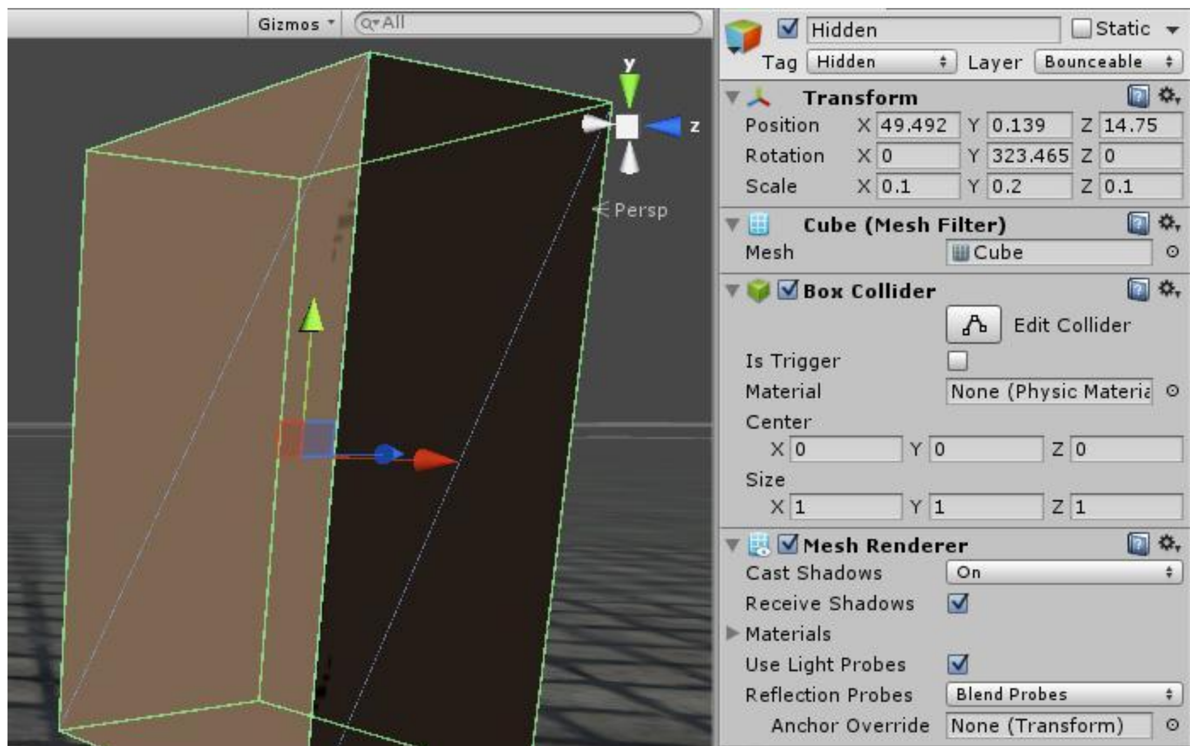

Ilustración 4.2. Objeto oculto modelado como cubo escalado.

4.2. Diagramas UML

A continuación se encuentran los diagramas que ayudarán a comprender el funcionamiento del simulador y la interacción entre sus clases o scripts. Todos los diagramas en formato jpg como el proyecto en Visual Paradigm se encuentran en la carpeta UML.

Diagrama de clases

La función de la clase **"Collision_particle"** es almacenar los datos de la colisión de una partícula con el detector de la femto-cámara. Esta clase se utilizará a la hora de generar el heatmap captado por la femto-cámara. Se guarda la posición tridimensional de el punto en el que ha sido detectado, realmente en la escena desarrollada sólo se utilizará la componente 'x' de este parámetro pero por si en un futuro se desea complicar la escena o el simulador será útil almacenar el punto entero. También se guarda el tiempo que ha pasado desde que la cámara está abierta grabando. La intensidad que tiene la partícula en el momento de la captura, para ello se accede al script adjunto al sistema de partículas llamado **"IntensityParticle"**. Esta clase extiende de la clase *System.ValueType*, esta herencia es necesaria para poder almacenar instancias en arrays del sistema y por lo tanto poder guardar un vector con todas las colisiones para posteriormente generar el heatmap. El script encargado de generar las instancias de las colisiones de las partículas es **"DectectorScript"**. Este se encuentra en el collider que detectará las colisiones de las partículas con la pared difusora.

"DifusserMaterial" es el script que simula el comportamiento de una superficie difusa. Se encuentra en la pared difusora lógicamente y cuando recibe una colisión genera una cantidad de partículas igual a uno de sus parámetros. La intensidad de estas nuevas partículas depende de la intensidad de la colisión entrante y de las propiedades de absorción y reflexión del script.

El script que se encarga de generar los mapas de calor (Heatmaps) es **"HeatmapGenerator"** al cual se le pasa el array de colisiones. Para generar las

tablas html de colisiones está **“CounterScript”** con variables estáticas para que cualquier otro script que necesite actualizar la información lo haga.

El Script principal, **“MainScript”**, es el encargado de que se siga el proceso correctamente y es el que se ocupa de activar, desactivar el detector y llamar a las clases necesarias. Controla los tiempos de ejecución, etc.

El script **“GUIController”** es el controlador de la interfaz de usuario y se encuentra en GameObject con todos los campos de la interfaz. Antes de que comience el algoritmo se actualizan todos los valores de las variables que se encuentran en **“Variables”**. **“Variables”** almacena todas las Variables importantes para el experimento y son accesibles desde cualquier otro script que necesite consultarlas.

Para acabar el script **“CameraSwitcher”** controla la cámara de la escena que está activa y permite cambiar entre una serie de cámaras.

Justo debajo en la ilustración 4.3 podemos ver el diagrama de clases UML. Y más abajo el diagrama de clases del prototipo 5 que incluye el script **“Dispersion”**, ilustración 4.4.

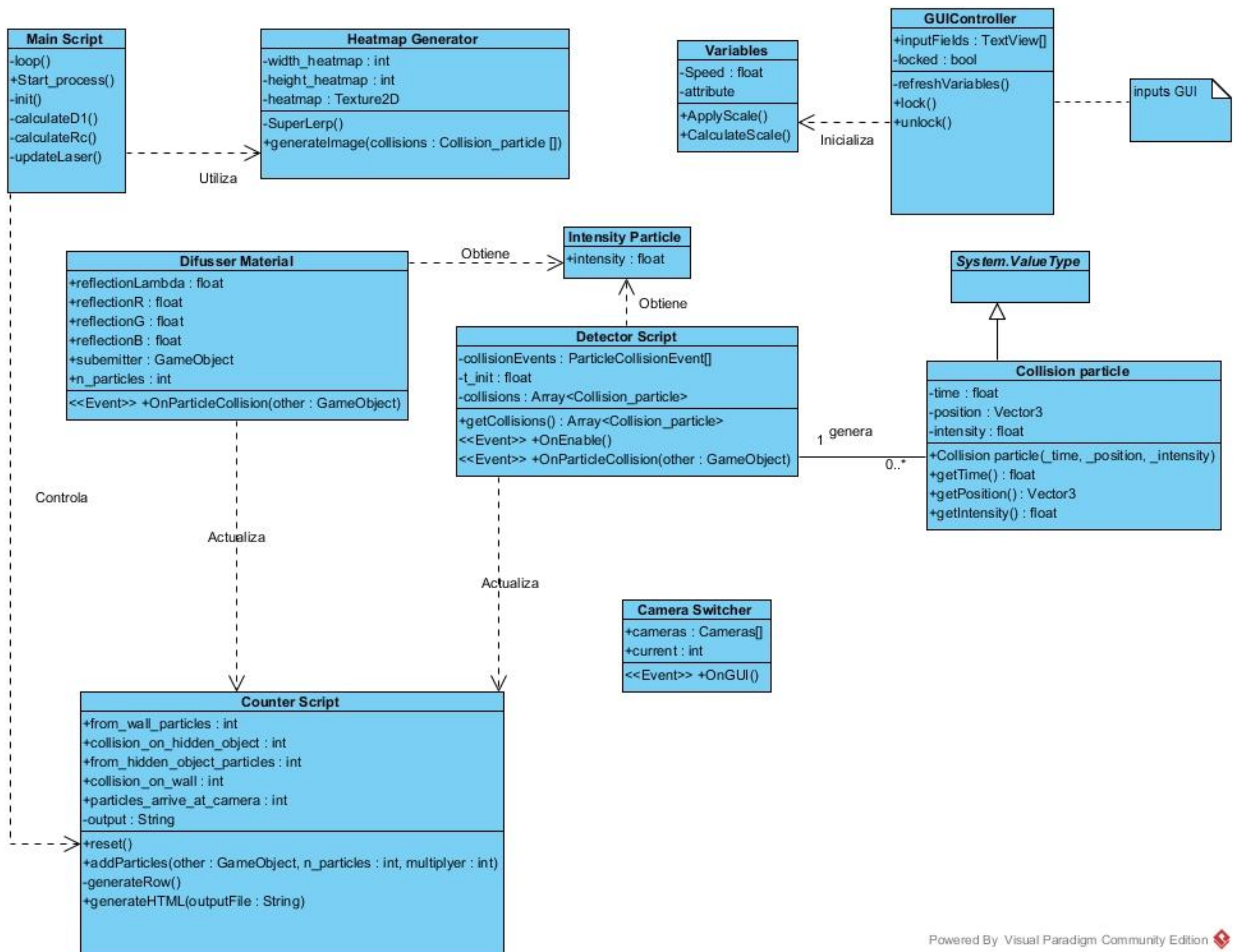

Ilustración 4.3. Diagrama de Clases y/o Scripts.

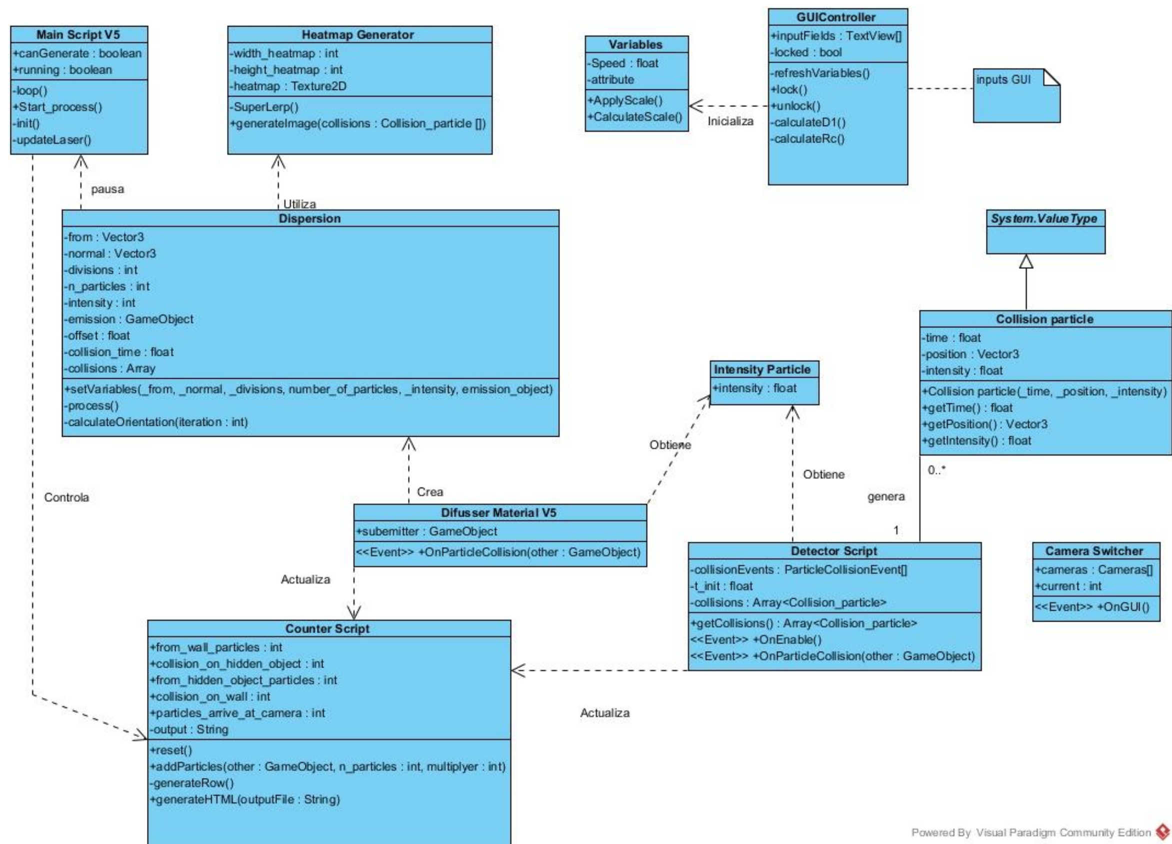

Ilustración 4.4. Diagrama de Clases del prototipo 5.

Diagrama de secuencia

El diagrama de secuencia muestra el orden en que los scripts son ejecutados y las interacciones entre unos y otros. Ver ilustración 4.5.

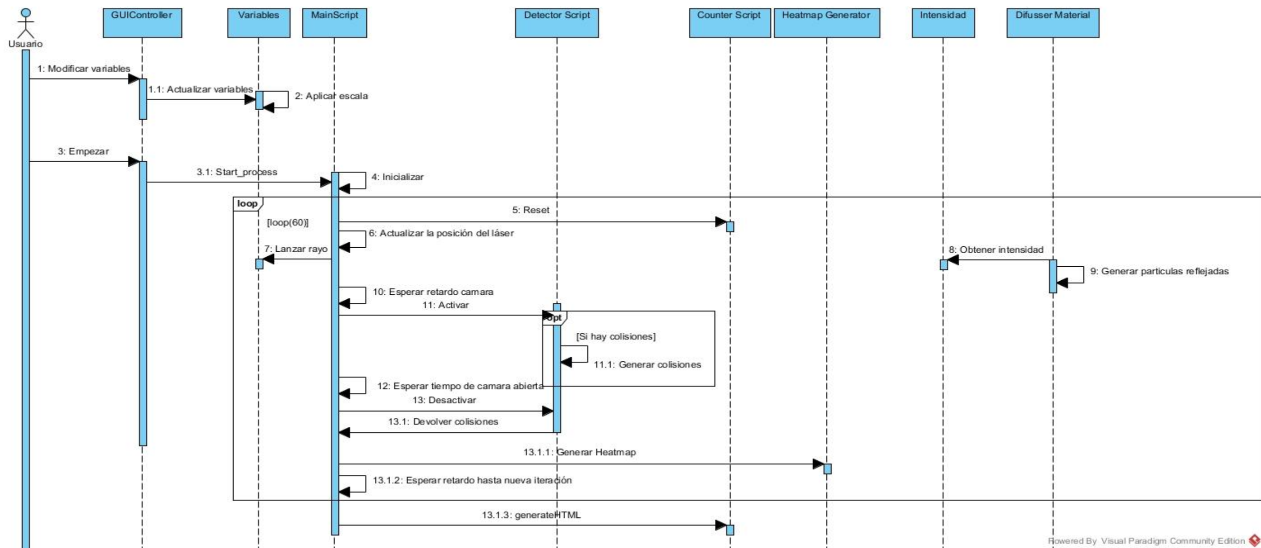

Ilustración 4.5. Diagrama de Secuencia.

Diagrama de tiempos

El diagrama de tiempos lo he centrado en el bucle principal para esclarecer el tiempo que consume cada iteración del bucle. Intervienen principalmente tres clases. El detector puede estar activo o inactivo. Cuando está activo es capaz de detectar las colisiones de las partículas. Y permanece activo después del retardo para activar la cámara hasta que pasa el tiempo que la cámara debería estar grabando/detectando. El Script principal siempre está ejecutándose. El generador de heatmaps solo trabaja a la hora de generar los mapas de calor.

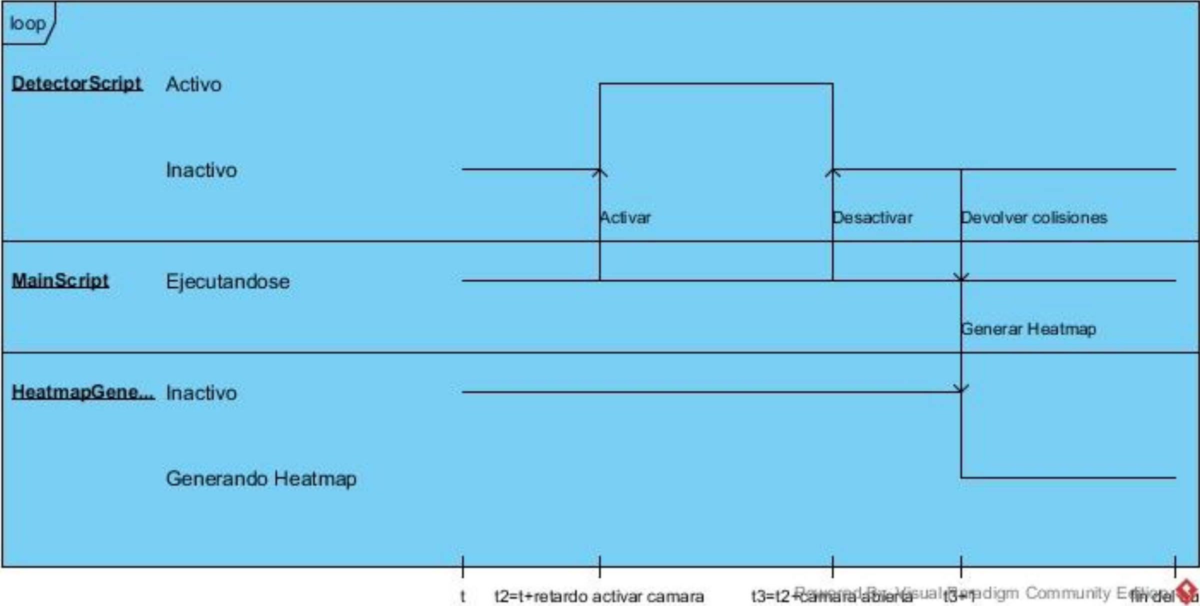

Ilustración 4.6. Diagrama de Tiempos.

Diagrama de máquina de estados

En el diagrama de máquina de estados en la ilustración 4.7 se pueden ver los estados por los que pasa el algoritmo y qué acciones encadenan cada estado.

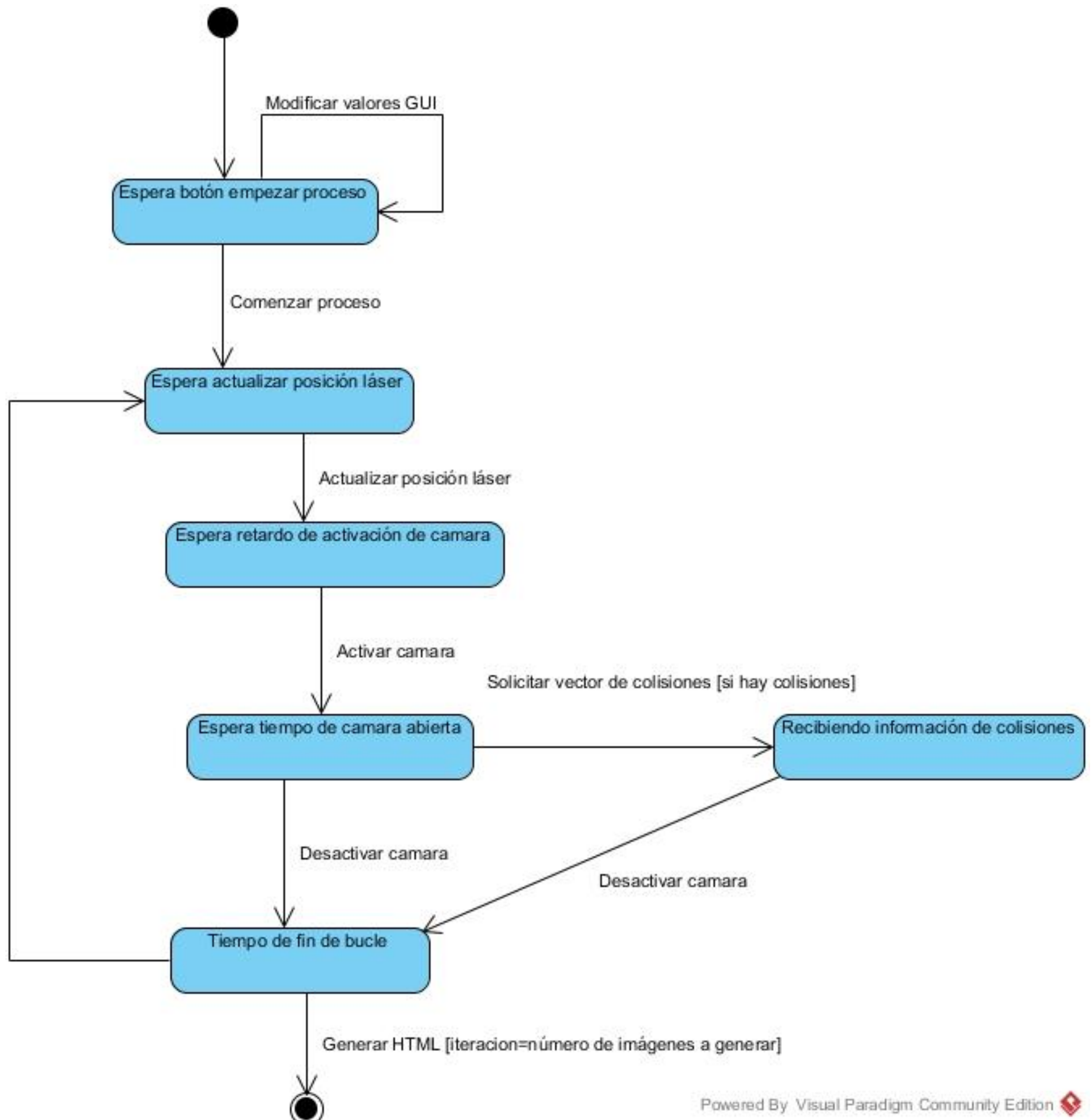

Ilustración 4.7. Diagrama de Máquina de Estados.

Diagrama de actividades

En este diagrama se muestran las actividades principales del algoritmo. Comienza con la actividad “Modificar valores”, que hace referencia a si el usuario modifica los valores del simulador. Calcula la escala necesaria según el tamaño deseado del heatmap para que el simulador sea fluido y aplica la escala. A partir de aquí comienza el algoritmo en sí.

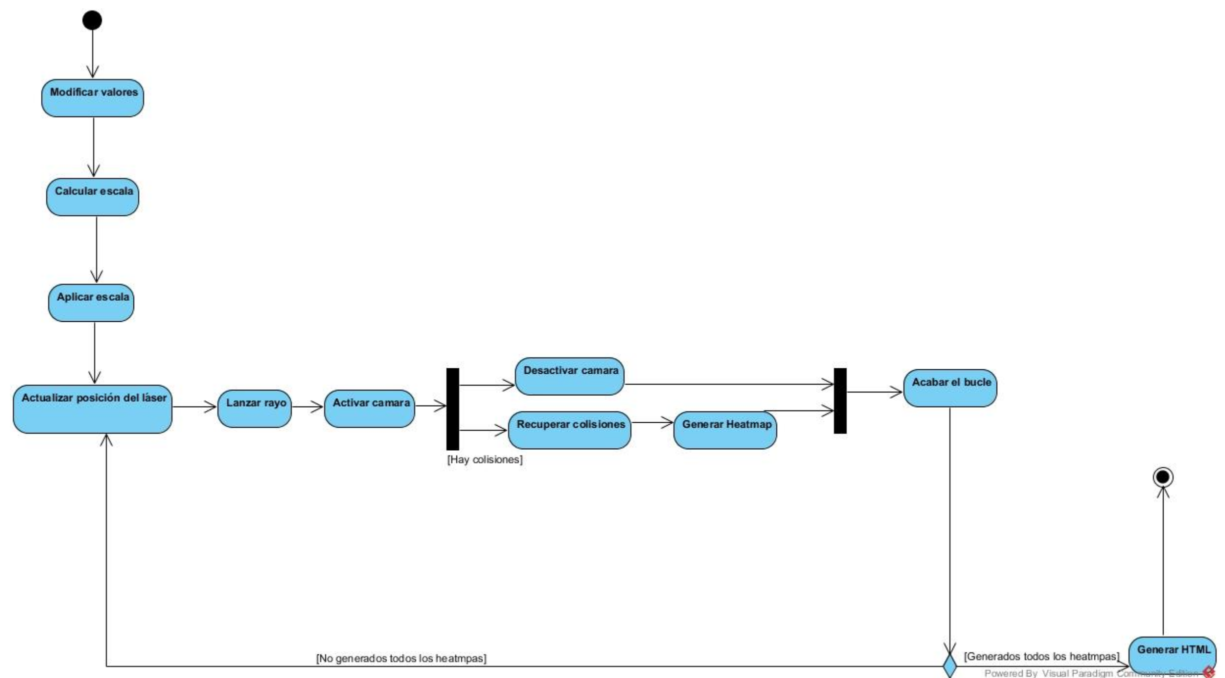

Ilustración 4.8. Diagrama de Actividades.

5. IMPLEMENTACIÓN

El simulador empezó ha ser implementado en el motor de videojuegos **Unity** en la versión 4.6. Cuando Unity sacó la versión 5 de su motor gráfico cambió nombres de clases referentes a la colisión de las partículas por lo tanto cuando yo lo actualicé el día 25 de marzo de 2015 el proyecto no compilaba ya que ahora las referencias a las clases actualizadas no concordaban. Tuve que reescribir algunos *Scripts* para que estos compilarán en la versión Unity 5 y seguir trabajando en el proyecto. La

clase que daba conflictos era **ParticleSystem.CollisionEvent** que era una subclase de ParticleSystem, esta dejó de ser una subclase y quedó en **ParticleCollisionEvent**. También cambiaron la forma de acceder a algunos de los valores de los sistemas de partículas.

Ilustración 5.1. Imagen promocional de Unity 5.

Detector

Para la detección de las partículas que llegaban a la cámara de barrido se planteaba el problema de que en el mundo virtual realmente se tenían que detectar antes de que llegaran a la cámara, ya que la cámara es solo un punto y lo que detecta está en su campo de visión. La solución tomada ha sido colocar un collider en la pared difusora con una anchura como el campo de visión de la cámara y una altura razonable para detectar las partículas, se puede ver en la ilustración 5.2. Por lo tanto el recorrido de las partículas es menor. En la ilustración 5.3 se observa la diferencias del viaje de las partículas en la teoría y en el simulador, la partícula es captada en el punto “p”.

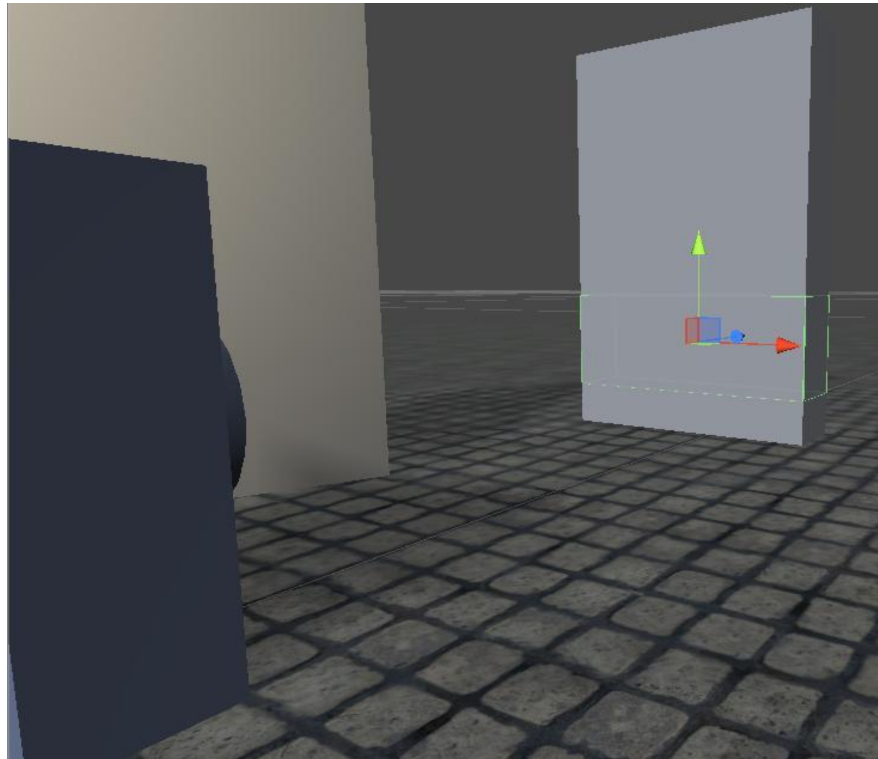

Ilustración 5.2. Collider del detector.

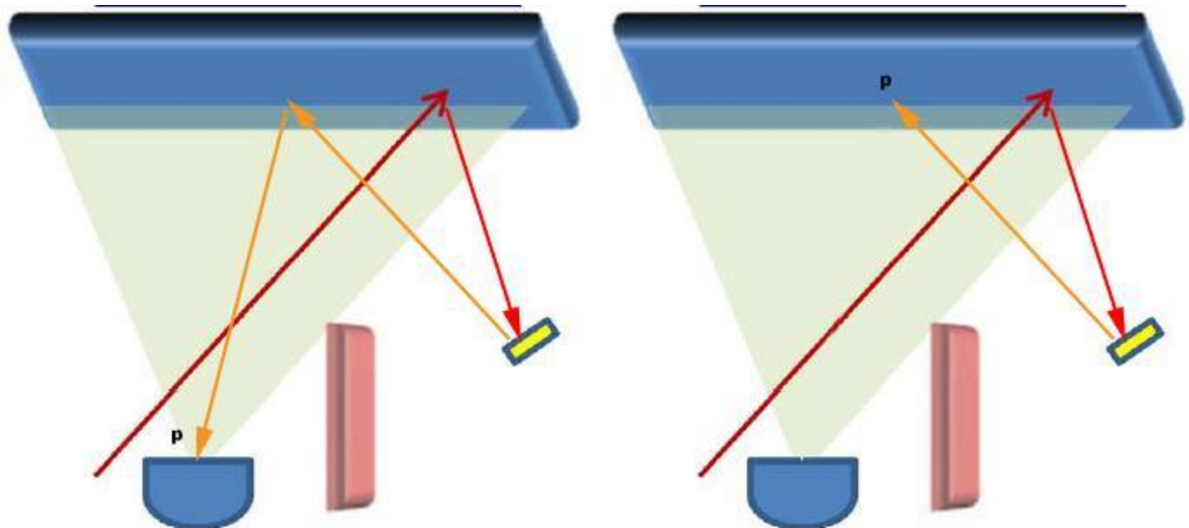

Ilustración 5.3. Ruta de la luz. Izquierda en la realidad y derecha en el simulador.

Pared difusora

Las posiciones del láser en la pared difusora tiene un gran impacto para el rendimiento del algoritmo de reconstrucción. Para obtener buenos resultados, se deberían elegir posiciones para proveer de una buena diversidad angular.

La dirección del láser se calcula automáticamente para que apunte en un punto exacto de la pared difusora dependiendo de la iteración actual. En la primera versión, las posiciones a las que apunta el láser se pueden observar en la ilustración 5.4, son 10 posiciones en el eje de abscisas(x) y 6 en el eje de ordenadas(y) dando un total de 60 posiciones que se asociarán a 60 imágenes.

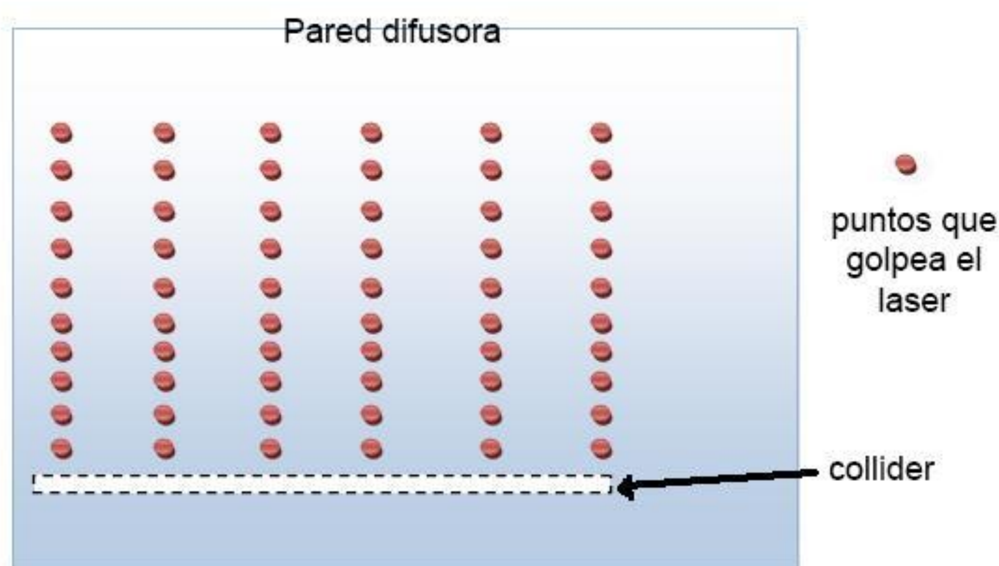

Ilustración 5.4. Posiciones a las que apuntará el láser. Versión 1.

Según el artículo del experimento se usan en torno de 3 a 5 columnas verticales a las cuales se apuntará el láser. Así que modifiqué el código para que el usuario pueda elegir el número de columnas. Como se puede ver en la ilustración 5.5 las intersecciones en el *Grid*(líneas rojas discontinuas) serán las posiciones a las que apuntará el láser. La posición cambia en cada iteración empezando por la esquina inferior izquierda, continúa hacia la derecha y luego vuelve a la izquierda pero una fila por encima. El recorrido queda (Fila-Columna) 0-0, 0-1, 0-2, 0-3, 0-4, 1-

0, 1-1, 1-2, 1-3, 1-4, 2-0, …, 11-4. Este modelo es el que seguí a partir a partir del prototipo 3.

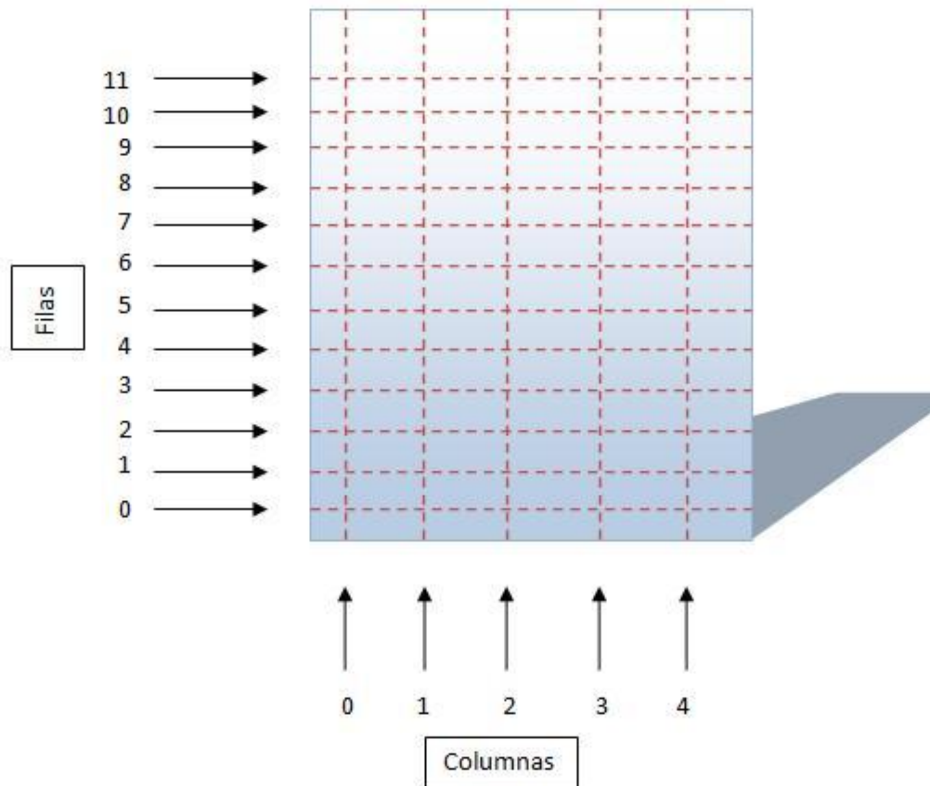

Ilustración 5.5. Posiciones a las que apuntará el láser. Versión 2.

La posición se calcula así:

X = Width/max_c*current_c

Y = Height/max_r*current_r

Donde:

X, Y = punto donde apuntar

Width = Ancho del muro

Height = Alto del muro

max_c = número de columnas

max_r = número de filas

current_c = columna actual

current_r = fila actual

(Fórmula 5.1)

El problema del bajo framerate.

Para hacer pruebas con el simulador es importante que la escena virtual sea fiel a la escena en la que se usó la femto-cámara para así poder determinar si los resultados del simulador son más o menos reales.

Los valores del experimento son números exageradamente altos o exageradamente bajos, la velocidad de la luz, la frecuencia del láser, el tiempo que la femto-cámara permanece grabando, etc. ver tabla 1.1. Para poder trabajar con estos números en el simulador han sido escalados dejando así una simulación a "cámara lenta" respecto a la original. En el experimento real la velocidad de los fotones era la velocidad de la luz ($3x10^8$ m/s) y la cámara grababa un intervalo de 1 ns (10^{-9} s), esto significaba que los fotones recorrían una distancia de 30 centímetros en el intervalo en el que la cámara grababa. Si escalamos la velocidad y suponemos una velocidad de las partículas de 0,3 m/s , es decir una escala de 10^{-9}, la partícula recorrerá los 30 centímetros que recorría en la realidad en un segundo. Esta escala simplifica mucho los grandes valores de las constantes del experimento. Por lo tanto quedaría una velocidad de 0,3 metros por segundo, un intervalo en el que graba la cámara de un segundo y la frecuencia con la que el láser envía pulsos de luz quedaría en 13,3 segundos.

El factor de escala se multiplica por las magnitudes de velocidad y se divide por las magnitudes temporales.

Variable	Valor real	Valor simulado
Velocidad de las partículas	$3x10^8$ m/s	0,3 m/s
Tiempo de cámara abierta	10^{-9} s	1 s
Intervalo entre	$13,3x10^{-9}$	13,3 s

pulsos del láser	s	
Intervalo de digitalización	$2x10^{-12}$ s	0,002 s

Tabla 5.1. Tabla de los valores del experimento aplicando una escala de 10-9.

Autoescala

El proceso de adquisición de datos puede ser muy costoso computacionalmente así que la escala del sistema es muy importante para obtener buenos resultados. Para automatizar esta escalada hay que tener en cuenta dos cosas distintas.

Autoescala según la potencia de cómputo

Una simulación a la misma escala en dos computadores con distintas prestaciones tendrán resultados muy distintos. Posiblemente en el de menor potencia se salte colisiones ya que el tiempo entre frames, desde ahora llamado Δ time (delta time), será mayor y quizás en el de mayor potencia pueda hacer la simulación más rápido. Sería interesante hacer que la velocidad de la simulación se adaptará a la potencia del computador en el que se ejecuta. De esta forma se tendrían resultados fiables independientemente de la potencia de cómputo. Para saber la potencia de cómputo se puede utilizar la variable delta time, si es muy pequeña el ordenador es rápido y la escala será menor, si es un valor grande la escala deberá ser mayor para que procese cada partícula con precisión. Un algoritmo de auto escala podría ser: antes de iniciar el proceso lanzar una función costosa y calcular la media de framerate. Si la media es sobre 60 fps entonces la escala se puede aumentar con valores más rápidos, si sale un bajo framerate disminuir los valores. No he llegado a implementar esta función en el simulador ya que finalmente la escala se puede realizar manualmente, pero sería una buena funcionalidad para implementar en el futuro.

Autoescala según variables

Si los valores de la simulación son los de la tabla 5.1 y el intervalo de digitalización es de 0,002 s la velocidad de la simulación debe ir a fotograma por cada 0,002 s esto es equivalente a 1/0,002 = 500 fps. Si el tiempo que permanece la cámara abierta es de 2 s el intervalo de digitalización es 0,004 s, esto da 250 fps. Esto es inviable ya que la velocidad máxima suele llegar a 60 fps sin contar la caída de framerate que produce el cómputo de las partículas (Autoescala según la potencia de cómputo) y el rendering de la imágen.La forma de solucionar esto es escalar aún más el sistema o aumentar el intervalo de digitalización, que al fin y al cabo significa reducir la altura del heatmap.

En la tabla 5.2 se puede ver los FPS necesarios para poder captar el heatmap con total precisión según el tamaño en altura del mismo y el tiempo que la cámara permanece abierta.

Tiempo de cámara abierta	FPS necesarios según tamaño del heatmap en Y (Δ time)		
	500 píxeles	256 píxeles	128 píxeles
1 ns	(2 ps) $5{\cdot}10^{11}$ fps	(3,90625 ps) $2{,}56{\cdot}10^{11}$ fps	(7,8125 ps) $1{,}28{\cdot}10^{11}$ fps
1 s	(0,002 s) 500 fps	(0,00390625 s) 256 fps	(0,0078125 s) 128 fps
2 s	(0,004 s) 250 fps	(0,0078125 s) 128 fps	(0,015625 s) 64 fps
3 s	(0,006 s) 166,66 fps	(0,01171875 s) 85,33 fps	(0,0234375 s) 42,66 fps
4 s	(0,008 s) 125 fps	(0,015625 s) 64 fps	(0,03125 s) 32 fps
5 s	(0,01 s) 100 fps	(0,01953125 s) 51,2 fps	(0,0390625 s) 25,6 fps
6 s	(0,012 s) 83,33 fps	(0,0234375 s) 42,66 fps	(0,046875 s) 21,33 fps
7 s	(0,014 s) 71,43 fps	(0,02734375 s) 36,57 fps	(0,0546875 s) 18,29 fps
8 s	(0,016 s) 62,5 fps	(0,03125 s) 32 fps	(0,0625 s) 16 fps
9 s	(0,018 s) 55,55 fps	(0,03515625 s) 28,44 fps	(0,0703125 s) 14,22 fps

10 s	(0,02 s) 50 fps	(0,0390625 s) 25,6 fps	(0,078125 s) 12,8 fps

Tabla 5.2. Tabla de los FPS necesarios según tamaño de imagen y tiempo de cámara abierta.

Para calcular esta escala habría que dividir la resolución del alto de la imagen entre el tiempo que la cámara graba y así conoceríamos los fps que necesitamos para que el simulador detecte las colisiones sin saltos. Este “fps_need” es un número muy alto, superior al límite de framerate del PC por ello se divide entre “FPS_max”, que es el framerate al que suponemos trabajará el simulador, normalmente 60 fps.

fps_need = Height_heatmap / Open_camera_time;

scale = fps_need / FPS_max;

Donde:

fps_need, framerate necesario para que la simulación vaya fluida según los valores de la cámara.

Height_heatmap, número de píxeles de alto del heatmap

Open_camera_time, tiempo que permanece la cámara grabando

scale, escala final

FPS_max, framerate límite (usualmente 60 fps)

(Fórmula 5.2)

Cuando el framerate es demasiado bajo el delta time será alto, esto implica que varias colisiones se computarán en un mismo instante y por lo tanto se perderá precisión en el eje Y del heatmap provocando líneas horizontales o “saltos”. En la ilustración 5.6 se puede ver las diferencias en los resultados que se generan con un un Δ time alto y uno ideal.

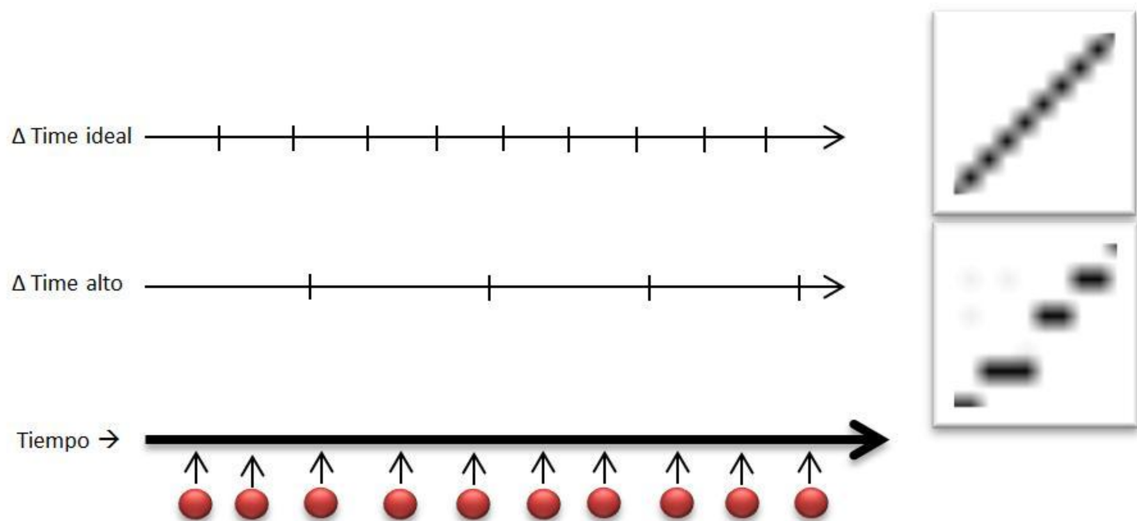

Ilustración 5.6. Diferencias de las soluciones dependiendo del delta time.

Calcular retardo de activación de cámara.

En la bibliografía sobre el experimento no he encontrado ninguna información acerca del retardo con el que la cámara se activa una vez se ha lanzado el rayo. Así que he estimado este tiempo de espera como el tiempo que tarda un partícula de luz en salir desde el láser, golpear la pared, rebotar en el objeto oculto suponiendo que está a una distancia igual a la desde el láser hasta la pared difusora.

Cálculo del delay de activación de la cámara:

$$Rc = Rc1 + Rc2$$

$$Rc1 = c/d1$$

$$Rc2 = c/d2$$

$$Rc1 = Rc2, si\ d1 = d2, entonces\ Rc = 2 * Rc1$$

Donde:

Rc, es el retardo de activación

Rc_1, tiempo que tarda una partícula en llegar a la pared desde el láser

Rc_2, tiempo que tarda una partícula en llegar a la pared desde el objeto oculto

c, velocidad de la luz

d1, distancia entre el láser y la pared difusa

d2, supuesta distancia entre el láser y la pared difusa

(Fórmula 5.3)

Corrutinas

Cuando se llama a una función, esta se ejecuta en su totalidad en ese frame. Esto significa efectivamente que cualquier acción dentro de una función debe suceder en una sola actualización de frame; una llamada a una función no puede ser usada para contener una animación procedimental o **una secuencia de eventos en el tiempo**.

En Unity una **coroutine** (corrutina) es como una función que tiene la habilidad de pausar su ejecución y devolver el control a Unity para luego continuar donde dejó en el siguiente frame.

Para implementar las esperas de tiempos con JavaScript en Unity, se lanzan como **corrutinas** con la función **StartCoroutine()**; en esta función se llamará al método **WaitForSeconds** donde le pasamos el tiempo en segundos como una variable flotante y pausa la ejecución de la función esa cantidad de segundos. Al ser una corrutina puede esperar sin bloquear los demás procesos. Se utiliza por ejemplo en el Script “Dispersion” donde es necesario hacer esperas y medir tiempos, ilustración 5.7.

```
    offset=180/divisions;

    heatmapGen = new HeatmapGenerator(Variables.Width_heatmap, Variables.Height_heatmap);

    StartCoroutine("process");
}

private var collisions = new Array();

private var heatmapGen : HeatmapGenerator;

private function process(){

    for(var i : int = 0; i<divisions*divisions; i++){

        ● ● ●

        //Delay
        yield WaitForSeconds(Variables.Delay_camera - collision_time);

        //Enable detector
        Variables.Detector.gameObject.SetActive(true);

        //Wait open camera time
        yield WaitForSeconds(Variables.Open_camera_time);

        //Disable camera
        Variables.Detector.gameObject.SetActive(false);

        //Get collisions
        collisions = collisions.concat(Variables.Detector.GetComponent(DetectorScript).getCollisions());

        //Wait end loop
        yield WaitForSeconds(Variables.Frecuency - Variables.Open_camera_time - (Variables.Delay_camera - coll
    }

    (GameObject.Find("Femtocamara").GetComponent(MainScriptV5) as MainScriptV5).canGenerate = true;
    heatmapGen.generateImage(collisions);
    Destroy(this);
}
```

Ilustración 5.7. Uso de las corrutinas en Unity con JavaScript.

Femto-cámara

La femto-cámara es un GameObject que tiene dos hijos: el láser y el modelo de la cámara. Mientras que la cámara es un mero modelo 3D hecho en unity con dos cubos y un cilindro para darle aspecto visual. En la femto-cámara se encuentra el “MainScript” que se encarga de rotar el láser en cada iteración.

Ilustración 5.8. GameObject de la femto-cámara.

Sistemas de partículas

La luz se ha simulado con sistemas de partículas Shuriken de Unity. Estos sistemas de partículas tienen el script asociado “IntensityParticle” para almacenar la intensidad que lleva la luz. Ha sido necesario crear dos sistemas de partículas: el primero que es el que va desde el láser hasta la pared difusa y el segundo que es la luz que se propaga difusamente.

En la ilustración 5.9, podemos ver la configuración del primer sistema de partículas. Tan solo enviará una partícula en línea recta, por ello el valor **Max Particles** es igual a 1 y el campo **Shape** es un cono con un radio ínfimo.

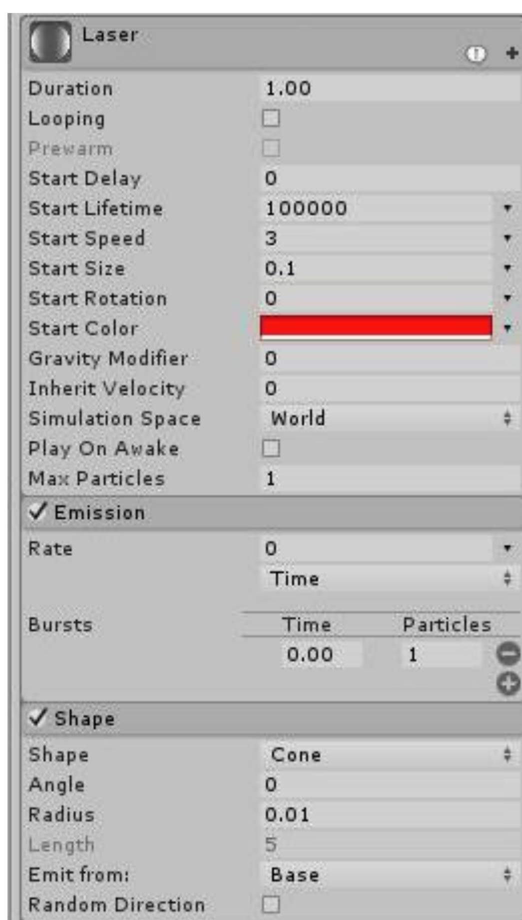

Ilustración 5.9. Uso de las corrutinas en Unity con JavaScript.

En la ilustración 5.10 aparece el segundo sistema de partículas que se instancia después de que el primero choque contra la pared difusa. Desde los prototipo del 1 al 3 se utiliza el sistema de partículas de la ilustración 5.10 izquierda, este genera las partículas como una semiesfera. El de la derecha las genera como un cono y se utiliza en los prototipos 4 y 5.

Ilustración 5.10. Sistemas de partículas instanciados al chocar con la pared difusora.

El segundo sistema de partículas se modifican los valores vía scripting según sea necesario en la escena, como se puede ver en la ilustración 5.11.

```
//Through ray
var reaction = Instantiate(emission, from+0.01*Vector3.back, Quaternion.LookRotation(normal))
reaction.transform.rotation *= quat;
reaction.GetComponent(ParticleSystem).GetComponent(IntensityParticle).intensity = intensity;
(reaction.GetComponent(ParticleSystem) as ParticleSystem).maxParticles = n_particles;
(reaction.GetComponent(ParticleSystem) as ParticleSystem).startSpeed = Variables.Speed;
var so : SerializedObject = new SerializedObject(reaction.GetComponent(ParticleSystem));
if(offset == 180){
    so.FindProperty("ShapeModule.type").intValue = 2; //Hemisphere
}
so.FindProperty("ShapeModule.angle").floatValue = offset/2;
so.ApplyModifiedProperties();
(reaction.GetComponent(ParticleSystem) as ParticleSystem).Emit(n_particles);
```

Ilustración 5.11. Cambiar propiedades de los sistemas de partículas vía Scripting en el prototipo 5.

Prototipo 1.

La implementación consiste en añadir como componente el script “DiffuserMaterial” tanto a la pared difusora como al objeto oculto, para cada vez que una partícula golpee uno de estos objetos se instancie un sistema de partículas con una propagación en semiesfera, simulando la reflexión difusa.

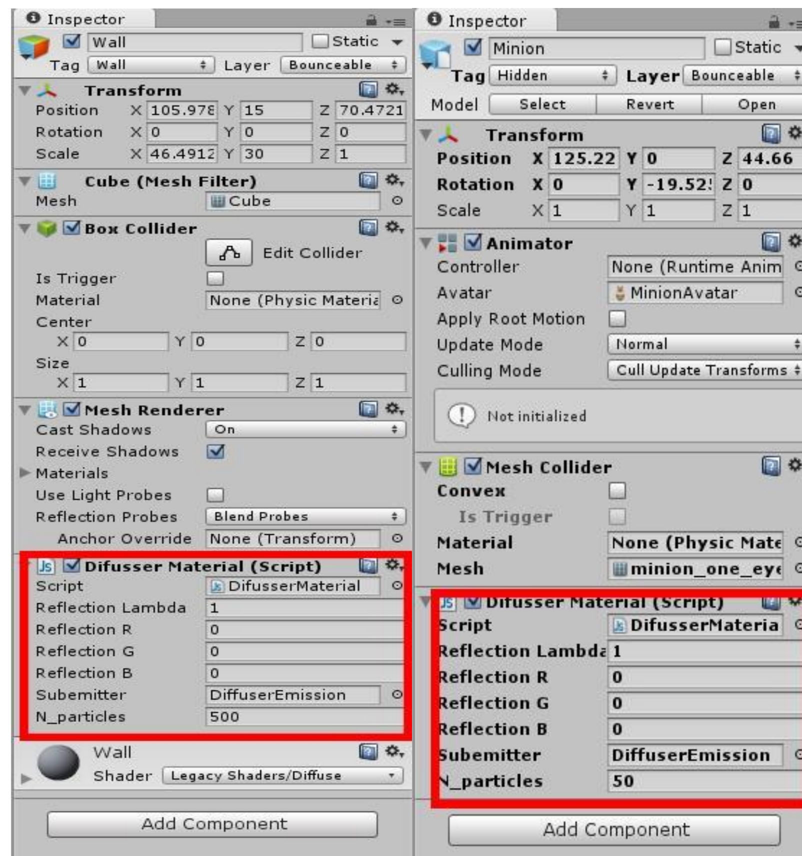

Ilustración 5.12. Pared difusora a la izquierda y objeto oculto a la derecha. Ambos tienen “DiffuserMaterial”, actúan como elementos difusos.

Prototipo 2.

Ahora el tiempo en el que la cámara graba si es el correcto. Para ello solo ha sido necesario crear una variable “Detector_interval” con un valor por defecto proporcional al que tiene en el experimento real. También puede modificarse al gusto del usuario. En esta versión esta variable se encuentra en el script “MainScript” en la última pasará al script “GUIController”.

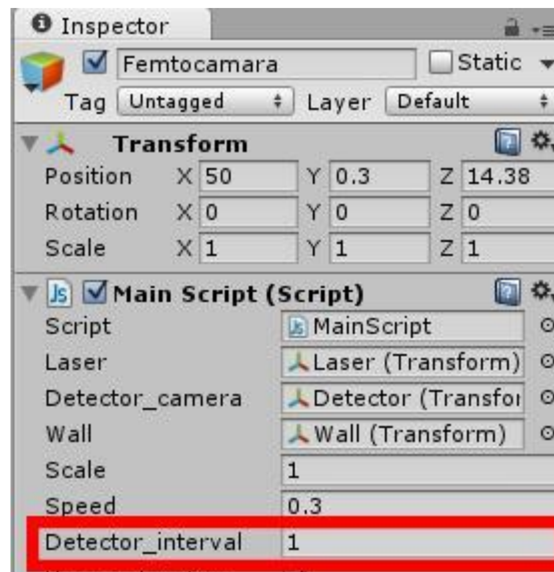

Ilustración 5.13. Variable de tiempo que permanece la cámara grabando.

Prototipo 3.

Implementamos el modelo de Pattanaik, es tan sencillo con quitar el script “DiffuserMaterial” del objeto oculto o aumentar el epsilon para que no se instancien más partículas. Y definir un número constante de partículas que se introduce en el “DiffuserMaterial” de la pared.

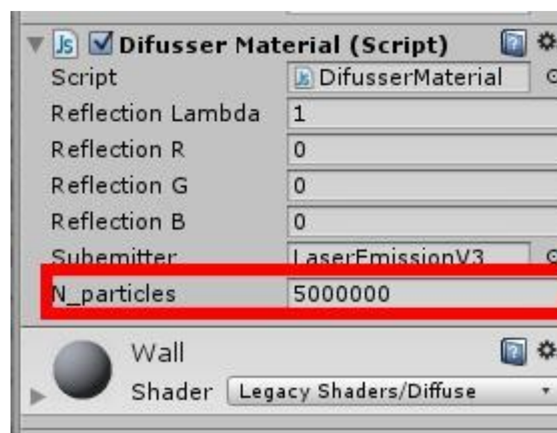

Ilustración 5.14. Variable de número de partículas que se instanciarán.

Prototipo 4.

Para esta variación del prototipo anterior, hacemos que la luz difusa enfoque al objeto oculto en un ángulo mucho más reducido. Para ello el script debe conocer dónde está el objeto oculto aunque no debería. Definimos un ángulo del cono que ocupará las partículas que emitiremos. Y modificamos el sistema de partículas para que la emisión sea un cono en vez de una semiesfera.

```
if(intensity > Variables.Epsilon){
    var reaction = Instantiate(subemitter, pos+0.01*Vector3.back, Quatern
    reaction.transform.LookAt(Variables.Hidden_object);//V4
    reaction.GetComponent(ParticleSystem).GetComponent(IntensityParticle)
    (reaction.GetComponent(ParticleSystem) as ParticleSystem).maxParticle
    (reaction.GetComponent(ParticleSystem) as ParticleSystem).startSpeed
    //reaction.GetComponent(TimedObjectDestructor).timeOut = 1;
    (reaction.GetComponent(ParticleSystem) as ParticleSystem).Emit(n_part
}
```

Ilustración 5.15. Línea de código que dirige el sistema de partículas al objeto oculto.

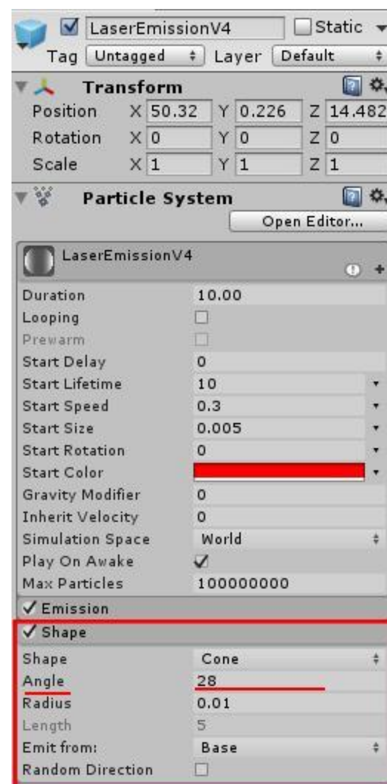

Ilustración 5.16. Propiedad forma del sistema de partículas puesta a cono.

Prototipo 5.

En este prototipo usamos la optimización del prototipo anterior pero de forma que sea genérico y el sistema no tenga que conocer dónde se encuentra el objeto oculto.

Esto ha sido implementado de tal forma que una vez que la luz choca contra la pared difusora el script asociado, “Diffuser material” que ahora se llama “DiffuserMaterialV5” (versión 5), añade un nuevo script llamado “Dispersion” como componente de la pared difusora. “Dispersion” almacena el punto de la pared de donde sale la luz de forma difusa, la normal de la pared y el tiempo en el que chocó la luz. Con la normal de la pared podríamos generar un sistemas de partículas que se dispersaran como una semiesfera, esto sería lo más realista y costoso si se lanzan muchas partículas. En cambio lo que hace el script es calcular porciones de la semiesfera como se muestra en la ilustración 5.17. Por cada porción estimada se lanzará un sistema de partículas que abarque toda este espacio. Lo ideal sería que se cubrieran todos los huecos sin solaparse pero realmente por facilidad he utilizado conos para simular estas porciones de semiesfera. Estos conos si se solapan pero realmente la información solapada no es falsa sino sólo redundante. El proceso consiste en un bucle que depende de las divisiones de la semiesfera al cuadrado y por cada iteración calcula la dirección que debe llevar el sistema de partículas y, también, calcula el ángulo del cono para abarcar el espacio correspondiente. Este script es ahora el encargado de activar y desactivar la cámara debidamente en vez del “MainScript” (ahora llamado “MainScriptV5”) como en las versiones anteriores, y por ello debe conocer el tiempo en el que se crea para calcular lo retardos. Almacena un vector de colisiones durante todo su ciclo de vida hasta que el bucle llega a su fin y entonces le pasa la información al script encargado de generar los heatmaps y se destruye a sí mismo.

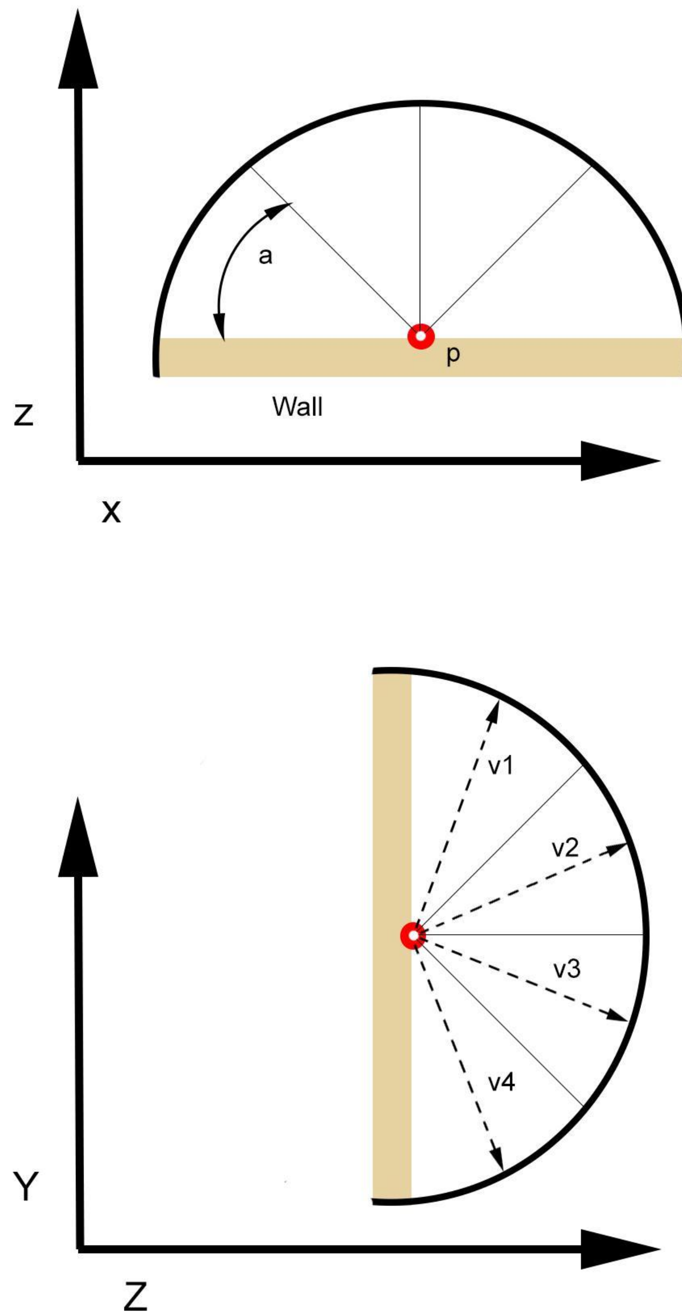

Ilustración 5.17. División de la semiesfera en porciones. Divisions = 4.

En la primera imagen de la ilustración 5.17, se ve la pared difusa desde arriba, el punto “p” es donde el láser golpea la pared. El semicírculo representa a la semiesfera que se generaría con la dispersión de la luz difusamente, la cual abarca

un ángulo de 180º. La semiesfera se divide entre el valor de la variable “divisions”, teniendo que “a” es el ángulo que debe abarcar cada porción, el cual se calcula como 180 grados entre “divisions”. En la segunda imagen, vista de perfil, se puede observar la misma división de la semiesfera. Cada porción equivale a un cono con un ángulo “a” y que se dirige con las direcciones v1, v2, v3 y v4 en cada iteración. Para tres dimensiones, el número de porciones de la semiesfera es el cuadrado de “divisions” ya que se divide a lo alto y a lo ancho por este número. Como se puede observar en la ilustración 5.18, más abajo, las porciones de la semiesfera realmente serían pirámides con vértice en el centro de la misma. Pero por facilidad de implementación he usado conos por cada porción de tal forma que abarquen todo el espacio sin dejar huecos y por lo tanto los conos adyacentes se solapan en los bordes.

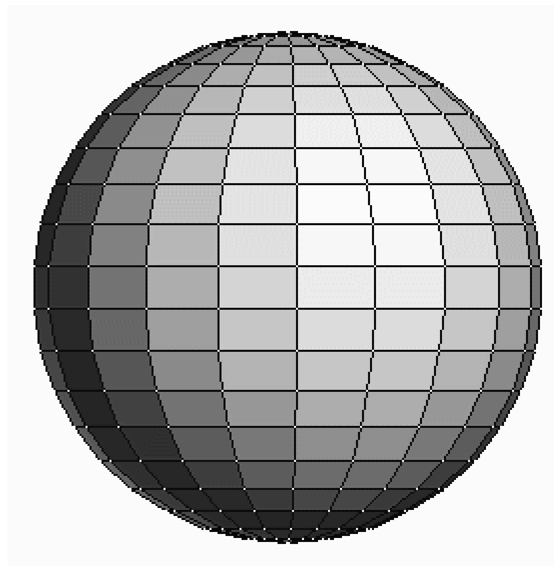

Ilustración 5.18. División de la semiesfera en pirámides.

Para calcular la dirección del cono de luz, he calculado los ángulos alfa respecto al eje X y beta respecto al Y. Las rotaciones en Unity son cuaterniones así que he usado la composición de cuaterniones para poner la dirección correspondiente.

Cálculo de la dirección del cono de luz:

```
offset = 180/divisions

alpha = offset/2 + (i%divisions)*offset;

beta = offset/2 + (i/divisions)*offset;


q1 : Quaternion = Quaternion.AngleAxis(90 - alpha, Vector3.right);

q2 : Quaternion = Quaternion.AngleAxis(90 - beta, Vector3.up);

result : Quaternion = q1*q2;
```

Donde:

offset, ángulo del cono

divisions, número de divisiones

i, iteración actual

alpha, ángulo de dirección respecto a X

beta, ángulo de dirección respecto a Y

q1, cuaternión asociado a alpha

q2, cuaternión asociado a beta

result, resultado de la composición de q1 y q2

(Fórmula 5.4)

Manual

El simulador no es un ejecutable, es simplemente un proyecto en Unity. He tomado esta decisión por la flexibilidad y facilidad para cambiar los elementos de la escena. La interfaz de usuario necesaria para el simulador requiere cierta complejidad y por lo tanto más tiempo de desarrollo del que no disponía. La interfaz ideal para el simulador debería permitir modificar los elementos de la escena, no solo la posición, rotación y escala sino las propiedades del comportamiento con la

luz de cada uno y poder modificar los valores de las variables y constantes de la simulación, tiempos, escala, etc.

Para empezar a trabajar con el simulador lo primero es abrirlo con Unity. Recordad que el simulador ha sido desarrollado en Unity 5. Una vez abierto tendremos que abrir la escena ya creada para la simulación, abrimos la escena ***Setup(v.5)*** situada en *Assets/Scenes*. Esta es la escena que corresponde al prototipo número 5, es decir, el válido. Las demás escenas son versiones anteriores que no son usables, solo están por información histórica.

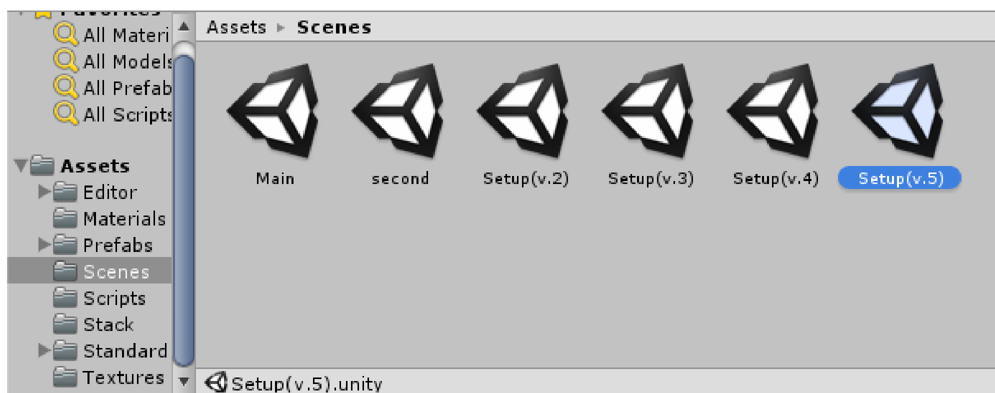

Ilustración 5.19. Ruta de la escena.

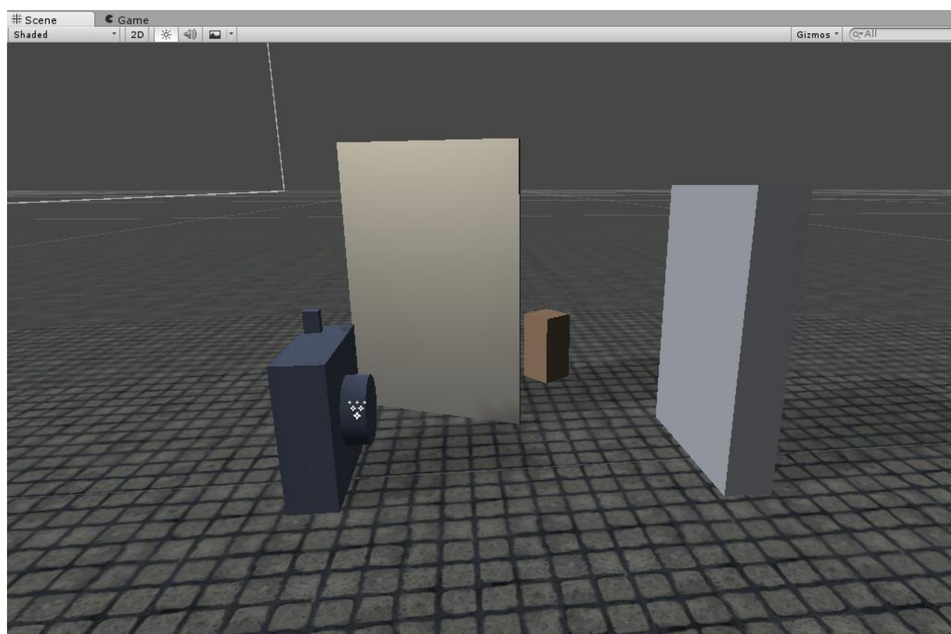

Ilustración 5.20. Escena de la simulación.

Se abrirá una escena como la de la imagen de arriba, donde podemos ver la femto-camera azul, el *occluder* es la pared justo al lado, detrás está el objeto oculto de un color marrón y a la derecha del todo se encuentra la pared difusa.

La escena está preparada para ejecutarse con los valores por defecto. Para cambiar estos valores hay que buscar el GameObject **VALUES** el cual contiene el script "GUIController" que es el encargado de modificar los valores de la simulación.

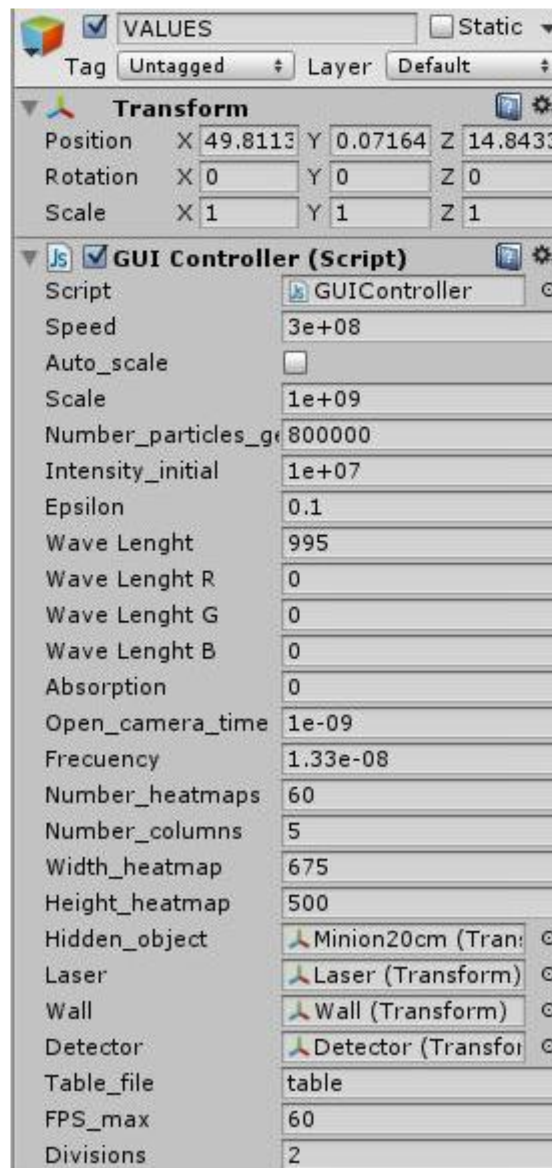

Ilustración 5.21. Variables modificables desde "GUI Controller".

A continuación explicaré los valores modificables del simulador desde el script. Estos valores se dividen en cuatro categorías:

- **Físicas**, valores asociados a las propiedades físicas de la luz, velocidad, etc.
  - **Speed**, velocidad de la luz.
  - **Auto_scale**, si está activada escalará la velocidad de la luz y los tiempos según los valores para que la simulación sea fluida.
  - **Scale**, especifica la escala de tiempos en el caso de que *Auto_scale* no esté activado.
  - **Number_particles_generated_on_collision**, indica el número total de partículas que se lanzan.
  - **Intensity_initial**, es la intensidad inicial de la luz. Esta se divide entre el número de partículas después de colisionar con el muro difuso.
  - **Epsilon**, es un umbral el cual limita la generación de nuevas partículas. Aconsejo dejarlo con un valor alto para que no se creen series que puedan colgar el simulador.
  - **Wave Lenght**, es la longitud de onda asociada a las partículas de luz que refleja la pared difusa. Se puede especificar por nanómetros o por componentes RGB, tiene un carácter de utilidad para ampliar el simulador.
  - **Absorption**, el índice de absorción indica la proporción de intensidad que absorbe la pared difusa.
- **Femto-cámara**, en esta categoría se pueden modificar valores referentes a la femto-cámara como tiempos, número de imágenes a generar, tamaño de las mismas, … .
  - **Open_camera_time**, tiempo que permanece la femto-cámara grabando.
  - **Frecuency**, es el intervalo de tiempo que pasa desde que se lanza el haz de luz láser hasta la siguiente vez que se lanza.
  - **Number_heatmaps**, número de imágenes de calor que va a generar el simulador. Es igual al número de iteraciones.

  - **Number_columns**, número de columnas a las que el láser apuntará, según el artículo *Velten* [Velten2012] es 3 o 5. Ver sección *Pared difusora*.
  - **Width_heatmap**, ancho de cada mapa de calor.
  - **Height_heatmap**, alto de las imágenes de calor. Este parámetro está íntimamente conectado al framerate ya que depende del tiempo.
- **Escena**, se especifica al simulador donde se encuentran los elementos.
  - **Hidden_object**, indica posición del objeto oculto. Este valor se utilizaba para versiones anteriores del simulador.
  - **Laser**, es el láser de la femto-cámara. Es necesario para que "MainScript" modifique la dirección a la que apunta.
  - **Detector**, es un collider que concuerda con el campo de visión de la femto-cámara. Se activa y se desactiva como si fuera la femto-cámara en sí.
- **Otros**.
  - **Table_file**, nombre del fichero que genera el simulador con las tablas con información del número de partículas que detecta la cámara y se lanzan. Se guarda en *html*.
  - **FPS_max**, es una variable para indicar el número ideal de frames por segundo para calcular la auto escala.
  - **Divisions**, es un valor de optimización del simulador que indica el número de divisiones de la semiesfera que se genera una vez choca la luz con la pared difusora.

Una vez tengamos hayamos modificado las variables al gusto tan solo tenemos que ejecutar la escena y veremos lo siguiente, ilustración 5.22.

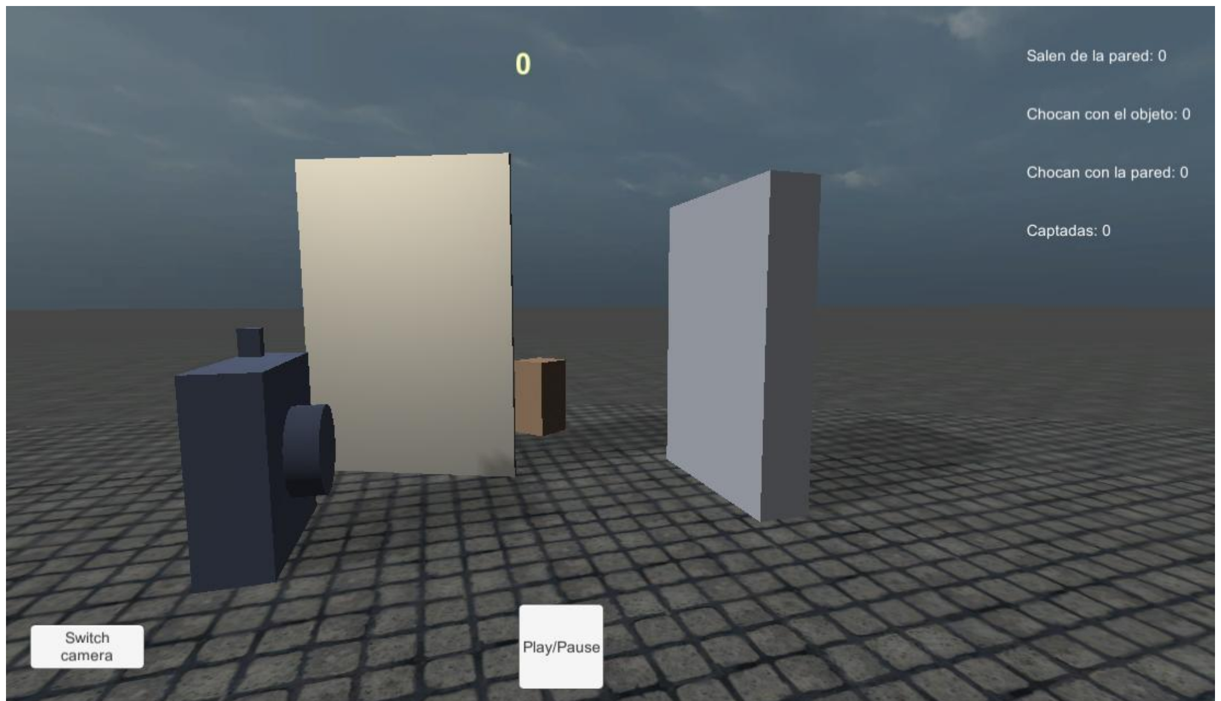

Ilustración 5.22. Escena pre-simulación.

En esta escena de pre-simulación podemos ver una poca de interfaz de usuario. El botón “Switch camera” para cambiar el punto de vista de la escena, ilustración 5.23. El botón “Start/Pause” para empezar a ejecutar la simulación, y si está en ejecución pausarla. El número de arriba es la iteración actual por la que va la simulación, ilustración 5.24. Y por último el texto a la izquierda de la pantalla indican por cada iteración las partículas que salen de la pared difusa, las que llegan al objeto oculto, las que vuelven a la pared y finalmente las que son captadas por la cámara, ilustración 5.25. Esta es la información que luego se exporta en forma de tabla html.

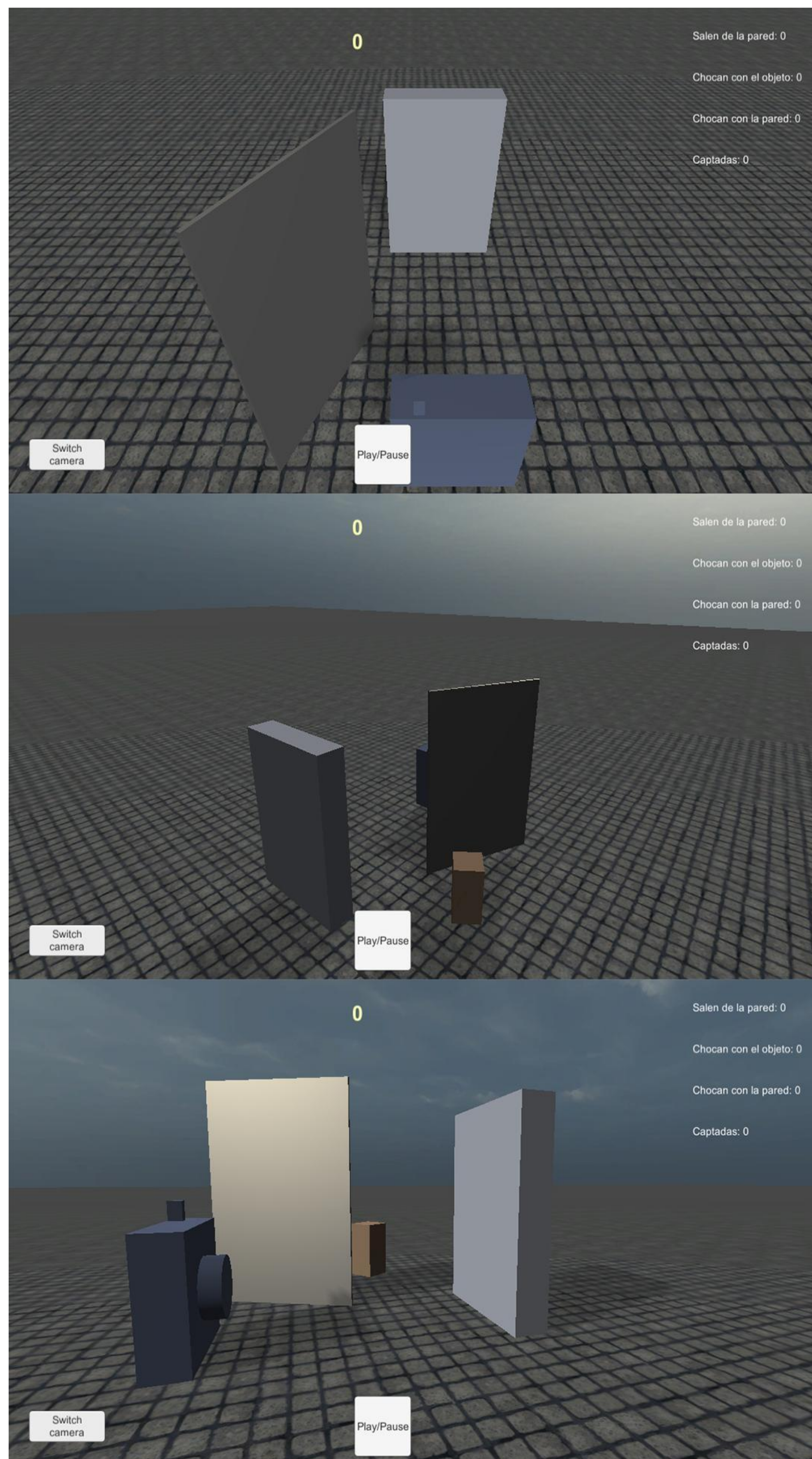

Ilustración 5.23. Distintos puntos de vista posibles.

Ilustración 5.24. Iteración actual y total de iteraciones.

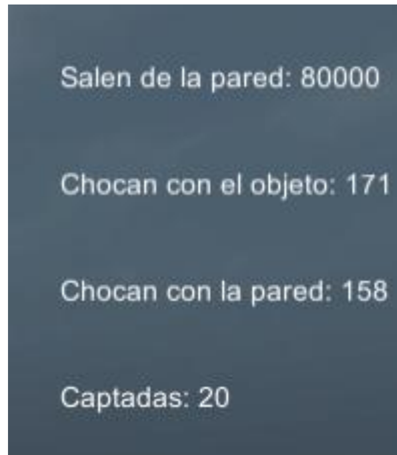

Ilustración 5.25. Datos de seguimiento de las partículas.

Cuando una iteración acabe se mostrará el Heatmap generado en la parte izquierda de la pantalla, como se puede ver en la ilustración 5.26.

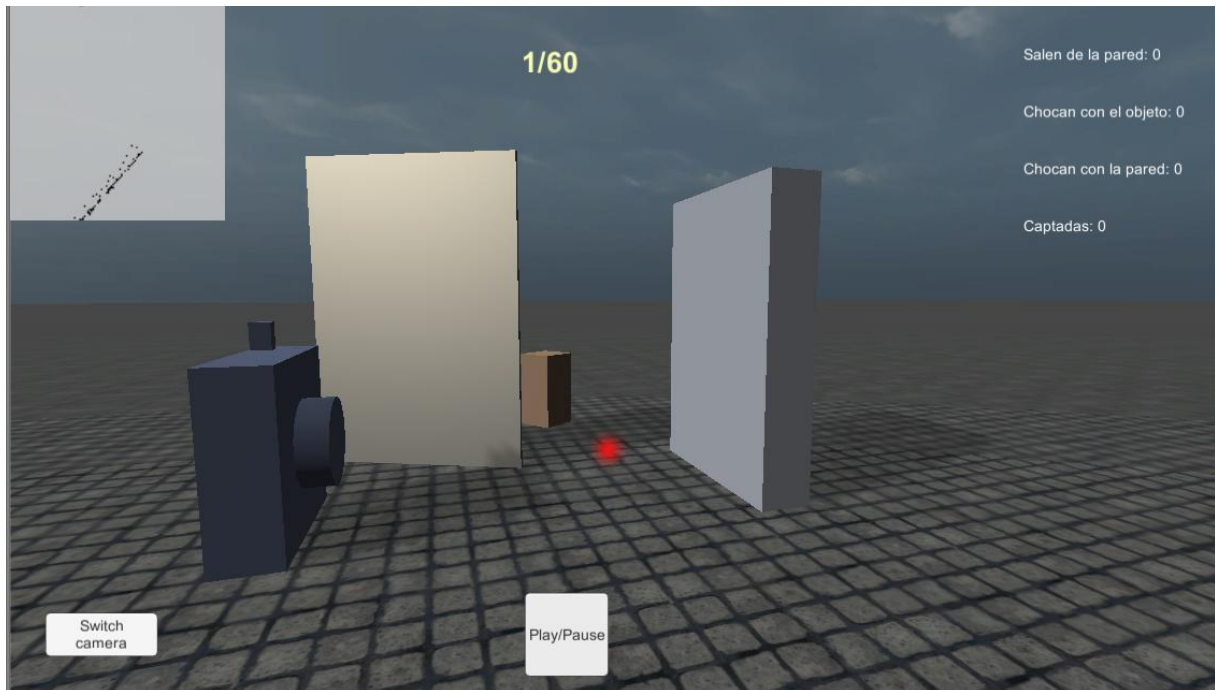

Ilustración 5.26. Escena mostrando Heatmap.

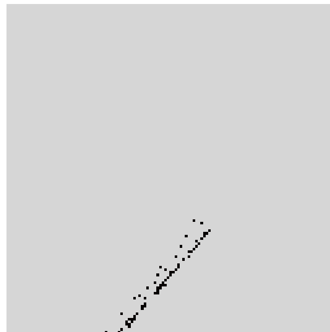

Ilustración 5.27. Heatmap de ejemplo.

Las imágenes generadas se guardarán en la ruta *Assets/Stack*. El nombre de las imágenes será el número de iteración correspondiente con extensión JPG. **Nota:** cada vez que se ejecute la simulación guardará las imágenes en esa ruta por lo tanto sobreescribirá las que haya. Si no se desea perder heatmaps anteriores copiarlos antes de ejecutar una nueva simulación.

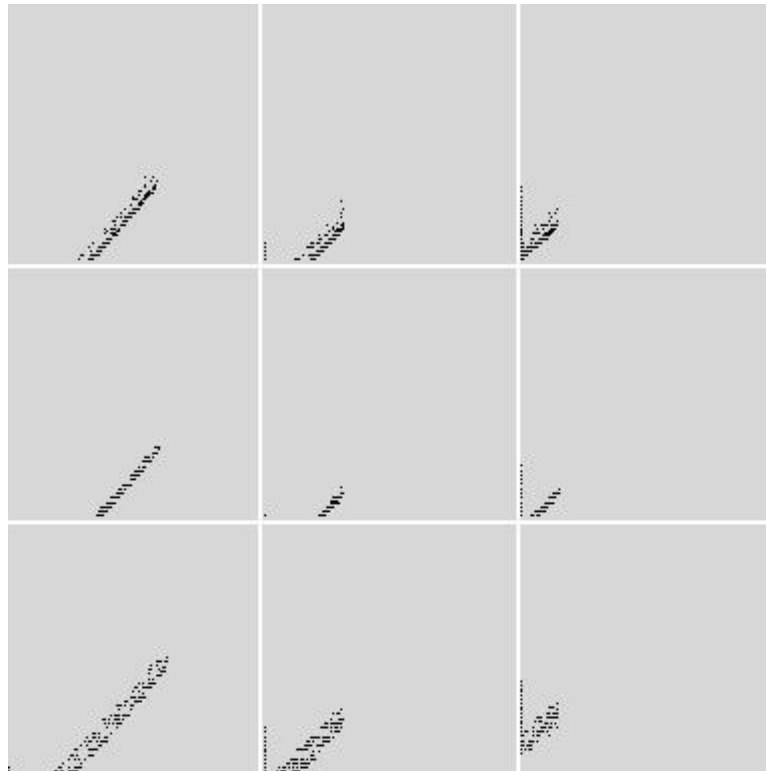

Ilustración 5.28. Nueve Heatmaps de los 60 generados.

6. RESULTADOS

Primera prueba. Prototipo 1.

Al trabajar la luz de forma discreta con sistemas de partículas el cómputo se ve muy limitado dependiendo de la capacidad de cómputo del PC. Al estar tan limitado hay que cambiar algunos conceptos para optimizar y poder capturar la información. La luz difusa reflejada, una vez reflejada por la pared difusora, se propaga de forma esférica por la escena en la vida real pero si esto lo traspasamos a Unity deja muchos huecos vacíos entre partículas y lo que es peor un gran porcentaje de esas partículas serán inútiles ya que nunca colisionarán con el objeto oculto y no aportarán información. Una medida de optimización es aumentar el tamaño de las

partículas que colisionan con el detector teniendo así más información, ya que tendrá más posibilidad de que choque con el detector, pero menos precisa.

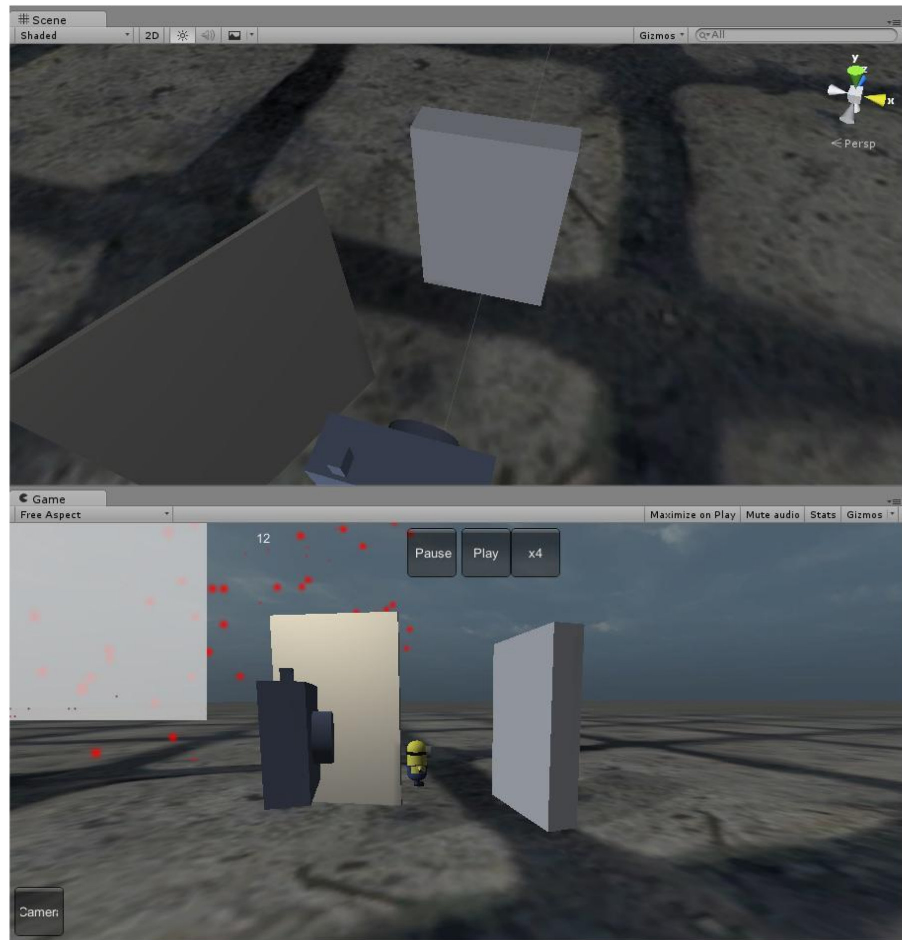

Ilustración 6.1. Captura de la primera prueba del simulador.

He montado la escena con las dimensiones reales del experimento ver tabla 1.1. Para este prototipo no he seguido los datos reales sobre tiempos e intervalos ya que son cantidades muy pequeñas. Esto no es realista y una opción es desarrollar una forma más realista de hacer esto. Los resultados de las colisiones con una resolución de 128 pixeles tienen muy poca información. La imagen de la ilustración 6.2 es el resultado con más colisiones detectadas.

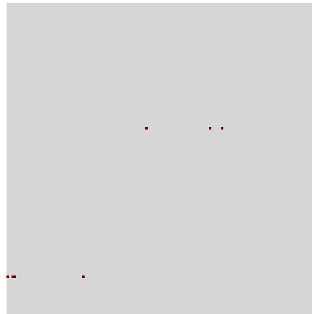

Ilustración 6.2. Uno de los 60 mapas de calor.

Dado el problema anterior he optado por generar unas tablas con información del número de partículas que se pierden y las que colisionan con el objeto oculto y la pared difusa. Para esta tarea he creado un script llamado “CounterScript” que lleva un contador de partículas que chocan en cada objeto y va generando un fichero *html* con una tabla con estos datos para cada iteración del algoritmo. Ver fichero table.html en la carpeta adjunta “tablas”.

Prototipo 2.

Para el segundo prototipo el tiempo ya está proporcional a la realidad y llegan muchas más partículas. Como se puede observar en la tabla conforme el rayo golpea la pared en un punto más alejado del objeto oculto menos partículas capta la cámara. Esto puede fallar por dos factores: no es buena idea apuntar a zonas tan alejadas del muro, o el tiempo de retardo con el que la cámara se acciona es demasiado temprano.

Otro factor a tener en cuenta en este modelo es el epsilon, ya que si tiene un valor alto puede crear nuevas difusiones y esto provocará muchas series.

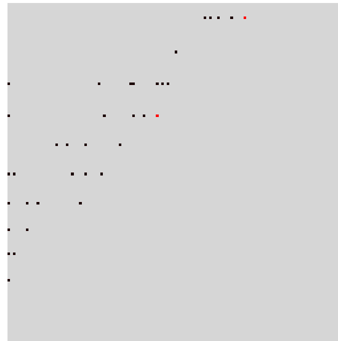

Ilustración 6.3. Mapa de calor con un epsilon mayor de lo habitual.

Este prototipo fue desechado porque la propagación de la luz no es realista.

Modelo de Pattanaik. Prototipo 3.

Después de ver estancado el modelo anterior ya que los resultados no eran buenos, decidí pasarme al modelo de Pattanaik.

Siguiendo el modelo de Pattanaik hacemos que las **partículas** sea un **número constante**, es decir, no se creen nuevas partículas en la colisión con el objeto oculto. Para que este modelo dé suficiente información hay que aumentar mucho el número de partículas generadas, esto a su vez crea el problema de bajo framerate cuando se computan el alto número de partículas.

Con 500000 partículas va a 2 o 3 fps y captura 2 partículas en 3 segundos de exposición de la cámara. Con 250000 va a 10 fps y captura 1 o 2 partículas en 3 segundos. Con un millón de partículas va a 1 fps o menos y captura 3 partículas. Con 5 millones captura 11 partículas y se cuelga el PC.

El tiempo de exposición de la cámara no es el problema ya que siguen capturando pocas partículas aunque triplique el tiempo. El problema viene dado de que la mayoría de partículas se pierden en el espacio y no llegan a rebotar con el objeto y las que lo hacen pocas llegan a la pared y de esas muy pocas muy muy pocas son capturadas por la cámara.

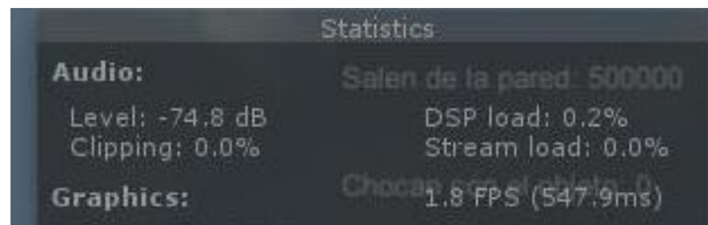

Ilustración 6.4. Caída de framerate debido al número de partículas.

Siguiendo el modelo actual una buena forma de mejorarlo podría ser reducir el ángulo de la semiesfera que genera la reflexión difusa dejando un cono de difusión. Esto fue desechado al principio de las pruebas porque no era realista pero para hacerlo más realista se podría calcular la energía que lleva ese rango de luz y así mejorar el rendimiento o la información capturada o ambas.

Prototipo 4.

Para optimizar esto lo que he hecho ha sido **reducir el ángulo** con el que se propagan las partículas y hacer que estas se dirijan hacia el objeto oculto.

Sigue el modelo de Pattanaik al igual que el anterior pero reduce el ángulo de la semiesfera de luz (partículas) que se genera enfocando directamente al objeto oculto y de esta forma logrando que en una proporción mucho mayor colisione con el objeto y consiga más información.

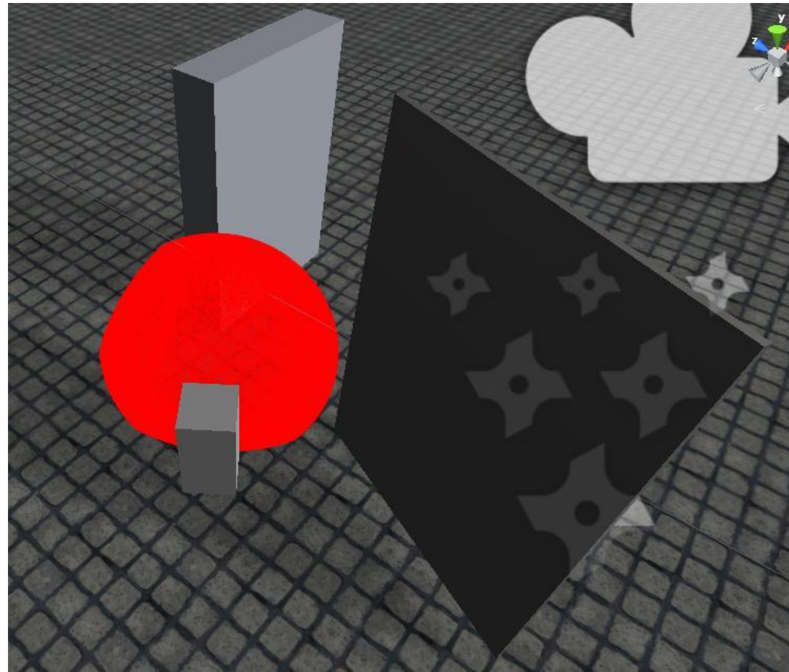

Ilustración 6.5. Propagación de las partículas enfocando al objeto oculto.

Los resultados de este prototipo ya son bastantes mejores y es capaz de capturar 158 partículas lanzando 200000 partículas. Este prototipo proporciona bastante información comparados con los demás prototipos. La única contra es que el sistema tiene que conocer dónde se encuentra el objeto oculto y eso no es muy real, incluso generaría problemas si en el futuro se desea poner dos objetos ocultos.

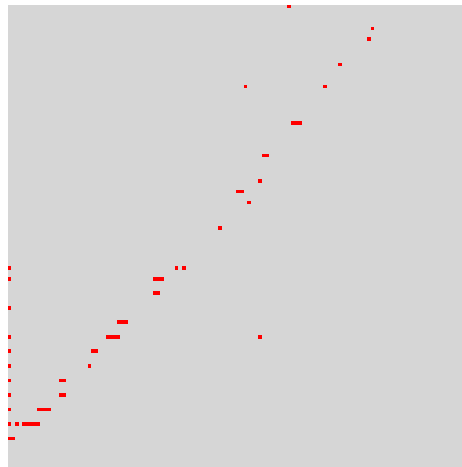

Ilustración 6.6. Mapa de calor generado en el prototipo 4.

Prototipo 5.

Esta es la última y definitiva versión del simulador. Obtiene unos resultados buenos como la anterior pero se **arregla el problema** de dependencia que sufría el simulador en la versión anterior al tener que conocer el objeto oculto. Es muy probable que la cámara recoja información redundante ya que los conos se solapan en el espacio, aunque esta información no es errónea así que no es un problema.

Como contra decir que al tener que dividir el bucle y encima tener que escalar los tiempos hace que el sistema sea muy muy lento si se desea recoger mucha información o si el pc no tiene demasiado potencia de cómputo.

En la ilustración 6.7 se puede observar una muestra de nueve imágenes generadas con este prototipo con bastante información. Se puede observar que en las imágenes que corresponden a cuando el láser apunta a una zona de la pared difusa más alejada horizontalmente la **información** que recoge es mucho menor llegando incluso a ser casi **nula**. Pensé en aumentar el tiempo de retardo en abrir la cámara para que las partículas llegaran cuando la cámara estuviera abierta, pero ocurre que entonces las posiciones más cercanas en el eje X no se detectan. Es decir el mismo problema pero al revés. Otra opción sería disminuir al rango donde el láser apunta en el eje X pero esto perdería diversidad angular. Así que finalmente se ha quedado tal cual ya que de o no de información es lo real.

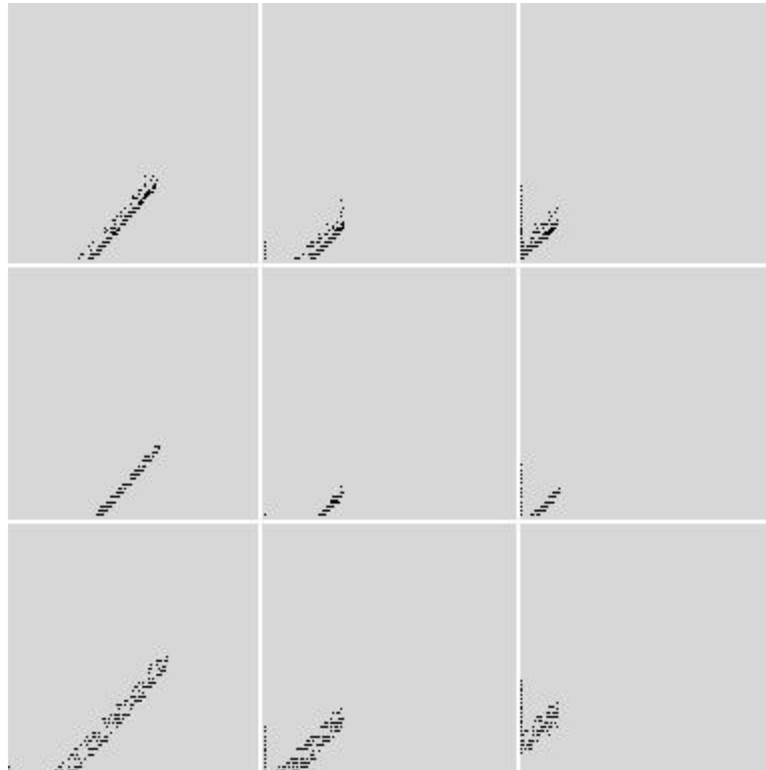

Ilustración 6.7. Nueve Heatmaps generados con el prototipo 5.

Ilustración 6.8. Heatmap generado cuando el láser apunta más alejado en el eje X.

7. CONCLUSIONES

La femtofotografía es una técnica muy moderna, con poco tiempo de maduración y aún en desarrollo. Esta técnica tiene mucho potencial para el futuro pero requiere de trabajo y desarrollo en este campo. Para ello con el simulador

creado en este Trabajo de Fin de Grado quiero facilitar en cierta medida la investigación y la experimentación de esta técnica en una escena concreta con un PC, sin necesidad de los costosos materiales que son necesarios para este experimento como la femto-cámara, el láser y otros elementos.

Realmente este simulador es una herramienta con mucha proyección de ampliación en el futuro.

En resumen, el **principal problema** que ha tenido el simulador ha sido optimizarlo para que diera suficiente información sin que se colgara. Si aumentaba mucho el número de partículas podían pasar dos cosas: el PC era incapaz de computarlas o el delta time era tan grande que ocurría el problema del bajo framerate y las colisiones tenían una muy mala precisión.

Las soluciones para que el PC no tuviera problemas eran dos: **bajar la escala de tiempos** para hacer una simulación más lenta y precisa, **decrementar el número de partículas** y/o hacer que las partículas enfoquen al objeto oculto en un cono de difusión. Bajar la escala es una solución que sirve hasta cierto número de partículas ya que cuando hay muchas los tiempos se hacen eternamente grandes y se convierte más en un problema que en una solución. Decrementar el número de partículas sin duda aumenta el rendimiento del procesamiento pero lógicamente se puede imaginar que la información que proporciona es bajísima. La solución de reducir la semiesfera a un cono de difusión de las partículas y hacer que se dirijan al objeto oculto daba buenos resultados. El simulador era relativamente rápido y daba relativamente suficiente información en las colisiones pero el problema era que el simulador tenía que saber donde se encontraba el objeto para dirigir el sistema de partículas a él. Esto se traduce a que no era genérico a cualquier escena, ni escalable, rígido a modificaciones y obviando el resto de escena.

La **solución**, como ya podréis haber leído, ha sido dividir el espacio en el que se propaga la luz difusamente. De esta forma se abarca todo el espacio y se procesa cada porción con menos carga de partículas y por lo tanto más eficientemente.

8. TRABAJOS FUTUROS

Algunos problemas que tiene esta técnica son las superficies con **poca componente difusa**. La luz no se propaga de forma esférica siendo así incapaz de llegar a todos los puntos de la escena y por lo tanto ninguna o poca información puede ser captada por la femto-cámara.

Se presentan desafíos como el de la **atenuación de la luz**. Para distancias mayores entre la cámara el difusor y el objeto oculto hay una mayor atenuación de la luz. Es necesario que se compense con una fórmula matemática para que el algoritmo de reconstrucción funcione correctamente.

Respecto al simulador, queda pendiente convertir los heatmaps generados que tienen información discretizada, solo son puntos ubicados en un eje X y en el tiempo. Hay que aproximar estos mapas de calor discretos a unos más realistas que sirvan para el algoritmo de reconstrucción. También queda por implementar el **algoritmo back-projection** para la generación del mapa de calor del objeto al cual se le aplica la segunda derivada. Y para acabar también será necesario implementar el proceso de **reconstrucción** dada la imagen anterior para conseguir el modelo 3d según esta técnica.

También sería interesante diseñar una escena con **múltiples femto-cámaras** y/o **múltiples fuentes de luz**, aunque esto ya es más avanzado. Para esto lo ideal es dotar al simulador con una interfaz más intuitiva y amigable que permita modificar más libremente la escena para complicarla y cargar distintos modelos 3D. De esta forma simular escenas reales en las que se puede aplicar la femto-cámara.

Otra ampliación que se puede presentar para el simulador es tener en cuenta las propiedades de la luz y de los materiales. Qué longitudes de onda de la luz son absorbidas y reflejadas por un material concreto. Las longitudes de onda equivaldrían a los componentes RGB (Red, Green, Blue) del color de la luz.

La **auto-escalada** respecto a la potencia del PC queda también pendiente de implementar ya que al final el simulador solo tiene en cuenta el intervalo de digitalización y el tiempo de apertura de cámera.

Aplicaciones

La femto-cámara puede tener muchas aplicaciones en diversos campos, como seguridad vial, aplicaciones médicas, herramienta de reconocimiento de una escena en situaciones de riesgo y muchas más que se nos ocurran.

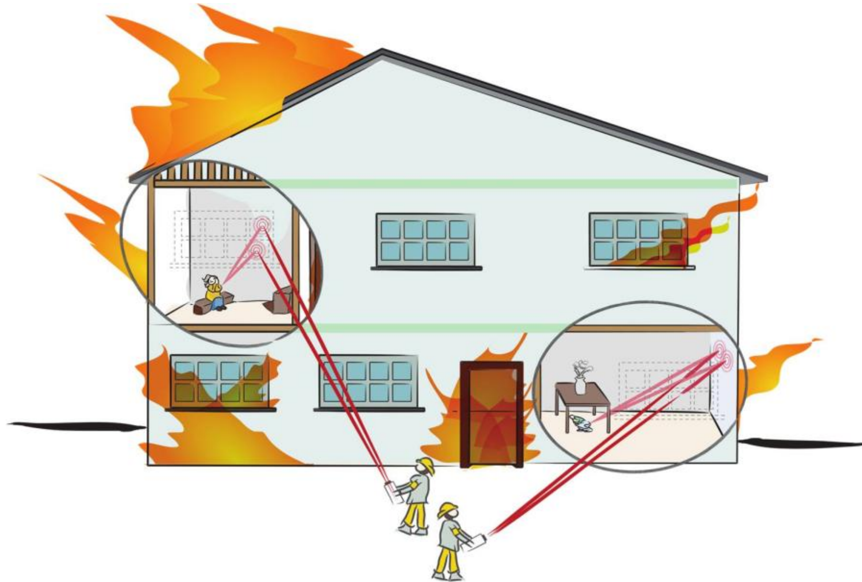

Ilustración 8.1. Aplicación de la femto-cámara en una situación de riesgo.

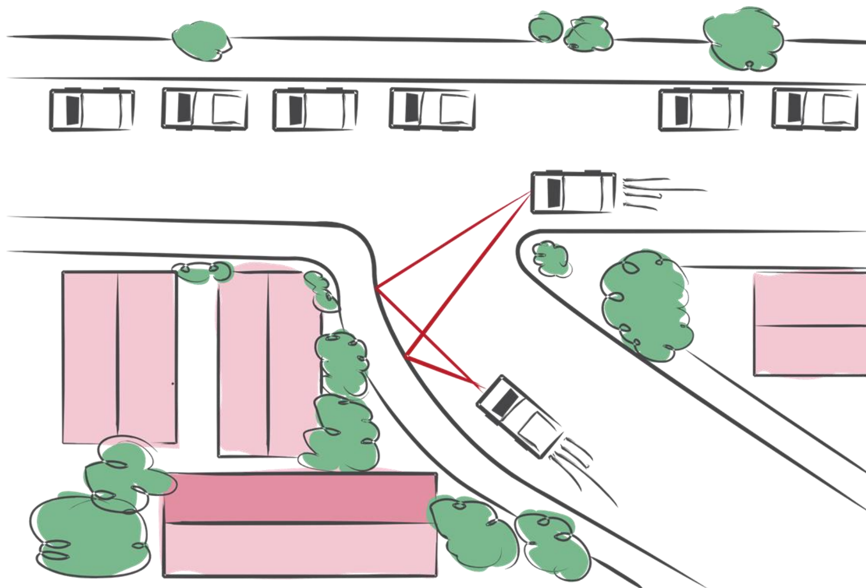

Ilustración 8.2. Aplicación en seguridad vial. Detección de vehículos.

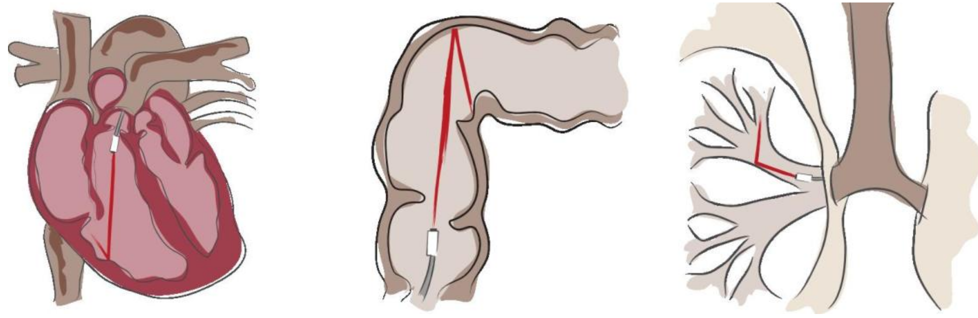

Ilustración 8.3. Aplicaciones médicas de la femto-cámara.

Bibliografía

[Kirmani2011] Kirmani, A. (2011). *Looking Around a Corner using Ultrafast Transient Imaging.*

[Pandharkar2011] Pandharkar, R. (2011). *Estimating Motion and Size of Moving Non-Line-of-Sight Objects in Cluttered Environments.*

[Velten2012] Velten, A. (2012). *Recovering three-dimensional shape around a corner using ultrafast time-of-flight imaging.*

Martínez, L. (2013). Apuntes de la asignatura *Gestión y Control de Procesos Informáticos.* Universidad de Jaén.

Ruiz de Miras, J. (2013). Iluminación y Texturas. Informática Gráfica y Visualización. Universidad de Jaén.

[WEB Femto-Photography] visitado: 21/06/2015. http://web.media.mit.edu/~raskar/trillionfps/

[WEB Looking Around Corners using Femto-Photography] visitado: 21/06/2015. http://web.media.mit.edu/~raskar/cornar/

[WEB Descargar Unity] visitado: 21/06/2015. *http://unity3d.com/es/get-unity*

[WEB Herramienta online para diagrama de Gantt Wrike] visitado: 21/06/2015. *https://www.wrike.com/es/*

[WEB Descarga Visual Paradigm 12.1 Community Edition] visitado: 21/06/2015. *http://www.visual-paradigm.com/download/community.jsp*

[WEB Licencia Unity] visitado: 21/06/2015. *http://unity3d.com/es/company/legal/eula*

[WEB COCOMO] visitado: 21/06/2015. *http://es.wikipedia.org/wiki/COCOMO*

[WEB PF de JavaScript] visitado: 21/06/2015. *http://dspace.unach.edu.ec/bitstream/123456789/60/1/FI-ESC-40A005.pdf*

[WEB Propiedades de la luz] visitado: 21/06/2015. *http://www.aulafacil.com/cursos/l7576/aficiones/fotografia/fotografia-iii-la-luz/propiedades-de-la-luz-absorcion-reflexion-y-transmision*

[WEB Calculadora de Vatios] visitado: 21/06/2015. *http://www.extreme.outervision.com/psucalculatorlite.jsp*

[WEB Ficha del monitor] visitado: 21/06/2015. *http://www.pccomponentes.com/lg_22mp55hq_p_22__led_ips.html*

[WEB Cálculo del consumo eléctrico del PC] visitado: 20/04/2015. *http://elblogdejfinfo.net/como-saber-el-consumo-electrico-de-un-ordenador/*

[WEB Colegio Profesional de Ingenieros en Informática de Andalucía] visitado: 21/06/2015. *http://cpiiand.es/wordpress/*

[Email de Pedro Espina. Decano del CPIIA] leído: 21/06/2015.

[WEB Precio del KWh] visitado: 21/06/2015. http://tarifasgasluz.com/faq/endesa/tarifas/luz

[WEB Colegio Profesional de Ingenieros Técnicos en Informática de Andalucía] visitado: 21/06/2015. *http://www.cpitia.org/*

[WEB Respuestas Plan Bolonia] visitado: 21/06/2015. *http://www.mecd.gob.es/boloniaensecundaria/img/damos_respuestas.pdf*

[WEBLOG JavaScript o C#] visitado: 21/06/2015. *http://thebloodyshadowslan.blogspot.com.es/2013/08/programacion-en-unity-unity-js-vs-c.html*

[WEB GuGames DEV. Software easyPERT] visitado: 21/06/2015. *http://www.gugames-dev.com/*

[WEB Documentación corrutinas Unity] visitado: 21/06/2015. http://docs.unity3d.com/es/current/Manual/Coroutines.html