Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

GameBrush

Plugin de generación de formas abstractas y de alteración visual del entorno para Unity3D

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Guillermo Ferrando Monzón

Tutor (UPV): Ramón Pascual Mollá Vayá

Cotutora (Empresa): Marina González Oller

2015-2016

Resumen

Este proyecto final cubre el desarrollo de un plugin para Unity implementado en C# y orientado al desarrollo de videojuegos para Windows. Su objetivo principal es otorgar a cualquier entidad a la que esté vinculado la capacidad de generar fluidos modelados con blobs de distintos aspectos visuales configurables y de estampar tales aspectos en el entorno virtual. Estos aspectos (por ejemplo: pintura plástica, mercurio, burbujas, etc) estarán definidos por shaders escritos en el lenguaje Cg/HLSL que utiliza Unity, y para la creación de los blobs se utilizarán los geometry shaders, disponibles en las implementaciones de DirectX10 y OpenGL3.2 o superiores. Se creará además un proyecto de Unity para demostrar todas las funcionalidades del plugin y con la intención de analizar su potencial en la creación de videojuegos de exploración artística. Con todo esto se busca obtener una visión general del desarrollo de videojuegos y de la programación de shaders.

Palabras clave: plugin, Unity, unity, blob, geometry, shader, metaball, cg, hlsl, videojuego.

Abstract

This final project tracks the development of a Unity plugin implemented in C# focused on Windows videogame development. Its main purpose is to grant the ability of generating fluids modeled with blobs to whichever entity the plugin is attached to. Each blob generated can have a different configurable appearance and each one of these aspects (such as plastic paint, mercury and bubbles, as example) can be printed all around the virtual environment. The aspects will be defined by shaders written in Cg/HLSL supported by Unity and the blobs will be created using the geometry shaders available on DirectX10 and OpenGL3.2 or greater. A Unity project will be created as well in order to show all the plugin features and to analyze its potential to create artistic exploration videogames. With all this, the personal objective of this project is getting a general vision of videogame development and shader programming.

Keywords: plugin, Unity, unity, blob, geometry, shader, metaball, cg, hlsl, video, game, videogame.

Tabla de contenidos

1. Introducción

El objetivo principal de este proyecto nace a partir del interés del equipo de desarrollo de videojuegos Deconstructeam[1] en crear un prototipo de un videojuego de exploración artística y comprobar su potencial como proyecto más grande, pero no se reduce únicamente a esto. Independientemente del éxito que tenga esta herramienta a nivel interno, se pretende lanzar esta herramienta como plugin en la tienda virtual del entorno de desarrollo de videojuegos Unity3D (Unity en adelante) para que otros desarrolladores puedan utilizar el sistema.

La gran ventaja de la tienda virtual de Unity es que permite que los equipos de desarrollo pequeños puedan centrarse en el diseño del juego en sí y no en programar las herramientas que necesitan. También está el caso de desarrolladores novatos que desean hacer sus ideas realidad y para ello hacen uso de los recursos que encuentran en la tienda virtual para reducir al máximo las tareas de programación.

1.1 Motivación

El interés personal por estudiar el pipeline gráfico y aplicar efectos visuales al propio gameplay del juego y el interés de Deconstructeam de explorar nuevas formas de interacción poco explotadas en los videojuegos han dado lugar a este proyecto.

1.2 Objetivos

El objetivo principal de este proyecto es el diseño e implementación de un prototipo de un plugin que incorpore la funcionalidad de un sistema de fluidos tridimensional al entorno Unity3D. El prototipo contará con un subconjunto de la funcionalidad del producto final, pero deberá ofrecer la visión global del producto acabado.

El desarrollo del prototipo se centra fundamentalmente en obtener:

1. Una herramienta intuitiva, fácilmente configurable en su comportamiento por el usuario.
2. Una herramienta en la que resulte sencillo incorporar nuevos comportamientos no contemplados en el diseño original del plugin.

[1] Ver ANEXO

3. Un aspecto de fluido parametrizable, dando además la posibilidad al usuario de programar e integrar nuevos shaders que definan otro aspecto diferente.

4. La capacidad de pintar las superficies que han resultado impactadas por un fluido.

5. Un sistema eficiente que pueda ejecutarse en máquinas de gama baja a una tasa razonable de fps.

Estos puntos definen la base del plugin. En este prototipo no se va a hacer hincapié en obtener muchos tipos de aspectos programados diferentes con distintos modelos de iluminación, sino en la creación de un único aspecto configurable que permita el cambio de la apariencia del fluido y en la posibilidad de agregar nuevos aspectos programados. Tampoco se busca implementar un conjunto de comportamientos que se puedan utilizar en el mayor número de situaciones posibles sino en lograr que la creación e inclusión en el sistema de estos comportamientos resulte sencillo.

A nivel personal, durante el desarrollo se busca aprender los pilares fundamentales del desarrollo de herramientas, mecánicas y efectos gráficos para videojuegos, así como las características y posibilidades de cada una de las herramientas escogidas. También se pretende mejorar los conocimientos de diseño de software, intentando lograr una arquitectura del plugin flexible que permita su extender su funcionalidad fácilmente. Otro objetivo importante que se busca es mejorar la capacidad de gestión de tiempos, siendo esta una cualidad muy valorada en la industria del videojuego.

Respecto al producto desarrollado, se pretende explorar con él nuevas formas de interacción del jugador, nuevas mecánicas de juego, no explotadas hasta la fecha.

1.3 Estudio DAFO

Para cumplir con los objetivos mencionados en el apartado anterior se ha realizado un pequeño estudio de nuestra propuesta para enfocar el ritmo de trabajo de forma adecuada. Para ello se ha elaborado un pequeño análisis DAFO que se muestra a continuación:

	INTERNO	EXTERNO
Negativo	**Debilidades**	**Amenazas**
	- Falta de experiencia en el campo.	- Existen plugins similares en el mercado.
Positivo	**Fortalezas**	**Oportunidades**
	- Gran motivación personal. - El tiempo de desarrollo es flexible, no existen fechas de entrega ajustadas.	- No hay videojuegos que exploten los fluidos como elemento diferenciador. - El mercado de plugins para entornos de desarrollo de videojuegos está en auge.

Hecho esto se han extraído los riesgos más importantes de este análisis, que resultan ser de planificación y de diseño, por la inexperiencia.

1.4 Metodología

Para paliar estos riesgos se ha decidido dedicar aproximadamente un mes de aprendizaje y seguir un enfoque iterativo en el desarrollo, típico de las metodologías ágiles. Esto es, realizar varias veces el proceso de análisis, diseño, implementación y pruebas del proyecto, mejorando el prototipo en cada iteración la anterior. De esta forma, cualquier falta que se cometa en una iteración por la ausencia de experiencia o cualquier otro motivo, puede corregirse en la siguiente. Hay que mencionar que se ha decidido no utilizar pruebas formales de software debido a su complejidad de uso en el entorno Unity y en la programación gráfica. Así pues, las características principales de la metodología seguida son:

- Desarrollo iterativo e incremental.
- Mantener el código simple.
- Uso de la refactorización del código como herramienta para avanzar entre iteraciones.
- Pruebas informales constantes sobre el código creado.

1.5 Estructura de la obra

En lo que respecta a continuación, en el capítulo 2 se va a proceder a realizar un estudio del estado del arte con la intención de analizar las amenazas externas y comprobar las herramientas disponibles que se pueden utilizar para desarrollar el plugin. En el capítulo 3, análisis del problema, se va a realizar un estudio de las distintas etapas de análisis por las que ha pasado el sistema debido al enfoque iterativo seguido. En el capítulo 4, diseño de la solución, se comenta y justifica el diseño de la arquitectura del plugin que se ha conseguido en la última iteración del prototipo. Más adelante, en el capítulo 5, implementación, se estudian las bases matemáticas y tecnológicas que han hecho posible el desarrollo del fluido en sí. En el capítulo 6 se realiza un repaso a los resultados obtenidos y en el capítulo 7 se estudia el trabajo que queda por realizar.

Finalmente, en los capítulos 8, 9 y 10 se encuentran los agradecimientos, la bibliografía y el ANEXO, que a su vez contiene un glosario con los términos más específicos de la materia en caso de que el lector no esté familiarizado con ellos.

1.6 Convenciones

Es necesario aclarar aquí que Unity utiliza el lenguaje propio ShaderLab que puede embeber snippets de Cg (“**C** for **g**raphics”) para la creación de shaders. Cg actualmente está descontinuado[2] , y se trata de un lenguaje con exactamente la misma sintaxis que HLSL (soportado por Direct3D) cuya propuesta era convertirse en un lenguaje único capaz de compilar el código escrito en las distintas implementaciones gráficas existentes. Debido a esto y a que Unity utiliza los conceptos de Direct3D y no de OpenGL, se van a obviar todas las particularidades de OpenGL excepto en casos puntuales y se va a explicar todo lo relacionado con el aspecto gráfico enfocado a Direct3D.

Puesto que se va a utilizar la nomenclatura de Direct3D, resulta vital aclarar qué es un fragmento, y qué es un pixel. En Direct3D, la etapa del sombreado en el pipeline gráfico se conoce como **pixel shader**, a diferencia de en OpenGL, que se nombra **fragment shader** cuando, sin embargo, ambas etapas tienen exactamente la misma responsabilidad: tomar un fragmento como entrada y devolver otro fragmento con, al menos, un color y una profundidad del pixel en la salida. Devuelve otro fragmento y no

[2] https://developer.nvidia.com/cg-toolkit

directamente un píxel coloreado porque aún queda realizar otros procesos para garantizar que el color que se va a mostrar por pantalla es correcto [1].

Así pues, cuando se hable de fragmento en el presente texto, se habla de una estructura que contiene toda la información necesaria para calcular el color de un píxel.

Por otro lado, para generar el fluido se hace uso de la técnica de las *metaballs* o metabolas. En la literatura sobre esta materia no existe ninguna convención en la nomenclatura de los términos, por lo que resulta necesario definir ahora los conceptos que se utilizarán en todo este apartado.:

- **Blob**: Es el átomo del fluido. Varios blobs en conjunto generan el fluido.
- **Metaball (metabola)**: Conjunto de varios blobs, el fluido en sí.
- **Fluido**: Efecto generado por la metabola.

En muchos casos el concepto de metabola y fluido pueden ser intercambiables, pero se utiliza el concepto de metabola a nivel de implementación como estructura de varios blobs y el concepto de fluido cuando no es necesario conocer cómo está modelado el sistema.

En el caso de la bibliografía, se ha empleado la convención propuesta por la organización IEEE.

2. Estado del arte

2.1 Entornos y motores de videojuegos

En la actualidad existen muchas alternativas para escoger. Se pueden realizar dos grandes grupos según el tipo de facilidades que otorgan.

***Frameworks* de videojuegos**: Son un conjunto de librerías que dan soporte para programar sistemas complejos como el sistema gráfico o el físico. Por ejemplo:

Ilustración 1: Torchlight, Ogre3D

- **Ogre3D**: Se trata de un motor gráfico escrito en C++. Tiene una amplia comunidad y una buena documentación.

Ilustración 2:
The Hinterlands , LibGDX

- **LibGDX**: Es un framework Java orientado al desarrollo de videojuegos 2D. De código abierto, gratuito y multiplataforma, cuenta con muchos seguidores en su comunidad y con una documentación muy completa.

Ilustración 3:
port de Bastion para Xbox360, Monogame

- **MonoGame**: Framework C# de código abierto y basado en el actualmente obsoleto XNA. Se ha utilizado sobre todo para crear ports de Xbox 360 de forma rápida.

Entornos o motores de desarrollo: Son mucho más que un conjunto de librerías. Se trata de programas completos con toda la funcionalidad necesaria para crear un videojuego y con un flujo de trabajo integrado para artistas, diseñadores y programadores. Normalmente en este caso, los programadores que intervienen en los sistemas del juego lo hacen a través de un lenguaje de scripting que provee el propio entorno. A continuación se muestra una lista de los más relevantes actualmente:

- **Unreal Engine 4 (UE4)**, de *Epic Games*. De código abierto y gratuito, si bien hay que pagar un royalty en el caso de comercializar un producto con este motor. Orientado a producciones de alta calidad gráfica y equipos de desarrollo grandes.

Su scripting es en C++ y en un sistema de programación visual propio conocido como Blueprint. Exporta a una gran cantidad de plataformas, tanto de escritorio, como móviles, además de las consolas PS4 y Xbox One.

Ilustración 4: UE4, Unreal Tournament 4

- **Unreal Development Kit (UDK)**, de *Epic Games*. Es la versión anterior del UE4. Utiliza un lenguaje de scripting propio conocido como *UnrealScript* y exporta a sistemas de escritorio y consolas de la pasada generación como PS3 y Xbox360.

Ilustración 5: UDK, Unreal Tournament 3

- **CryEngine**, de *Crytek*. Como el UE4 es de código abierto y gratuito, pero a diferencia de este no es necesario pagar ningún royalty. Orientado también a producciones de grandes equipos y alta calidad visual. Su sistema de scripting está basado en Lua y exporta a plataformas de escritorio y consolas de última generación. Su principal inconveniente es que presenta una curva de aprendizaje elevada.

Ilustración 6: CryEngine, Crysis

- **Lumberyard**, de *Amazon*. Es un nuevo motor, actualmente en fase temprana de desarrollo que aspira a competir con UE4 y CryEngine. También gratuito y de código abierto. Su scripting es en Lua y exporta a plataformas de escritorio, móviles y consolas de última generación. Debido a su estado de desarrollo, su comunidad es pequeña, su documentación escasa y Amazon ofrece ayudas económicas para quién desee producir un juego con su motor.

Ilustración 7: Lumberyard, demo técnica

Ilustración 8: Unity3D, Firewatch

- **Unity3D**, de *Unity Technologies.* Orientado a equipos de desarrollo pequeños. Principalmente estaba enfocado a videojuegos 3D, pero más adelante se le incorporó un flujo de trabajo para juegos 2D. Tiene tres lenguajes de scripting disponibles: JavaScript, C# y Boo. Exporta a móviles y consolas, además de a

Linux, Windows y Mac, si bien el editor solo está disponible para estas dos últimas. Existe una versión gratuita disponible.

Ilustración 9: Godot, Dog mendoza and pizzaboy

- **Godot**, de *Okam Studio.* De código abierto y gratuito, permite hacer juegos tanto 2D como 3D. Su lenguaje de scripting es propio y está basado en Python. El editor funciona en los tres grandes sistemas operativos, Windows, Mac y Linux, y permite exportar a las plataformas móviles y de escritorio más comunes. Su principal defecto es que se trata de un entorno joven y, por lo tanto, su documentación es limitada.

Ilustración 10: GameMaker, Gods Will Be Watching

- **Game Maker Studio**, de *YoYo Games*. Permite crear la lógica del juego sin necesidad de conocimientos de programación, pero también cuenta con su propio lenguaje de scripting. Está principalmente orientado a desarrollos de juegos 2D, y es capaz de exportar a una gran cantidad de plataformas distintas, móviles o de escritorio, si bien el editor funciona únicamente en Windows. Tiene una versión de pago y otra gratuita, aunque con limitaciones.

El uso de los *frameworks* ha quedado descartado desde el principio debido a que resulta más complejo realizar el desarrollo con ellos y no existe un mercado de plugins para ellos. Veamos a continuación una tabla comparativa con los aspectos más importantes de cada uno de los entornos de desarrollo:

MOTOR	SCRIPTING	COMUNIDAD	ENTORNO	PRECIO
UE4	C++/Blueprint	Pequeña, Creciente	2D/3D	Gratis, royalty
UDK	Propio	Grande, Decreciente	3D	Gratis, royalty
CryEngine	Lua	Pequeña	3D	Gratis
Lumberyard	Lua	Pequeña	3D	Gratis
Unity3D	C#, JavaScript, Boo	Grande	2D/3D	Versión gratuita poco limitada
Godot	Propio	Pequeña	2D/3D	Gratis
Game Maker Studio	Propio	Grande	2D	Versión gratuita muy limitada

Una alternativa viable habría sido Unreal Engine 4 o UDK. No obstante, UDK es un entorno de la anterior generación que está cayendo en desuso y UE4 es un motor gratuito desde hace relativamente poco tiempo, por lo que actualmente la comunidad interesada en los plugins en la tienda virtual no es tan grande como en el caso de Unity.

GameMaker está centrado en desarrollos 2D, por lo que no nos sirve. CryEngine tiene una comunidad muy pequeña y además es más difícil de usar que UE4, aportando ambos soluciones similares. La ventaja de CryEngine es la ausencia de *royalties,* pero esto no nos afecta debido a que se va a desarrollar un plugin para el motor y no un videojuego. Lumberyard está aún en desarrollo y eso lo convierte en un entorno inestable y difícil de trabajar con él. Finalmente, Godot carece de documentación, por lo que el desarrollo sería más largo y arduo que en Unity, ofreciendo ambos entornos similares.

Con todo esto, se ha decidido optar por Unity, que además resulta ser el entorno de desarrollo que se ha empleado a su vez en las prácticas de empresa. Si bien en la empresa se utilizaba fundamentalmente el aspecto 2D y sigue siendo necesario un aprendizaje del aspecto 3D, se trata de un entorno conocido y la curva de aprendizaje será menor. Otra de las características por las que se ha decidido optar por Unity es el factor de consolidación del motor entre los desarrolladores. Actualmente, Unity tiene una comunidad muy grande y gran parte de esta hace uso de los plugins disponibles en su tienda virtual para reducir el tiempo de desarrollo. Esto supone que el plugin final desarrollado tendrá mejor acogida en este entorno de desarrollo que en el resto. Hay que tener en cuenta que al utilizar Unity, resulta obligatorio el uso del lenguaje propio de shaders, ShaderLab, que hace uso de snippets de Cg/HLSL frente a OpenGL para crear shaders personalizados.

2.2 Videojuegos y sistemas de fluidos

Las aplicaciones prácticas de los fluidos en videojuegos actualmente son muy pocas debido al alto coste computacional que requiere generarlos, y más aún si pensamos que los videojuegos son aplicaciones en tiempo real. Se pueden utilizar precomputados como animaciones, pero en tal caso no pueden evolucionar de forma arbitraria según las acciones del jugador, que es lo que les hace realmente atractivos en un entorno interactivo.

En cualquier caso, como ejemplos de aplicaciones precomputadas tenemos:

- Sistema BlobMesh de 3Ds Max[3].
- Simulación de fluidos[4].

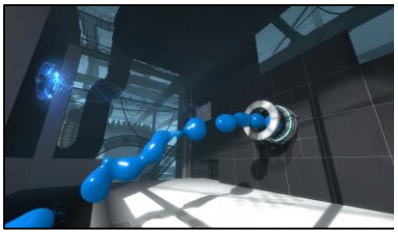


Ilustración 11: Portal 2

Como ejemplo de fluido en tiempo real:

- Fluido en el videojuego Portal 2 de Valve Software. Se utilizan grandes blobs y poca cantidad, por cuestiones de rendimiento.

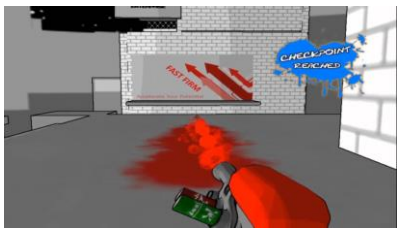


Ilustración 12: Tag

Ilustración 13: Splatoon

No hay muchos más videojuegos que utilicen fluidos de este tipo. Es por este motivo por el que resulta interesante crear un plugin que permita su integración en otros videojuegos: este tipo de fluidos son desconocidos por la mayoría de desarrolladores y pueden tener aplicaciones muy interesantes. Por ejemplo, en Tag, un videojuego creado por un grupo de estudiantes de la escuela Digipen, podrían haber utilizado un sistema de fluidos para hacer sus mecánicas más interesantes. Esto es algo que también se pensó en Valve Software, hasta el punto de contratar al equipo de desarrollo de Tag para ayudarles a crear el sistema de geles existente en Portal 2[5].

Un juego donde podría aplicarse un sistema de fluidos como el propuesto es el Splatoon. Se podrían crear armas o mecanismos que emplearan un sistema de fluidos para pintar el entorno.

También se puede explorar la capacidad de este sistema para crear mecánicas más introspectivas en las que no haya acción. Un videojuego de exploración en el que puedes pintar el entorno, y según cómo lo pintes el mundo evoluciona de una forma u otra, puede ser un ejemplo de videojuego que pueda utilizar esta herramienta para explorar nuevas ideas.

[3] https://www.youtube.com/watch?v=LYa6a4L4MzU
[4] https://www.youtube.com/watch?v=_Z9TAOTf7bs
[5] http://www.shacknews.com/article/62689/valve-hires-digipen-team-seemingly

2.3 Sistemas alternativos ya existentes

En lo que respecta a las herramientas disponibles actualmente para Unity en la tienda virtual, se pueden encontrar las siguientes:

- **Fluidity**: Permite crear fluidos volumétricos de varios tipos, pero especialmente gaseosos. No se pueden crear fluidos con la consistencia de gel que es lo que buscamos.

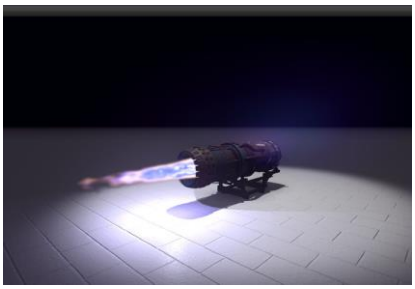


Ilustración 14: Fluidity

- **Cocuy**: Es un sistema que genera fluidos muy vistosos para aplicar a videojuegos 2D.

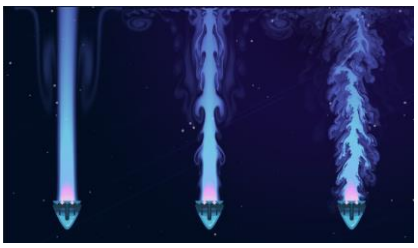


Ilustración 15: Cocuy

- **FluidSim**: Como Cocuy, también es para videojuegos 2D.

- **Fluvio**: Este plugin genera algo parecido a lo que se desea obtener, pero está pensado para generar muchos más blobs simultáneos de peor aspecto visual. Es un sistema mucho más genérico y versátil, pero menos vistoso. Está pensado más para efectos visuales que para una interacción directa por parte del jugador.

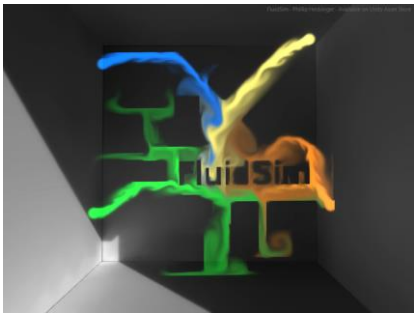


Ilustración 16: Fluidsim

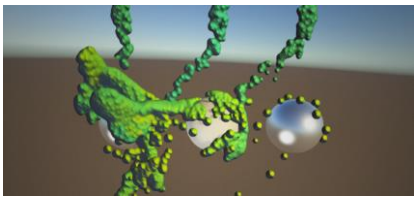


Ilustración 17: Fluvio

- **uFlex**: Este plugin integra la tecnología Nvidia Flex [2] en Unity. Esta tecnología es capaz de generar unos efectos impresionantes, pero por contrapartida se necesita un equipo de gama alta para que funcione con una tasa de fps aceptable.

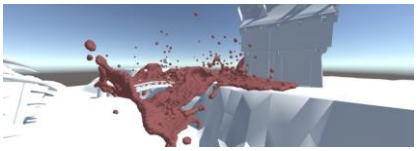


Ilustración 18: uFlex

- **CornFlex**: Este sistema modela el tipo de fluidos que buscamos, pero no está disponible en la tienda virtual de Unity para su uso externo.

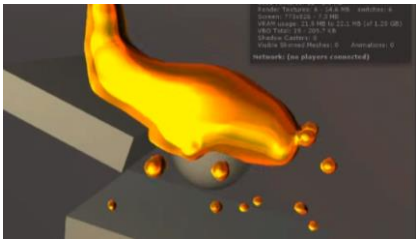


Ilustración 19: CornFlex

Es necesario resaltar que ninguno de estos sistemas tiene la capacidad de pintar de forma permanente las superficies por donde pasa el fluido.

2.4 Sistema propuesto

Resulta curioso que con la tecnología y herramientas actuales los sistemas de fluidos no tengan una mayor influencia en el mercado de videojuegos actuales. Analizando con detalle el estado del arte se puede inducir que el enfoque actual es utilizar sistemas de fluidos para conseguir elementos gráficos. No obstante, nuestro enfoque resulta más novedoso, damos un paso más allá y nuestro objetivo es conseguir un sistema de fluidos que si bien puede utilizarse para crear elementos gráficos decorativos, su principal fuerza reside en el aspecto interactivo que tendrán los fluidos generados.

Nuestra propuesta de sistema es, pues, un plugin para Unity capaz de generar fluidos de distintos aspectos centrado en la creación de relativamente pocos blobs pero de una alta calidad visual. Además, en el prototipo que se va a desarrollar, estos fluidos podrán tener un comportamiento especial al impactar contra ciertas superficies, pintándolas de su mismo color, o borrando la pintura ya creada, según el caso.

El objetivo que se espera alcanzar con el plugin final es tener un sistema lo suficientemente parametrizable como para poder generar el fluido que se desea, tanto en apariencia como en comportamiento, en el mayor número de situaciones posibles. Y, aun así, si no se puede conseguir el sistema buscado, tiene que resultar sencillo una vez entendida la arquitectura del plugin incluir la funcionalidad deseada a través del scripting de Unity.

Por otro lado, el prototipo se va a desarrollar con las máquinas Windows en mente y aunque en las máquinas MAC y Linux también debería de funcionar no se asegura con garantía que lo haga debido a la imposibilidad de testeo por falta de recursos. Para el plugin final estas dos plataformas sí que estarán soportadas.

3. Análisis del problema

En las anteriores secciones se ha realizado un repaso sobre el estado del arte y se ha definido el sistema propuesto en líneas generales. En este punto es necesario realizar un repaso exhaustivo de los requisitos del sistema y extraer de su estudio los subsistemas que se tienen que construir, así como la definición de su intercomunicación y viabilidad de implementación, para obtener el sistema final. Más adelante, en el apartado de diseño, se estudiará en profundidad la estructura interna de cada uno de estos subsistemas.

Empecemos desarrollando la lista de especificaciones del plugin que pueden ayudar a modelar el sistema:

1. El plugin debe poder integrarse en cualquier proyecto de Unity y configurarse fácilmente a través de su editor. Cuando se habla de configuración, se habla de:
    a. Forma en la que se genera el fluido.
    b. Forma en la que el fluido evoluciona con el tiempo en su entorno.
    c. Características del fluido (tamaño, resolución, etc).
2. Se tiene que dar la posibilidad al usuario de generar fluidos que pinten las superficies impactadas con los materiales que él mismo defina.
3. Se puede extender la funcionalidad del plugin de forma sencilla a través de scripts C#.
4. Se tiene que poder escribir nuevos shaders en Cg/HLSL que definan la apariencia del fluido.
5. Debe ser eficiente y funcionar a 30 fps en equipos modernos de gama baja-media, y a 60 fps en equipos de gama media-alta.

A la hora de modelar el sistema se observa que existe una entidad nuclear a partir de la cual encajan las demás piezas: el fluido. Al fluido se le configuran sus características de tamaño y aspecto, su forma de emisión y su manera de evolucionar en el entorno. El hecho de tener un fluido capaz de pintar su entorno se puede modelar como una forma de evolución del fluido.

Debido a la metodología iterativa empleada, el proyecto ha sufrido por varias etapas de análisis, modelándose y mejorando poco a poco con el tiempo en función de los resultados obtenidos en las iteraciones anteriores. Así pues, se ha empezado por lo más simple de todo: conseguir una implementación simple y muy básica de un fluido

estático. El objetivo de realizar directamente una implementación rápida sin tan siquiera plantearse el diseño del plugin es el de tomarle el pulso al sistema y descubrir rápidamente los problemas a los que va a ser necesario enfrentarse, realizando un análisis posterior acorde a estos problemas encontrados.

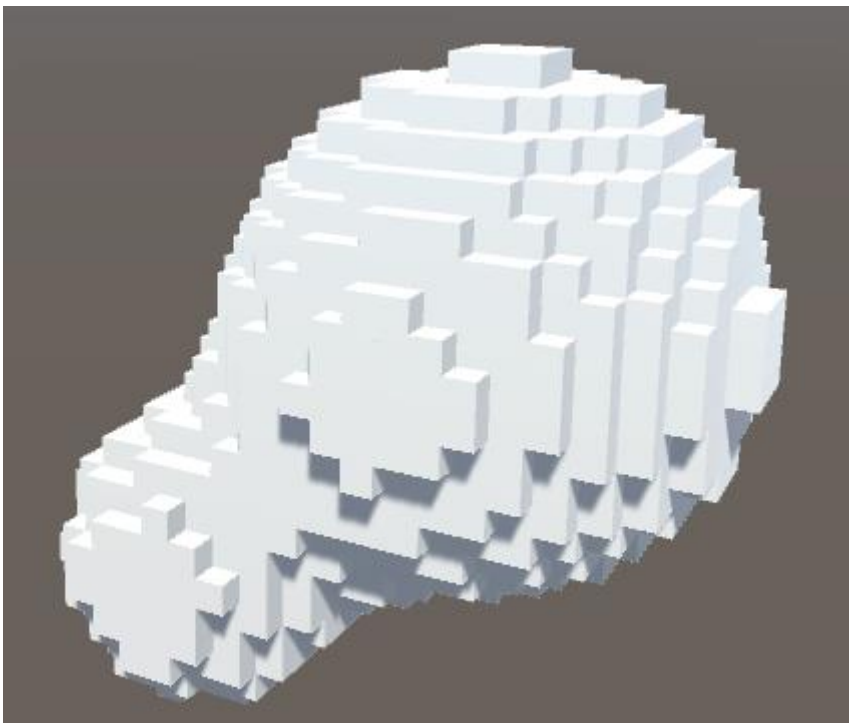


Ilustración 20: Primera aproximación del prototipo

Esta primera aproximación ha servido enormemente a las posteriores iteraciones de esta fase de análisis. Con ella se ha descubierto:

1. La complejidad computacional de la generación del fluido es muy costosa, inviable a realizar en la CPU si se desea conseguir fluidos dinámicos en tiempo real.

2. Existen dos técnicas para generar fluidos del aspecto visual deseado:
    a. Por recubrimiento de partículas [3].
    b. Utilizando el algoritmo de los Marching Cubes o su versión Marching Tetrahedra [4].

    La primera técnica se ha descartado por su complejidad, quedando la segunda a implementar en la GPU en la siguiente aproximación del prototipo.

El siguiente paso ha sido construir una versión dinámica del fluido, generando una malla irregular, relegando la implementación del algoritmo de los Marching Tetrahedra que consigue el suavizado de la malla para más adelante. Para ello ha sido necesario plantearse la gestión de:

1. La emisión del fluido.
2. La evolución temporal del fluido en su entorno.

Esta gestión puede realizarse en dos subsistemas separados: el subsistema de emisión y el de evolución. Debido a que el sistema tiene que extenderse fácilmente, cada comportamiento de emisión y de evolución puede entenderse como una unidad aislada en su subsistema que se inyecta al fluido para darle el comportamiento deseado. Podría haberse modelado esta gestión en un único sistema común, pero en tal caso se pierde la posibilidad combinatoria de tener varios comportamientos de emisión y de evolución en paralelo.

En este punto, nos encontramos con tres subsistemas:

1. **Subsistema de emisión**, encargado de generar cada uno de los blobs del fluido.
2. **Subsistema de visualización**, encargado de generar el aspecto de fluido.
3. **Subsistema de evolución**, encargado de definir el comportamiento de cada blob del fluido en función del tiempo y de su entorno.

El emisor de blobs se encarga de generar las instancias de fluido, arrancando el sistema de visualización y de evolución para cada fluido. Con este modelo tenemos la posibilidad de gestionar la emisión, visualización y evolución del fluido de forma independiente, únicamente teniendo en cuenta que es el sistema emisor el encargado de arrancar los otros dos subsistemas. Esto va a lograr un sistema poco acoplado y abierto a la extensión.

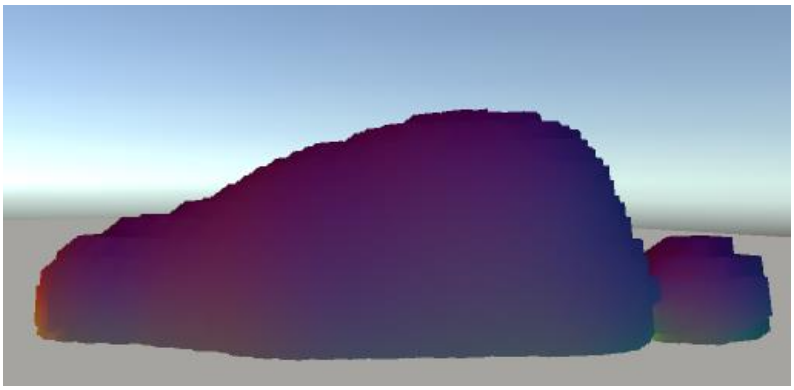


Ilustración 21: Segunda aproximación del prototipo

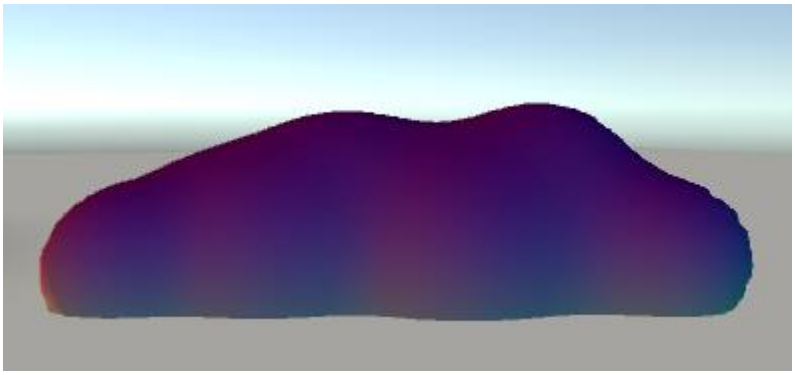


Ilustración 22: Tercera aproximación del prototipo

Con el shader encargado de generar la malla del fluido acabado, resulta necesario analizar cómo se va a modelar la gestión de las distintas apariencias del fluido. Ya existe un subsistema de visualización creado por lo que resulta lógico incluir esta gestión en él.

En este punto tenemos una buena parte de la especificación del plugin cubierta: se puede generar un fluido de forma eficiente en la GPU con sus comportamientos de emisión y evolución, así como su aspecto, gestionados en sistemas desacoplados, por lo que su extensión resulta sencilla. Por otro lado, el comportamiento de pintar las superficies con el impacto del fluido se puede incluir en el sistema de evolución.

Así pues, solo queda el aspecto configurable del plugin: la configuración de los comportamientos, características y aspecto del fluido tiene que ser intuitiva y sencilla. Para lograr esto es necesario exponer ciertas variables al editor de Unity. Una forma de hacerlo consiste en separar esta gestión en otro subsistema, el subsistema de interfaz, creando una arquitectura de dos capas:

1. **Capa de presentación**: Contiene el subsistema de interfaz y gestiona la entrada de usuario desde el editor de Unity.
2. **Capa de lógica**: Utiliza la entrada de usuario para definir el fluido. En esta capa se incluyen los subsistemas de emisión, visualización y evolución del fluido.

Combinar ambas capas en una sola no es solo una mala idea, sino que además con Unity resulta imposible. El aspecto de la exposición de variables de los scripts está integrado en el motor y únicamente se necesita construir una pequeña clase para gestionar la entrada en el caso de desear tener una interfaz con cierta complejidad. Si lo que se desea es una interfaz sencilla, el acceso a estas variables resulta casi automático. Toda la información recopilada en la capa de presentación se distribuirá por la capa lógica a través del subsistema de emisión. Será este el sistema encargado

de la distribución de la información y arranque de los otros dos sistemas, además de la emisión del fluido en sí.

Finalmente hemos acabado con tres grandes subsistemas en la capa lógica con las siguientes responsabilidades:

1. **Sistema de emisión**. Se trata de la definición de las acciones que ocurren cuando un blob se instancia en la escena. Esto incluye la forma en la que se genera (por ejemplo, de forma automática en un punto del espacio determinado o con un click del ratón en una posición señalada por el jugador) y qué sucede al inicio de la instanciación de cada blob (aparece justo en la posición indicada, en un área de alrededor, de forma instantánea o tras un tiempo de espera, etc.). También define la apariencia inicial del fluido, esto es, qué tipo de pixel shader se va a utilizar y con qué parámetros, y se define también el comportamiento evolutivo del fluido. Recibe toda esta información de la capa de presentación y la distribuye al resto de sistemas de la capa lógica.

2. **Sistema de evolución temporal del fluido en su entorno**. Una vez se haya instanciado cada blob es necesario establecer qué sucede en función del tiempo y cómo va a evolucionar el fluido según su entorno. Este sistema es el encargado de definir este comportamiento. El fluido puede quedarse estático, o bien colisionar contra el entorno y desaparecer, o permanecer en él. También puede quedar levitando o interaccionar con el jugador de alguna manera.

3. **Sistema de visualización**. Este subsistema se encarga de hacer el fluido visible con el aspecto definido por el usuario. Ninguno de los dos subsistemas anteriores tendría sentido si el jugador no pudiera ver lo que está ocurriendo, cómo se comporta el fluido ante sus acciones.

La funcionalidad de pintar las superficies que colisiona el fluido será, pues, un tipo concreto de comportamiento evolutivo y se analiza la solución adoptada con detalle más adelante, en el apartado de implementación.

3.1 Planificación

Como ha quedado patente en la sección anterior, para la implementación del plugin en Unity se ha decido seguir un enfoque iterativo, construyendo versiones muy primitivas al inicio para afianzar los conceptos y atajar los problemas de forma incremental.

Para gestionar el tiempo de trabajo, se ha realizado un diagrama de Gantt con una planificación inicial pesimista con la finalidad de modelar la inexperiencia.

Se ha reservado un espacio de un mes para el aprendizaje y se han definido cuatro iteraciones, de una duración de dos semanas cada una aproximadamente. Además, se ha creado una última tarea nombrada “colchón de contingencias” cuya finalidad es amortiguar los retrasos posibles que pueda sufrir el proyecto.

	Tarea	Inicio	Fin	Dur
	GameBrush	22/2/16	30/6/16	94
1	Documentación y aprendizaje	22/2/16	25/3/16	25
1.1	Aprender Unity3D	22/2/16	4/3/16	10
1.2	Aprender Shaders	5/3/16	18/3/16	10
1.3	Documentarse sobre fluidos	19/3/16	25/3/16	5
2	1ª Iteración	26/3/16	8/4/16	10
2.1	Implementar entorno inicial	26/3/16	30/3/16	3
2.2	Generar metabola estática CPU	31/3/16	1/4/16	2
2.3	Generar metabola dinámica, GPU	2/4/16	8/4/16	5
3	2ª Iteración	9/4/16	22/4/16	10
3.1	Diseñar e implementar sistema básico de emisión de fluido	9/4/16	15/4/16	5
3.2	Mejorar metabola dinámica GPU	16/4/16	22/4/16	5
4	3ª Iteración	23/4/16	3/5/16	7
4.1	Generar nuevos comportamientos de fluido	23/4/16	29/4/16	5
4.2	Parametrizar fragment shader del fluido	30/4/16	2/5/16	1
4.3	Exponer aspecto del fluido en el Inspector	3/5/16	3/5/16	1
5	4ª Iteración	4/5/16	19/5/16	12
5.1	Generar nuevos comportamientos de emisión	4/5/16	10/5/16	5
5.2	Interfaz del Inspector	11/5/16	12/5/16	2
5.3	Depurar	13/5/16	19/5/16	5
6	COLCHÓN DE CONTINGENCIAS	20/5/16	30/6/16	30

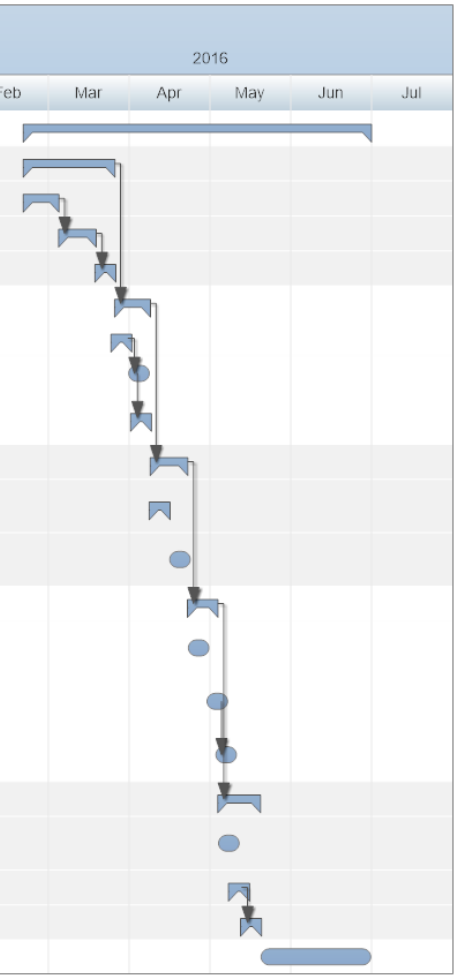


Ilustración 23: Planificación inicial

A continuación, se analiza con más detalle cada una de las iteraciones:

Iteración 1, 10 días

Objetivo: conseguir una primera implementación del fluido dinámico.

Tareas:

1. Generar escena inicial en Unity.
2. Incorporar funcionalidad de movimiento en primera persona para que el jugador pueda moverse por el escenario.
3. Realizar una primera implementación del fluido, estático, sobre la CPU.

4. Realizar una segunda implementación del fluido, ya dinámico, sobre la GPU y sin suavizado de la malla generada.

Iteración 2, 10 días

Objetivo: Conseguir un sistema básico de emisión de fluido dinámico y suavizado.
Tareas:

1. Diseñar la base del sistema de emisión.
2. Implementar el sistema de emisión. Con los siguientes comportamientos de emisión de blobs:
    a. Manteniendo el cick izquierdo del ratón.
    b. Por cada click del ratón.
    c. De forma pasiva, cada vez que transcurra cierta ventana temporal.
3. Implementar el algoritmo de los Marching Tetrahedra para conseguir el suavizado de la malla del fluido.

Iteración 3, 10 días

Objetivo: Conseguir el sistema de evolución de los fluidos y poder cambiar el shader que define su aspecto, así como exponer su configuración en la interfaz.
Tareas:

1. Implementar el sistema de evolución. Con los siguientes comportamientos:
    a. Normal: instancia el fluido en el espacio, sin ningún comportamiento especial.
    b. Físico: este tipo de fluido puede tener gravedad y colisiona con las superficies sin desaparecer.
2. Incluir en el sistema de visualización la posibilidad de cambiar el fragment shader empleado para la definición del aspecto del fluido.
3. Exponer las variables del fragment shader en la interfaz.

Iteración 4, 12 días

Objetivo: Implementar el sistema de pintura y depurar el plugin.
Tareas:

1. Implementar el comportamiento de pintura.
2. Construir una interfaz más intuitiva.
3. Depurar cada uno de los sistemas del plugin.

El tiempo restante hasta la entrega está destinado a trabajar en la escena demo del plugin y a pulir su funcionalidad.

4. Diseño de la solución

En esta sección se va a estudiar el diseño final resultante de todas las iteraciones realizadas, indicando las causas de cada decisión.

Antes que nada, es muy importante destacar que no se va a trabajar en una aplicación escrita puramente en C#. Se va a trabajar en un plugin para Unity cuya funcionalidad va a implementarse más adelante a través de su sistema de scripting en C#. Por tanto, es necesario saber cómo funciona este entorno. Si se realizara el diseño sin haber estudiado antes la plataforma que lo va a soportar es muy probable que el diseño resultante no fuera el más correcto.

4.1 Sobre Unity

En Unity, existe una clase especial llamada MonoBehaviour de la que resulta necesario extender si se desea que un elemento en concreto intervenga directamente en el *game loop*, que es el bucle principal del programa encargado de leer la entrada de usuario, actualizar todos los componentes internos según esta entrada y otras variables, realizar el proceso de render y de mostrar el resultado del raster por pantalla.

La forma que tiene Unity de gestionar los scripts es a través de los *GameObjects*. Un GameObject es una estructura que puede albergar *Components*, que son, entre otras cosas, scripts que heredan de MonoBehaviour. La idea detrás de este diseño es promover la creación de scripts modulares que tengan distintas configuraciones para obtener de esta forma, distintos comportamientos. Cualquier entidad que esté presente en el nivel o escena (esto es, el entorno interactivo donde se encuentra el jugador) se trata de un GameObject.

Resumiendo, para que un sistema intervenga en una escena de Unity se necesita un GameObject y al menos un script asociado a este que participe en el game loop y realice todas las acciones necesarias. Esto crea una importante restricción que hay que tener presente en el diseño: **cualquier script que intervenga en el game loop no puede heredar de ninguna clase**, ya que estará heredando de, posiblemente, MonoBehaviour, y la herencia múltiple en C# no está permitida.

4.2 Criterios de diseño empleados

Se han definido los siguientes criterios para identificar la calidad del diseño ideado. El plugin debe ser:

1. **Configurable** a través del editor de Unity.
2. **Fácilmente extensible**, agregando nuevas clases.
3. **Eficiente** para cumplir las restricciones de tasa de fps impuestas.

Para explicar el diseño realizado se va a analizar primero cada subsistema por separado, contrastando las decisiones tomadas con los anteriores tres criterios y, finalmente, se va a explorar la forma en la que se han conectado estos subsistemas.

4.3 Sistema de emisión

Puesto que la emisión es la fase inicial del proceso se ha optado por crear una única clase, BlobEmitter, que herede de MonoBehaviour, encargada de actuar de puente entre los parámetros señalados por el usuario en el editor y el resto del sistema. Esto quiere decir que todos los aspectos referentes al comportamiento de la emisión de los blobs, de su evolución y de su aspecto, estarán gestionados por BlobEmitter. Esta decisión rompe el principio de *una clase, una responsabilidad* [5] ya que la responsabilidad de esta clase es doble. Por un lado, actúa de emisor de la información del usuario al resto del sistema y por otro, se encarga de aportar el comportamiento de emisión. No obstante, puesto que a nivel conceptual tiene sentido que sea en la emisión del fluido donde se generen todas sus propiedades y en pos de la simplificación se cree justificada la decisión de romper este principio.

A continuación, se va a analizar:

a. La gestión de los distintos comportamientos de emisión.
b. La emisión per se, qué sucede en el momento justo de la emisión de un blob.

Para gestionar los distintos comportamientos posibles se ha seguido el patrón de diseño Estrategia. De esta forma, si se desea un nuevo comportamiento no hay más que crear una nueva clase que implemente los métodos abstractos de EmitterBehaviour y realizar una pequeña gestión en BlobEmitter para desvelar este nuevo comportamiento en el editor de Unity.

Para emitir el blob correcto se ha decidido aplicar el patrón Factory. Esta decisión sigue la dirección propuesta por el criterio 2 ya que al abstraer la creación misma de los objetos Blob se puede redefinir por completo la creación de cada nuevo tipo

creado, si es necesario. El inconveniente es que se necesita crear una clase que herede de Factory para cada nuevo Blob diseñado.

En cualquier caso, se ha creado un *pool* de blobs, una estructura que sirve de almacén de objetos para evitar el sobrecoste de creación y destrucción de los blobs. Esto apoya al tercer criterio de diseño, ya que pueden existir varios emisores de blobs simultáneos que creen y destruyan blobs del mismo tipo constantemente. De esta forma se elimina gran parte del sobrecoste de creación y destrucción de objetos.

La clase BlobCreatorManager es la encargada de gestionar la creación de cada blob en función de los parámetros de comportamiento y aspecto definidos por el usuario en el editor y enviados a través de BlobEmitter.

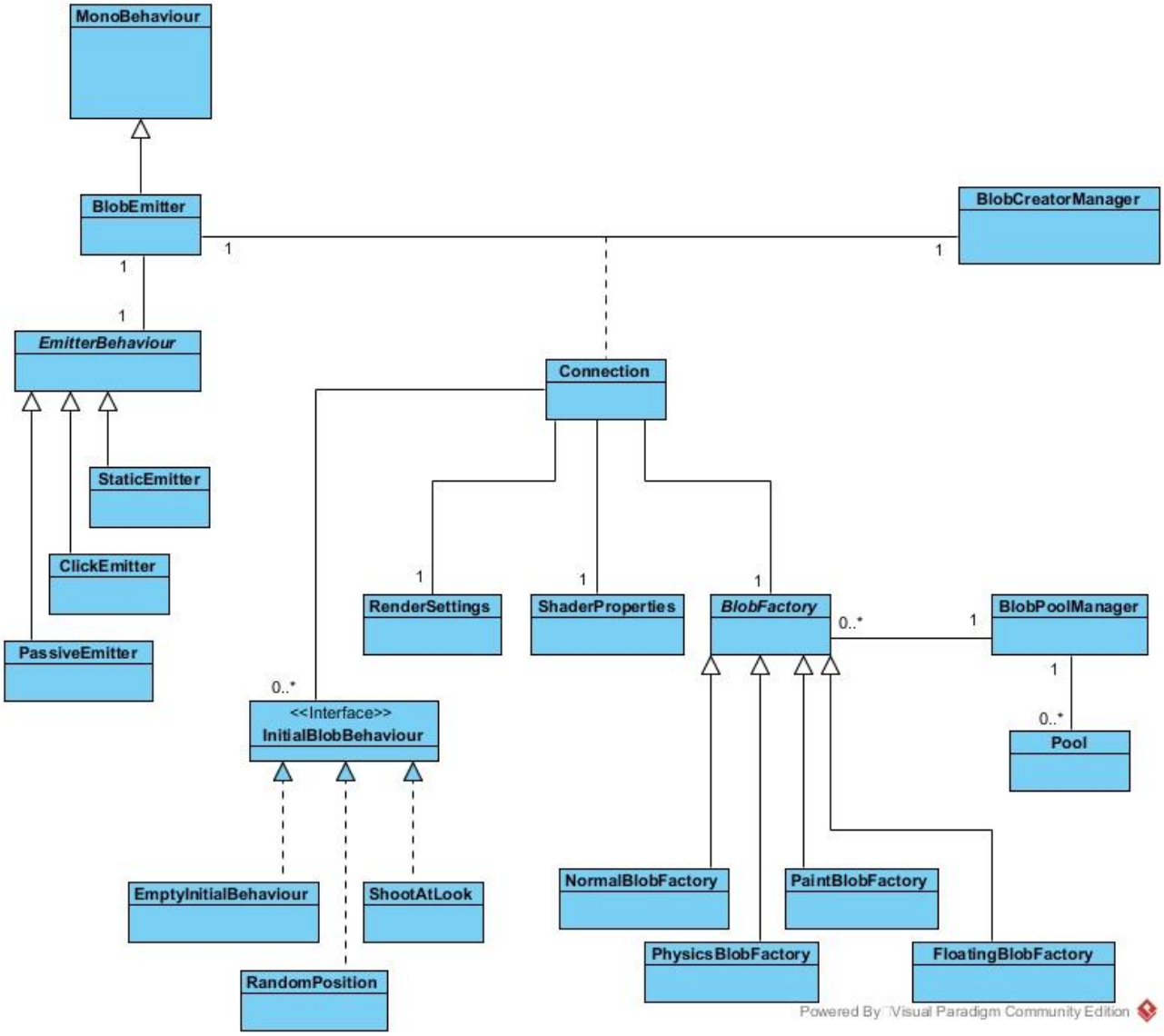


Ilustración 24: Diagrama de clases del comportamiento de emisión

Además, BlobEmitter también decide cuál va a ser el comportamiento inicial del fluido, es decir, qué comportamiento va a seguir cada blob en el momento de su instanciación en la escena. Para hacer esto, se envía a BlobCreatorManager un conjunto de objetos conteniendo los comportamientos iniciales que se ejecutarán más adelante para cada nuevo blob instanciado.

Resumiendo, BlobEmitter se encarga de recopilar la siguiente información:

- Aspecto de los blobs.
- Tipo de blob creado.
- Comportamiento inicial (en la instanciación) de cada blob.
- La forma en la que se genera el fluido.

Y una vez recopilada, distribuye a BlobCreatorManager el tipo (que define el comportamiento de los blobs), comportamiento inicial y el aspecto de los blobs para que pueda generar el fluido correcto e interviene en el game loop emitiendo el fluido de la forma que ha indicado el usuario.

4.4 Sistema de evolución

Debido a que el fluido es un elemento de la escena, cada uno de sus blobs también debe serlo y, por lo tanto, deben de intervenir en el game loop. La primera decisión que se ocurre es emplear varios scripts para cada tipo blob, uno para cada comportamiento, introducirlos todos en un mismo GameObject y gestionar qué script está activo en cada momento por medio de un switch o de una estructura similar. Esta decisión, además de introducir un *code smell* [6], habría ido en contra del segundo criterio de diseño, ya que se estaría duplicando código y no habría una guía clara sobre qué se espera que se implemente para generar un nuevo comportamiento.

En su lugar, se ha decidido crear la clase Blob que no hereda de MonoBehaviour y modela un tipo de comportamiento básico de la cual heredarán el resto de comportamientos. Esta clase Blob será la encargada de instanciar y gestionar el GameObject de cada blob del fluido.

Las distintas funcionalidades de cada comportamiento en este caso se consiguen inyectando *components* (scripts que heredan de MonoBehaviour) a los GameObjects que la clase Blob instancia.

El inconveniente de esta alternativa es el dualismo existente entre código que interviene en el game loop y código que no lo hace, obligando a realizar un

seguimiento en la clase Blob de los GameObject que se crean. La ventaja inherente es que el comportamiento está encapsulado y se pueden crear nuevos comportamientos de forma modular, inyectando los Components que más interesen. A continuación se puede ver el diagrama de clases de este sistema:

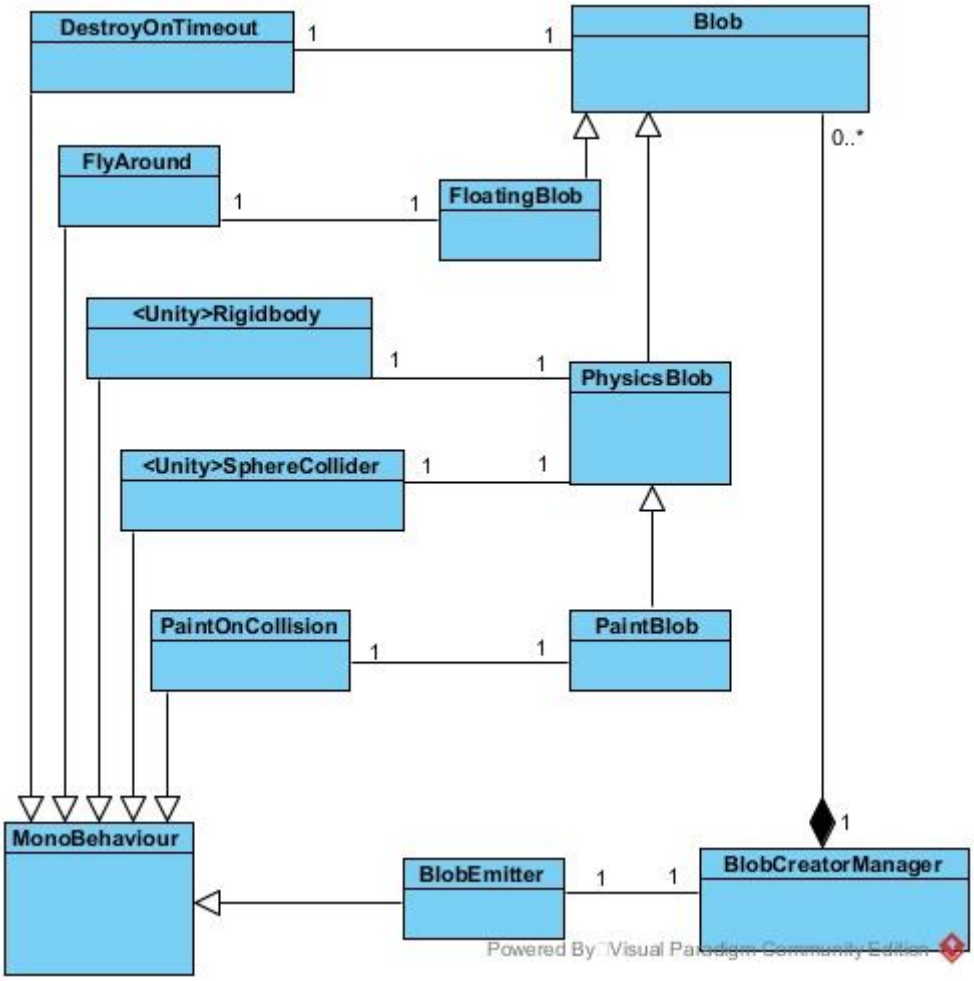


Ilustración 25: Diagrama de clases de tipos de blob

Como se ve, los distintos comportamientos de los fluidos se construyen de forma aditiva a base de extender comportamientos de tipos Blob e introduciendo nuevos bloques de funcionalidad según se necesite. Por ejemplo, tenemos la clase PaintBlob, que extiende el comportamiento de PhysicsBlob y además agrega el comportamiento PaintOnCollision.

4.5 Sistema de visualización

Este es el apartado más delicado a nivel de eficiencia. Aquí es donde se realiza la comunicación con la GPU y el fluido adquiere su forma característica. A nivel de diseño es muy sencillo ya que únicamente tiene que realizar un seguimiento de la posición y

radio de cada blob perteneciente a una metabola y ejecutar el shader correspondiente con estas entradas.

Para conseguir esto existe una clase, MetaballRender que realiza un seguimiento de todos los parámetros que definen el aspecto del fluido y de cada uno de los componentes de renderizado de los blobs. Su misión es gestionar el estado de una metabola, indicando a cada blob cuáles son sus compañeros para que el proceso de generación del fluido sea correcto.

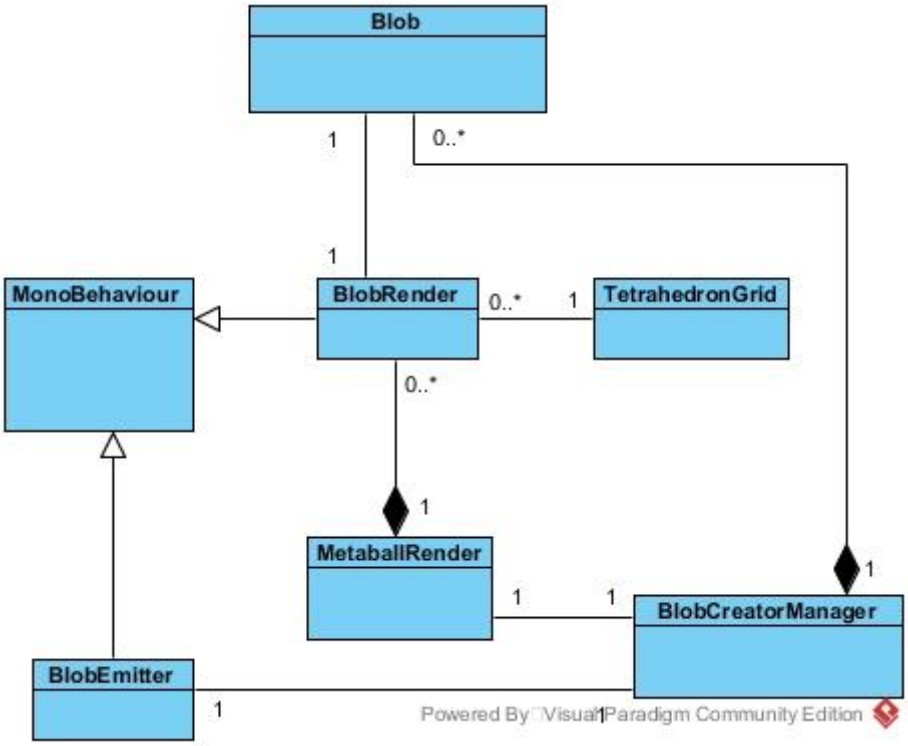


Ilustración 26: Diagrama de clases de visualización

La llamada al shader que genera el fluido, así como la gestión de los distintos tipos de pixel shaders disponibles para la definición de la apariencia del fluido, se realiza en la clase BlobRender.

4.6 Comunicación entre los tres subsistemas

En el ANEXO puede verse el diagrama de clases completo con todos los sistemas y sus conexiones. El punto de comunicación reside en BlobCreatorManager, cuya responsabilidad es gestionar el ciclo de vida de los blobs generados por un BlobEmitter. Para ello:

1. Recoge la información enviada por BlobEmitter referente a:
    a. El aspecto del fluido.

b. Tipo de blobs emitidos.

c. Comportamiento inicial del blob.

2. Envía la información del aspecto del fluido a MetaballRender para que gestione la visualización.

3. Crea los objetos del tipo Blob pertinente con los comportamientos iniciales recibidos y mantiene una referencia de cada uno de ellos, por si resulta necesaria su sustitución o eliminación.

Para concluir con esta sección se va a realizar una demostración de los pasos que habría que realizar para extender el comportamiento de los fluidos. Para ello, es necesario ampliar dos subsistemas: el comportamiento del blob y el comportamiento de emisión.

Para ampliar el primero, es necesario:

a. Crear una nueva clase que extienda a Blob.
b. Crear un nuevo Component (clase que hereda de MonoBehaviour) con la nueva funcionalidad del blob definido.
c. Inyectar al GameObject que generará el nuevo blob el Component creado.

Con estos pasos ya tenemos creado nuestro nuevo tipo de fluido. Ahora hay que ampliar el subsistema de emisión para poder generarlo. Para ello hay que:

a. Extender BlobFactory y definir en la nueva clase la forma de creación del blob (usando la estructura *pool*, el operador *new* o de alguna otra forma que el programador decida).

b. Agregar a BlobEmitter la información necesaria para desvelar la existencia de un nuevo tipo de blob para emitir.

Como se puede apreciar, para agregar comportamientos únicamente hay que agregar información ya que se ha seguido el principio de diseño *"abierto a la ampliación, cerrado a la modificación"* [7]. La gran ventaja de seguir este principio es que a la hora de introducir nuevos comportamientos existe la garantía de que el código ya existente antes de crearlos sigue siendo el mismo, reduciendo notablemente la probabilidad de aparición de bugs y aislando, en caso de que existan, su localización al código recién creado y no al sistema base.

5. Implementación

De los tres subsistemas mencionados (comportamiento de emisión, evolución y visualización), únicamente se va a analizar detalladamente la implementación del subsistema de visualización y de cómo se ha implementado la funcionalidad de pintar las superficies debido a su complejidad. Sin embargo, se considera necesario enumerar el tipo de comportamientos tanto de emisión como de evolución se han implementado:

- **Comportamientos de emisión:**
  - **Mantener click** para emitir el fluido de forma continua.
  - **Pulsar click** para emitir el fluido.
  - **Pasivo**: Se emite un blob cada cierto tiempo.
  - **Estático**: Se generan todos los blobs a la vez, generando un fluido que levita sin ningún tipo de interacción.
- **Comportamientos de evolución:**
  - **Normal**: se instancia el blob en la posición indicada y no tiene ningún tipo de interacción.
  - **Físico**: el fluido colisiona con las superficies y le afecta la gravedad.
  - **Pintura**: como el físico, solo que al impactar con las superficies, las pinta y el blob impactado desaparece.

A continuación, se va a realizar el estudio sobre la implementación del subsistema de visualización realizado, comenzando por los conceptos matemáticos en los que se basa.

5.1 Conceptos matemáticos

Una metabola es, como se ha mencionado anteriormente, un conjunto de blobs. Cada blob se define por dos propiedades únicas: su radio y posición en el espacio. Pero únicamente con estas propiedades no se puede generar la apariencia de gel que se pretende conseguir. Cada blob recibe una interacción del resto de blobs presentes en la metabola, y es con esta interacción con la que se genera tal apariencia.

Puesto de otra forma un poco más formal, una metabola es una forma geométrica que representa gráficamente el umbral de un conjunto de fuerzas escalares dispersas uniformemente en el espacio (en nuestro caso, espacio euclídeo tridimensional).

La forma geométrica de la metabola queda descrita por una isosuperficie, que es un tipo de función implícita que define un valor escalar en el espacio. En nuestro caso, al tratarse de un espacio tridimensional, la isosuperficie que define la metabola toma la siguiente forma:

$$f(i,j,k) = constante$$

O bien, en forma vectorial:

$$f(\boldsymbol{x}) = constante$$

Siendo **x** en este caso el punto (i, j, k) de nuestro espacio.

Para generar el fluido se va a utilizar la siguiente ecuación como base:

$$f(\boldsymbol{x}) = \sum_{i=1}^{N} \frac{r_i^2}{\|\boldsymbol{x} - \boldsymbol{p}_i\|^2}$$

N, $\quad$ $número\ total\ de\ blobs\ contenidos\ en\ la\ metabola$

$\boldsymbol{x}$, $\quad$ $posición\ de\ muestreo\ en\ el\ espacio$

r_i: $\quad$ $radio\ del\ blob\ i$

$\boldsymbol{p}_i$: $\quad$ $posición\ del\ blob\ i\ en\ el\ espacio$

Esta función calcula para cada punto del espacio **x** la suma de fuerzas originadas por cada uno de los blobs. Cada blob define un campo de fuerzas que disminuye conforme se aleja del centro hasta tener un valor despreciable.

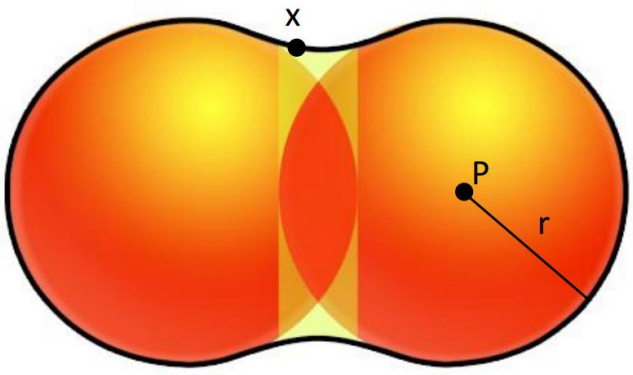


Ilustración 27: Detalle de las variables de la ecuación

5.2 Procedimiento

Para aclarar los conceptos se va a hacer uso de una representación 2D de las metabolas, conceptualmente fácil de extrapolar al 3D, identificando el proceso que se sigue para generar el fluido:

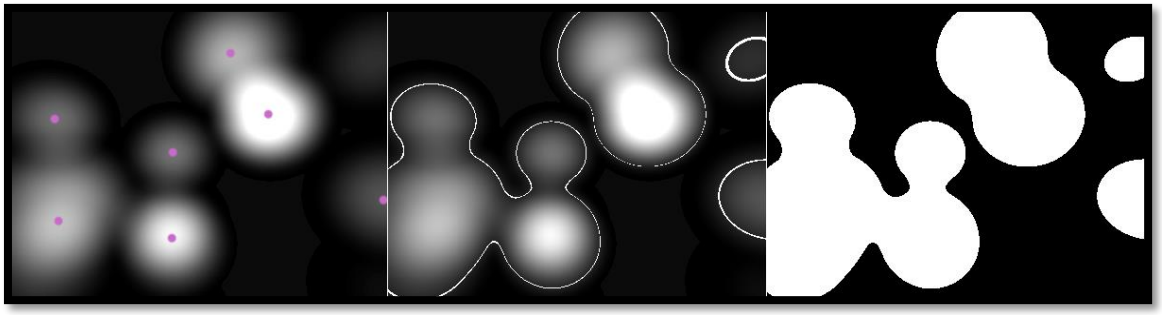


Ilustración 28: Procedimiento de generación del fluido

En la primera imagen se muestra con un punto magenta la posición de cada blob, y se puede intuir el radio de cada uno por el gradiente que emiten. La idea es definir un valor umbral de fuerzas (2ª imagen, línea en blanco que recubre los gradientes de blanco al negro) y aislar las formas que permanecen en el umbral.

Así, se diferencian los siguientes procesos:

1. Disposición de todos los blobs que forman una metabola: definir para cada blob su radio y su posición en el espacio.
2. Calcular el campo de fuerzas escalares en el espacio generadas por todos los blobs en conjunto.
3. Definir un umbral escalar de fuerza.
4. Renderizar la metabola resultante (los puntos del espacio tal que f(**x**) > umbral)

Existen dos dificultades especialmente importantes en este proceso:

a. El cálculo del campo de fuerzas en el espacio. Es necesaria una acotación y discretización espacial.

b. Una vez calculado el campo vectorial, ¿cómo se renderiza la metabola? Hay que tener presente que se va a trabajar sobre un espacio tridimensional a diferencia de las imágenes, así que el fluido generado tendrá que tener volumen. Además, el propósito es que los blobs sean interactivos, por lo que podrán alterar su forma en cualquier momento y, por tanto, aumentar o disminuir de tamaño de forma no uniforme en tiempo real.

Existen varios métodos para implementar este proceso, como se ha comentado en el apartado de análisis:

1) Aplicando el algoritmo de Marching Cubes (MC) o Marching Tetrahedra (MT), sobre la CPU o sobre la GPU [8]

2) Por recubrimiento de partículas, sobre la GPU [3]

La técnica de recubrimiento de partículas se ha descartado por su complejidad. Se ha probado una implementación del MC ya existente sobre la CPU y con unos pocos blobs ya no se cumplían los requisitos de rendimiento, así que se ha decidido apostar por una técnica sobre la GPU. El algoritmo MC se ha descartado por su dificultad de implementación sobre la GPU, por lo que finalmente se ha optado por usar MT, en concreto la técnica mostrada en [8].

No obstante, más adelante se ha descubierto que en Unity esta técnica no funciona correctamente debido a un problema que tiene en su implementación (el paso de vértices a través de la estructura **lineadj** no funciona correctamente), por lo que se han tenido que utilizar **Structured Buffers** para trasladar la información de la CPU a la GPU. Usando esta técnica, podría haberse implementado el algoritmo MC perfectamente pero debido a que la implementación del MT ya estaba en desarrollo se ha decidido continuar con MT.

5.3 Algoritmo: Marching Tetrahedra (MT)

Tanto el algoritmo MC como MT realizan la misma tarea: generan una malla dado un conjunto de fuerzas escalares repartidas por un espacio, y un umbral. El conjunto de fuerzas escalares se calcula a partir de una nube de vértices, resultando al final del proceso un valor de fuerza asociado a cada vértice. La forma en la que se genera la malla es a través de cortes en las aristas, emitiendo geometría en estos puntos de corte. Existen varios tipos de MT tal como se describen en [9] y en cualquiera de las versiones este algoritmo puede darse dos tipos de corte:

a. Aislando un vértice: da lugar a la emisión de un triángulo.

b. Aislando dos vértices: da lugar a la emisión de un quad (dos triángulos, debido a la implementación del dibujado del pipeline gráfico).

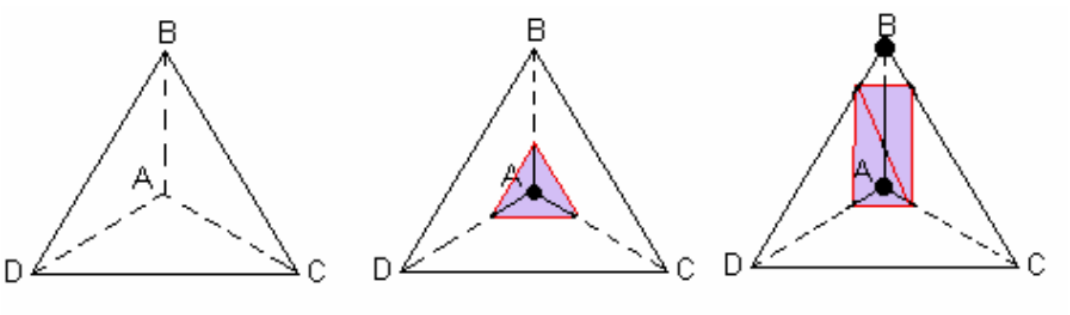


Ilustración 29: Tipos de cortes posibles en el algoritmo MT

En un principio se implementó MT5, que tesela cada cubo en 5 tetraedros. Como ejemplo de configuraciones de teselación se muestra la siguiente imagen:

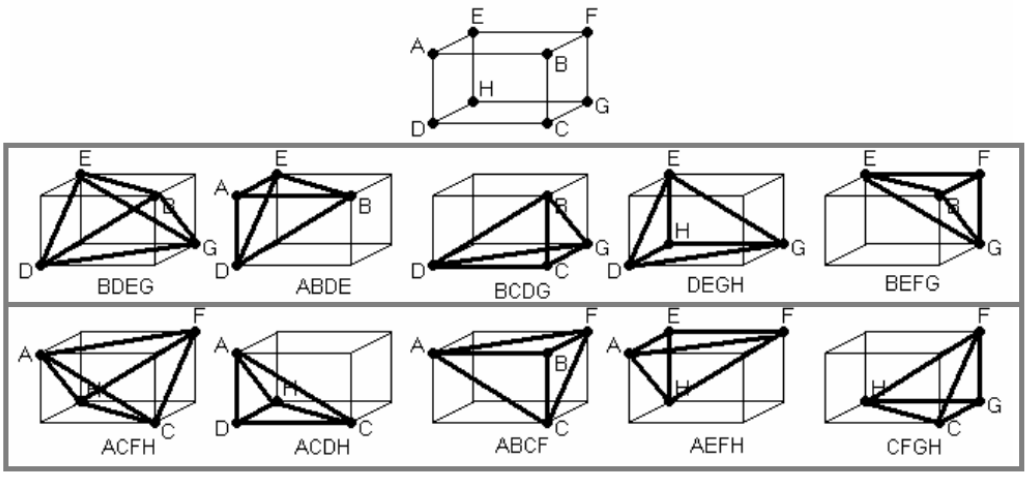


Ilustración 30: Ejemplos de configuraciones de tetraedros para MT5

No obstante, al teselar un cubo en 5 tetraedros existe el problema de que, entre las caras de dos cubos adyacentes, los vértices no coinciden en su posición:

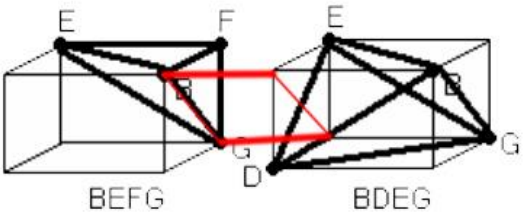


Ilustración 31: Problema de adyacencia de MT5

MT6 soluciona este problema teselando cada cubo en 6 tetraedros en lugar de en 5:

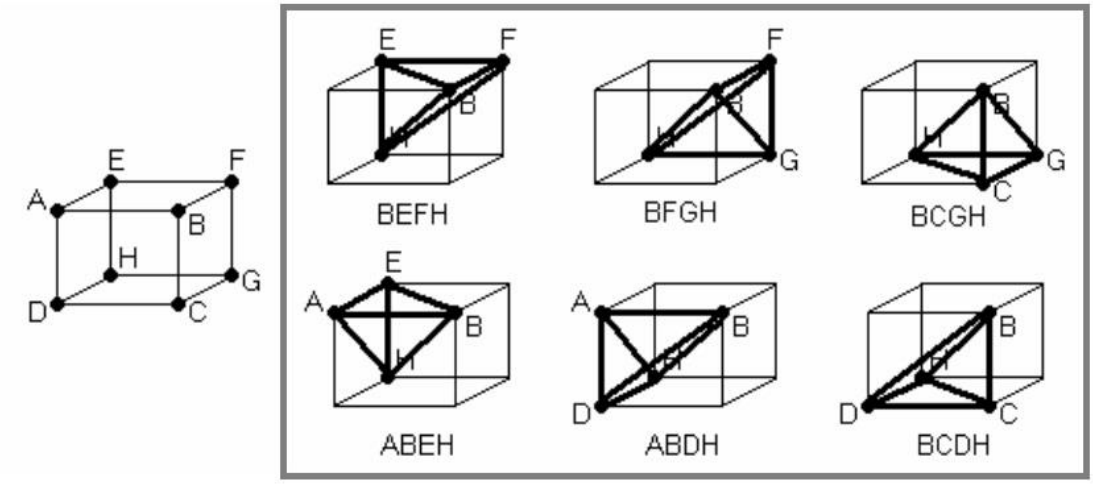
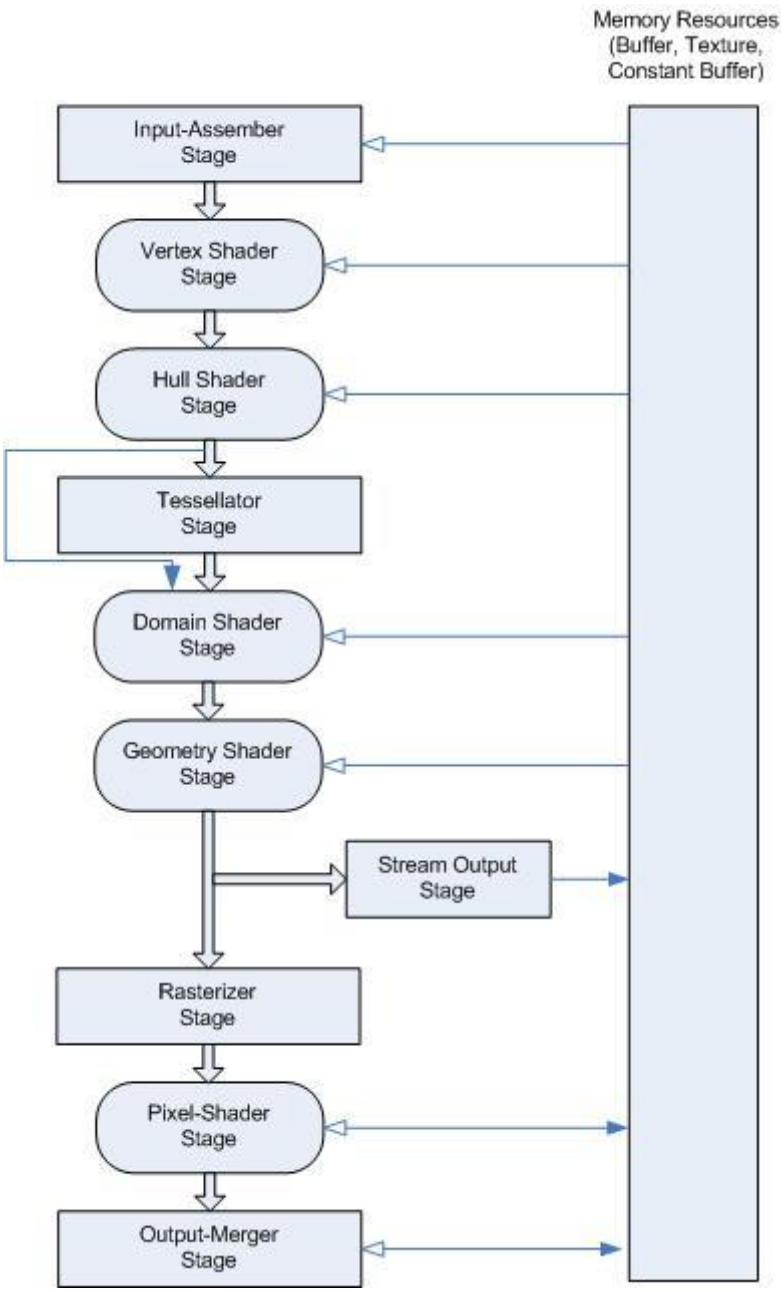


Ilustración 32: Ejemplo de configuración de tetraedros para MT6

5.4 Implementación en la GPU

La gran ventaja de el algoritmo MT es que es fácilmente paralelizable a través de un enfoque SIMD, que es el tipo de paralelismo existente en las GPUs. La generación de la malla se puede realizar por tetraedro. Así, solo se necesita la posición de cada vértice del tetraedro y el conjunto de blobs que pertenecen a la interacción para realizar los cálculos. Como el conjunto de blobs es información global y compartible por cada uno de los nodos de procesamiento de la GPU, cada nodo únicamente necesita conocer los 4 vértices del tetraedro que está procesando para generar los triángulos correspondientes. En conjunto, con todos los tetraedros procesados, el resultado será una malla continua.



Por lo tanto, la GPU deberá:

1. Recibir un conjunto de tetraedros, representados cada uno como cuatro vértices.
2. Calcular el valor escalar del campo de fuerzas que han generado los blobs en cada uno de los vértices del tetraedro.
3. Aplicar el algoritmo MT para generar la malla.
4. Colorear (sombrear) la malla generada.

Antes de empezar a estudiar cómo abordar cada una de estas etapas se va a realizar un breve repaso sobre las implementaciones gráficas existentes. En las primeras no existía la posibilidad de programar nada, únicamente existía una funcionalidad fija (*fixed function pipeline)* y se configuraba acorde a las necesidades del programa. Esto daba lugar a la imposibilidad de conseguir efectos nuevos no implementados en tales funciones. Más adelante se implementó el ***Shader Model 1.0*** (en adelante, SM), que introdujo etapas configurables del pipeline gráfico a través de pequeños programas conocidos como sombreadores o *shaders*.

Inicialmente había dos: el ***vertex shader*** y el ***pixel (o fragment) shader***. De forma muy simplista, con el primer tipo se pueden realizar cálculos por vértice mientras que con el segundo se realiza cálculos por píxel. A partir de aquí se ha ido mejorando la implementación, ampliando el juego de instrucciones y el tipo de shaders programables. Si nos quedamos únicamente con estos dos shaders, notamos que falta algo para poder solucionar el problema: no hay forma posible de generar geometría. Se necesita otro tipo de shader opcional, el ***geometry shader***, disponible a partir del SM4.0. Hay otros shaders típicos del SM5.0 como el Hull y el Domain shaders que también son opcionales y no se usan en nuestro caso.

Como es típico en un pipeline, lo que un estado devuelve el siguiente lo toma como entrada. Con los shaders sucede lo mismo, por lo que existe un orden de ejecución. En nuestro caso concreto primero se ejecutará el vertex shader, seguido del geometry shader y, finalmente, del pixel shader.

Realizado este pequeño repaso sobre la arquitectura moderna de las GPU, se puede tratar de abordar cada una de las etapas a resolver de nuestro problema:

5.4.1 Etapa 1: La GPU recibe un conjunto de tetraedros

En esta etapa se ha decidido crear una rejilla de tetraedros en la CPU que conformará el espacio sobre el que se generará el fluido. La creación de esta rejilla es crítica ya que según la disposición de los tetraedros se estará implementando un tipo de MT (MT5 o MT6, por ejemplo) y el tamaño de los tetraedros tendrá una repercusión directa en la resolución del fluido. Para crear la rejilla, se genera un cubo virtual de cierto tamaño y, con un paso definido, se discretiza su volumen en puntos del espacio. Cada uno de estos puntos define el centro de un pequeño cubo, y cada uno de estos cubos contiene los tetraedros que se utilizarán para el cálculo. Todo esto, en conjunto, genera el volumen de renderizado que necesita el shader para muestrear el campo de fuerza escalar.

Para controlar la resolución del fluido es necesario gestionar el tamaño del volumen de renderizado y el paso. El tamaño de este volumen se calcula a través de varias de las propiedades del fluido de forma indirecta y el valor del paso se genera a partir de un parámetro nombrado *resolution*, que define el número de divisiones que se van a realizar al gran cubo inicial.

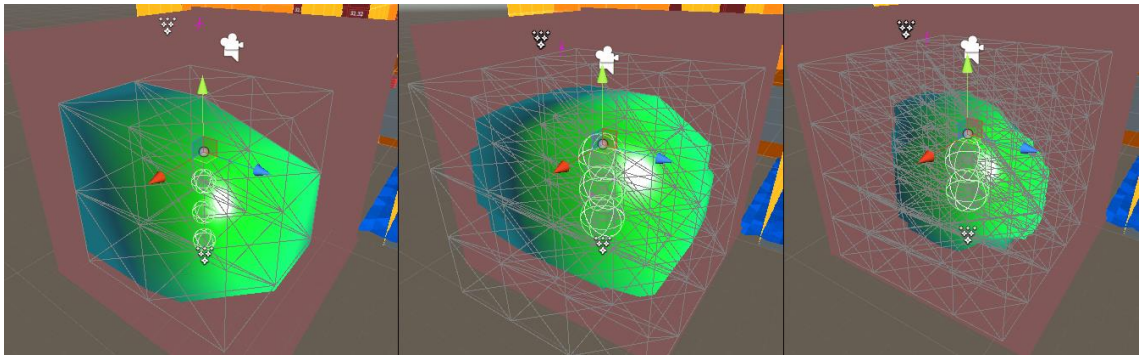


Ilustración 33: Valores de resolución adoptados, de izquierda a derecha: 1, 2, 3

Una vez creado el volumen de renderizado, hay que enviarlo a la GPU para que actúe de entrada en el vertex shader.

La CPU almacena esta rejilla de vértices con una estructura concreta para que la GPU la entienda como un conjunto de tetraedros como grupos de cuatro vértices y los traslada a la GPU utilizando un *Structured Buffer*, que es una estructura que permite el almacenamiento de datos arbitrarios en la GPU. El uso de esta estructura crea una restricción de hardware en la que se excluyen todos los procesadores gráficos que no soporten el *Shader Model 5* (Direct3D 11, OpenGL ES 3.1).

La disposición de los datos en el Structured Buffer resulta de vital importancia por dos motivos:

1. Existen casos de incompatibilidad entre Direct3D y OpenGL a la hora de interpretar el Structured Buffer debido a la estructura interna de datos de cada implementación: Direct3D permite estructuras desalineadas en memoria mientras que OpenGL no. En caso de memoria desalineada, Cg tendrá problemas al compilar sobre OpenGL.

2. Una disposición desalineada de los datos en Direct3D puede afectar negativamente al rendimiento[6].

Ambos motivos están relacionados y se puede evitar cualquier problema siguiendo la siguiente regla:

[6] https://developer.nvidia.com/content/understanding-structured-buffer-performance

“En el Structured Buffer solo tienen que almacenarse estructuras que tengan un tamaño múltiplo de 128 bits (16 bytes)”

5.4.2 Etapa 2: Cálculo del campo de fuerzas

Se realiza dentro del vertex shader y por cada vértice de cada tetraedro que se recibe se calcula el valor escalar del campo.

5.4.3 Etapa 3: Algoritmo Marching Tetraedra

Ocurre en el geometry shader. Por cada tetraedro recibido, primero comprueba si se encuentra en una zona de frontera definida por el umbral, y en caso afirmativo calcula los puntos de intersección por medio de una interpolación entre pares de vértices según su fuerza escalar y emite de uno a dos triángulos, según el caso. Si el tetraedro está totalmente dentro o fuera de la frontera, no se emite ningún triángulo.

Es importante destacar aquí el valor umbral. Se le ha nombrado *isoLevel* y adopta valores del rango [0, 1]. Se ha decidido que se tengan en cuenta para la generación de la malla del fluido todos los vértices **x** tales que f(**x**) > isoLevel. Hay que tener presente que el valor de f(**x**), contemplando un único blob, será infinito positivo en el centro del blob, uno en la frontera con su radio y menor que uno más allá de su radio. Este parámetro junto con el radio permite bastantes configuraciones distintas del fluido:

- **Si isoLevel < 1:** reducimos su frontera
- **Si isoLevel = 1:** su frontera coincide con el radio.
- **Si isoLevel > 1:** aumentamos su frontera.

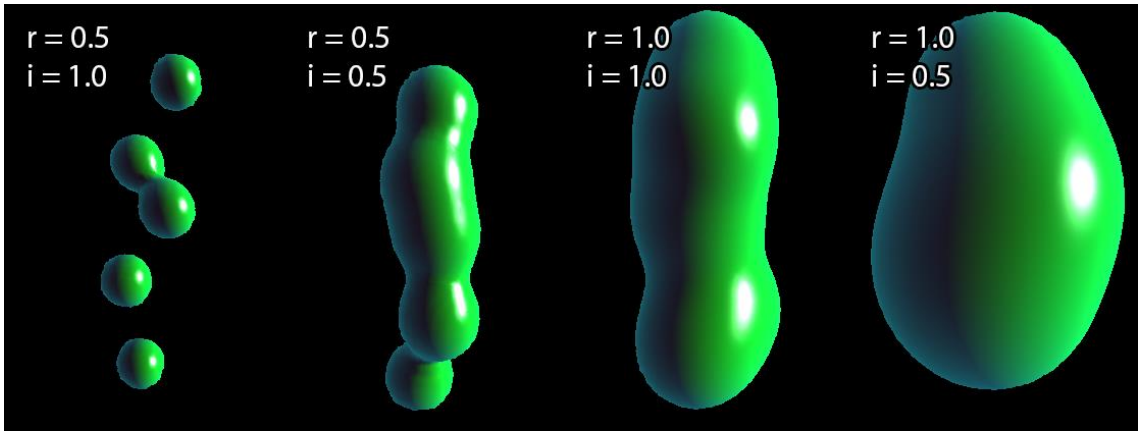


Ilustración 34: distintos valores de IsoLevel (i) y Radio (r)

Una vez generada la malla, se envía la información pertinente al pixel shader.

5.4.4 Etapa 4: Sombreado de la malla generada

Se aplica un modelo de sombreado Phong (esto es, se calcula la iluminación por cada fragmento en el pixel shader). Se podría haber utilizado el modelo Gouraud, que se calcula por cada vértice: parte del sombreado podría calcularse en el geometry shader y resultaría más rápido, pero de peor calidad. Por este motivo y por localizar en cada tipo de shader una etapa del algoritmo, se ha decidido calcular el sombreado en el pixel shader.

La idea es poder disponer de tantos sombreados posibles como se desee, por lo que se ha ideado un sistema que permite alternar distintos pixel shaders basándose en directivas del compilador y en definiciones externas.

En cualquier caso, se ha diseñado un pixel shader por defecto lo suficientemente versátil como para generar fluidos de varios aspectos diferentes. A continuación, se va a separar cada uno de los componentes de iluminación de este shader y a mostrar el efecto resultante al combinarlos todos:

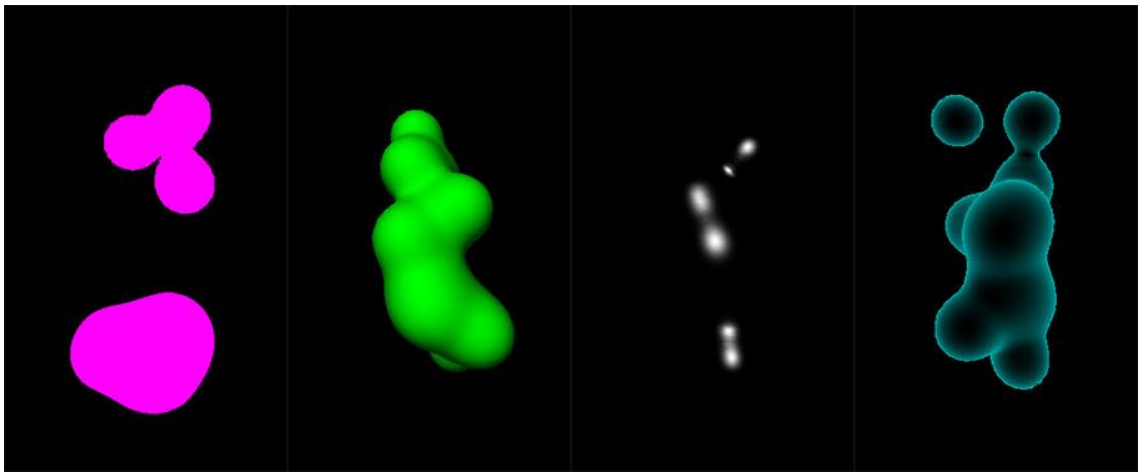


Ilustración 35: Componentes de iluminación por separado

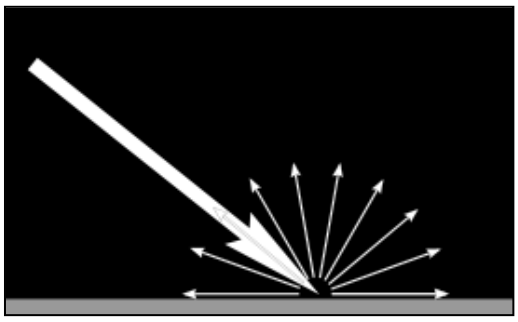
De izquierda a derecha se muestran los componentes: emisivo, difuso, especular, fresnel (cóncavo). El cálculo de cada uno de estos componentes se detalla a continuación:

Componente emisiva: modela el color que emite el cuerpo, se calcula de forma simple con la ecuación: $I_{emisiva} = k_e \cdot C_e$, siendo k_e el coeficiente de reflexión emisiva del material y C_e el color reflejado.

Componente difusa: modela el comportamiento de la luz reflejada por la superficie de un objeto de forma adireccional. La intensidad en cada punto de la superficie es independiente de la posición del observador.

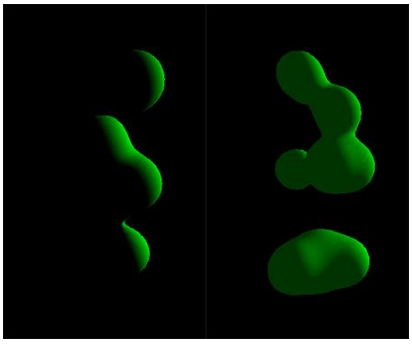
$$I_{difusa} = I_L \cdot k_d \cdot \cos(\theta) \cdot C_d$$

Siendo I_L la intensidad del foco de luz que interviene, k_d el coeficiente de reflexión difusa del material, θ el ángulo que forman los vectores de luz incidente $\vec{L}$ y normal $\vec{N}$ de la superficie, y C_d el color difuso del material. Considerando que ambos vectores, el de luz incidente y el vector normal, son unitarios, la expresión anterior puede simplificarse teniendo en cuenta que: $\vec{L} \cdot \vec{N} = |\vec{L}||\vec{N}| \cos(\theta) = \cos(\theta)$ resultando así la siguiente expresión:



$$I_{difusa} = I_L \cdot k_d \cdot \vec{L} \cdot \vec{N} \cdot C_d$$

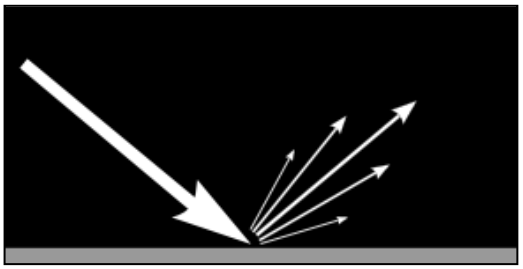
Considerando el aspecto práctico, $\cos(\theta)$ puede adoptar valores negativos. Cuando el valor del coseno pertenece al rango [-1, 0], esto se traduce a que, en ese punto del material, no incide ninguna luz. Para conseguir sombra absoluta y evitar los problemas que surgen con los valores negativos en la expresión, se puede realizar una saturación en cero o en otro valor superior a cero, si no se desea una sombra absoluta. Para modelar este caso de ausencia de sombra absoluta, se ha incluido una variable más, *albedo*, para simular el resto de luz existente en la sala. Así pues, la fórmula final será la siguiente:



$$I_{difusa} = I_L \cdot k_d \cdot \max(albedo, \vec{L} \cdot \vec{N}) \cdot C_d$$

La imagen anterior a la derecha muestra el efecto que tiene la variable *albedo*. El fluido de la izquierda tiene un *albedo* con valor 0.0, mientras que el de la derecha tiene un valor de 0.2.

Componente especular: Modela la capacidad del material de reflejar la luz. En un objeto liso, la reflexión especular será mayor cuanto más pulido esté y será máxima para el espectador cuando el ángulo de reflexión de la luz coincida con el ángulo de visión. De otra forma, cuando el vector de visión $\vec{V}$ coincida con la dirección del vector de



reflexión $\vec{R}$. En este caso, la reflexión es direccional y dependiente del espectador. La ecuación que modela este efecto y asume que $\vec{R}$ y $\vec{V}$ son unitarios es la siguiente:

$$I_{especular} = I_L \cdot k_s \cdot cos^n(\alpha) \cdot C_s = I_L \cdot k_s \cdot (\vec{R} \cdot \vec{V})^n \cdot C_s$$

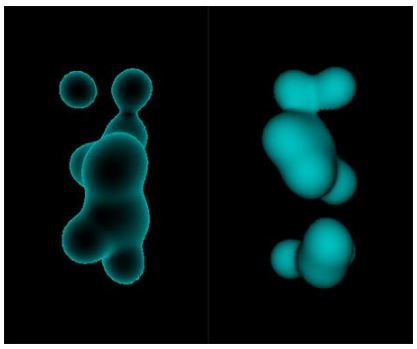
Siendo k_s el coeficiente de reflexión difuso, *n* un valor para tener control sobre el brillo (a mayor *n*, el brillo estará más concentrado), α el ángulo existente entre $\vec{R}$ y $\vec{V}$, y C_s el color especular. Como en el caso de la reflexión difusa, el coseno tiene un rango negativo que no interesa y es necesario saturar.

Componente fresnel: Modela un incremento de reflexión de la luz en las superficies que tienen una normal perpendicular al vector visión $\vec{V}$. El fresnel convexo se calcula de la siguiente forma:

$$I_{fresnel} = \left(1 - \max\left(\vec{N} \cdot \vec{V}, 0\right)\right) \cdot C_f$$

Siendo C_f el color del fresnel.

Además, se ha incluido en el código una variable de control sobre la convexidad del fresnel para poder conseguir distintos efectos.



No hay que confundir un fresnel cóncavo con el componente difuso de la iluminación. El fresnel depende del vector visión mientras que el componente difuso depende del vector de luz incidente.

Para finalizar, en el pixel shader se realiza una suma de los cuatro componentes anteriores que intervienen en el proceso de sombreado y se devuelve el color del fragmento calculado.

Algunos de los resultados que se pueden conseguir variando los parámetros son los siguientes:

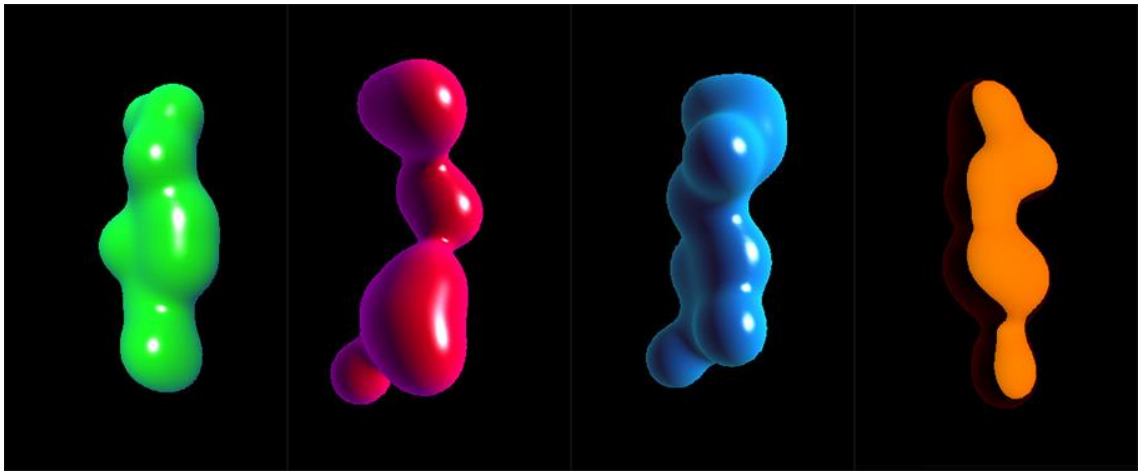


Ilustración 36: Ejemplos de aspectos que se pueden conseguir con el fragment shader creado

Puesto que la obtención de los distintos aspectos se ha conseguido únicamente modificando parámetros, se pueden crear interpolaciones para conseguir esta alteración visual en tiempo real.

Actualmente el shader está diseñado para aceptar solo una luz direccional, utilizando *forward rendering* [10]. Se puede modificar para que acepte más luces, pero se ha decidido dejarlo como está ya que el aspecto resultante para el prototipo es más que aceptable.

5.5 Sistema de pintura

Una de las especificaciones del sistema era poder pintar las superficies con los fluidos. Para pintar superficies se pueden utilizar decals, que son imágenes proyectadas sobre la superficie. La implementación más sencilla de esta técnica es demasiado simplista, debido a que no se adapta bien a superficies complicadas, y la más compleja, demasiado costosa debido a las operaciones de proyección que es necesario hacer, por lo que la técnica de los decals no es una opción viable. La alternativa es pintar directamente sobre la textura. La técnica más básica consiste en calcular las coordenadas uv de la textura de la superficie a partir del punto de impacto del fluido y pintar directamente un área de la textura. Los problemas que tiene esta técnica son:

1. Al pintar directamente sobre la textura, se pierde la información del color original, por lo que la pintura impresa no puede borrarse de ninguna forma.

2. Si existe algún tiling de la textura, al tomar las coordenadas uv y pintar directamente sobre la textura base, se genera esa pintura en todos los tiles y no solo en la superficie impactada.

3. En los bordes de la superficie existen condiciones de frontera en las que el área que se desea pintar es más grande que el área hábil seleccionada para ser pintada. Sería necesario realizar un recorte.

4. El coste de la impresión. Si se desea imprimir una imagen, hay que realizar un reemplazamiento o algún tipo de blending entre tal imagen y una zona concreta la textura de la superficie. Esto supone recorrer todos los píxeles de la imagen y de una zona de la textura de la superficie y realizar un pequeño proceso. Si la emisión del fluido es uniforme y el impacto contra las superficies es constante, el coste computacional puede ser muy elevado y suponer un cuello de botella importante que conviene evitar en el sistema.

Una alternativa que evita los dos primeros problemas es crear un shader especial que utilice, además de la textura base, una segunda textura, generada al vuelo e inicialmente totalmente transparente. Cuando un fluido impacte una superficie, se pintará sobre esta textura. Esta solución, además de evitar los dos problemas anteriores, otorga una nueva funcionalidad: permite crear fácilmente superficies no coloreables. Para ello, simplemente basta con que la superficie no se renderice con este shader.

Aún queda resolver los problemas 3 y 4. Su origen radica en el hecho de imprimir la pintura directamente dado un punto de colisión de la superficie. Una alternativa a la copia y reemplazo constante de texturas es utilizar las RenderTexture, que es un tipo especial de textura que provee Unity. Estas texturas están diseñadas para que se pueda renderizar sobre ellas de forma eficiente. Esto es, se puede instanciar una cámara, asignarle a la salida una RenderTexture en lugar de la pantalla y todo lo que renderice esa cámara será inmediatamente visualizado en la RenderTexture. Utilizando esta configuración, se puede definir un lienzo enfocado por una cámara que renderice a una RenderTexture. Cuando un fluido impacte sobre una superficie coloreable por primera vez, se crea una RenderTexture y se configura en la cámara. Hecho esto, se instancia un quad con la textura de pintura que se desea imprimir, en la posición correcta del lienzo calculada a partir de las uv obtenidas en el impacto. Para generar el tamaño correcto de la RenderTexture se tiene en cuenta el tamaño de la textura base y su tiling. Si el tiling es asimétrico es posible que la textura de pintura impresa aparezca con un efecto de estiramiento. Para evitar estos casos, es posible crear la RenderTexture a mano con el tamaño deseado y vincularla en la superficie.

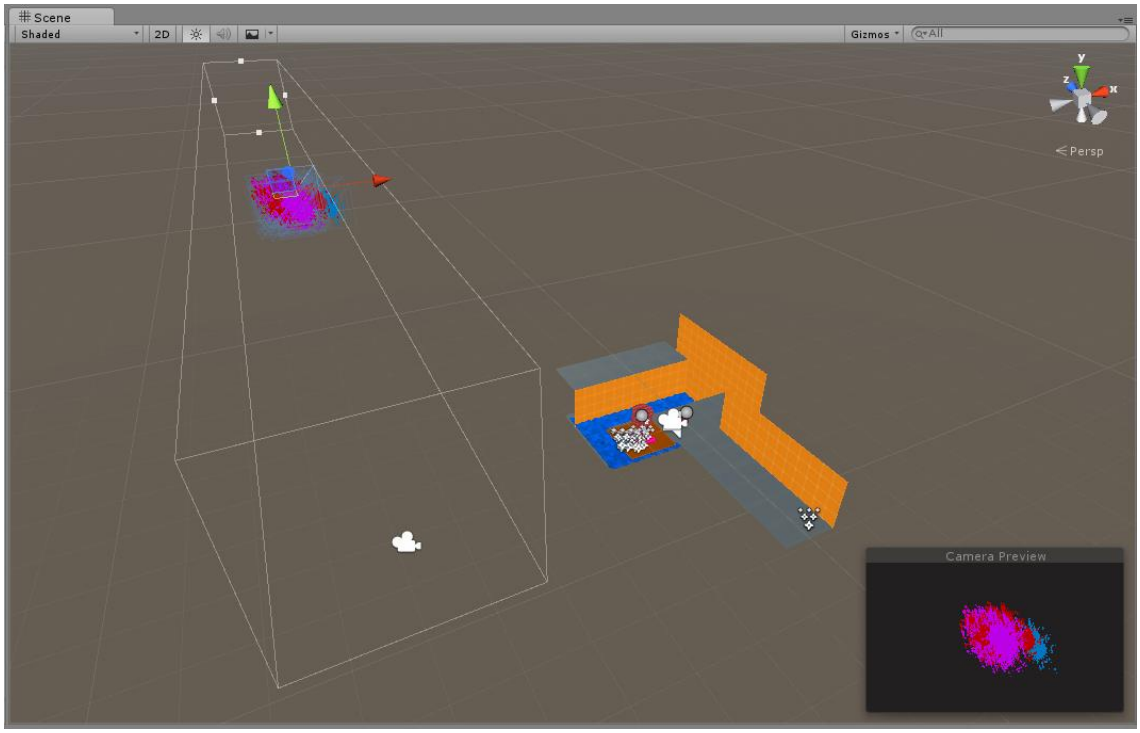


Ilustración 37: Configuración cámara y lienzo

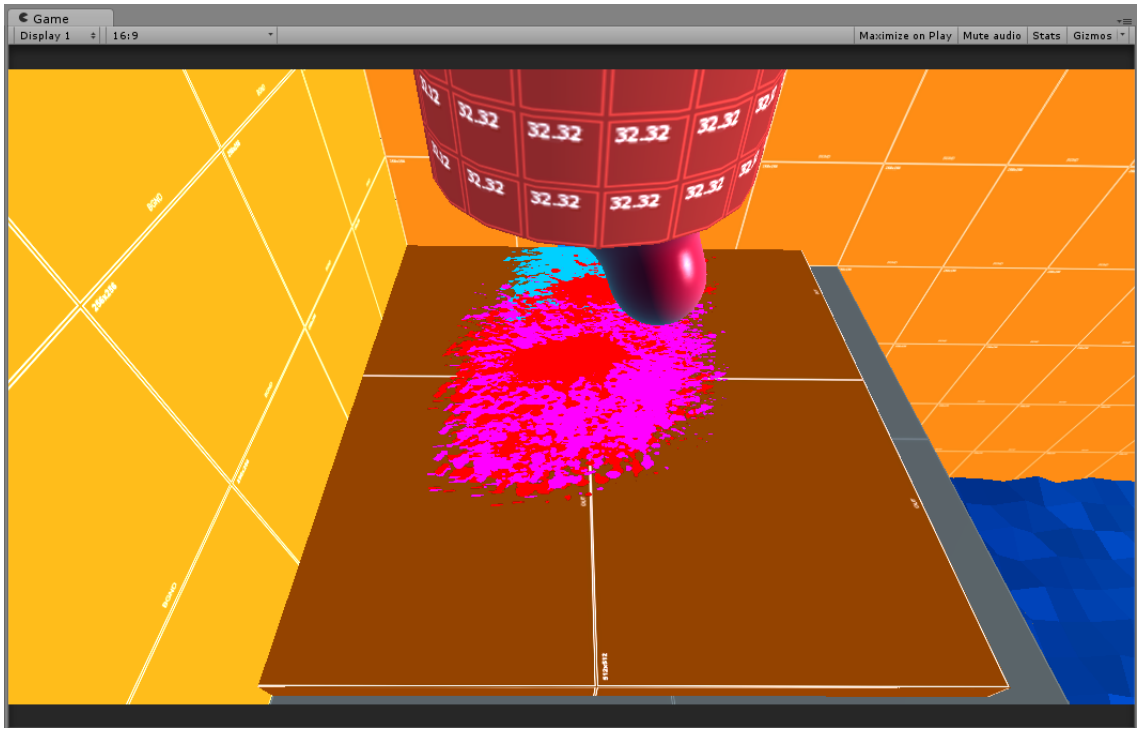


Ilustración 38: Resultados del sistema de pintura

Veamos cómo este método soluciona los anteriores problemas 3 y 4. El problema 3 se soluciona automáticamente: las condiciones de frontera ahora desaparecen ya que la cámara encargada de recoger las manchas de pintura a aplicar es la encargada de realizar el recorte. El problema 4 también se soluciona también de manera instantánea: la pintura que se va a imprimir ahora mismo es un material asociado a un

quad. Si se desea un blending en concreto de la pintura con la textura base, simplemente hay que modificar el shader de las superficies coloreables.

No obstante, para usar correctamente este método es necesario tomar decisiones sobre tres aspectos importantes:

1. Hay que definir qué sucede cuando existen impactos sobre superficies distintas.
2. La generación de quads tiene que estar controlada: no se puede esperar que el sistema genere quads de forma ilimitada y siga cumpliéndose la especificación de rendimiento del plugin.
3. Cómo recuperar el color de la textura base: pintura que limpia.

Si se configura la cámara para que no limpie el color de fondo sino únicamente los valores de profundidad, se puede cambiar de superficie de forma sencilla. Al impactar sobre la nueva superficie, se eliminarían los quads con la información de pintura de la anterior superficie, se creará una RenderTexture asociada a esta nueva superficie en caso de que no existiera, y se renderizaría sobre esta. Al impactar sobre una superficie con una RenderTexture asociada, esta se recupera y la cámara renderizará sobre ella. Al no limpiar el color de fondo, todos los colores ya existentes en la RenderTexture permanecerán, y los nuevos decals que renderice la cámara sobreescribirán los valores de color antiguos de la RenderTexture. Este sistema introduce la limitación de que no se pueden emplear shaders para las pinturas que modifiquen su color constantemente a lo largo del tiempo ya que, en el momento del cambio de superficie, perderían su dinamismo.

Sobre la generación controlada de quads, basta con limitar la instanciación una vez alcanzado cierto umbral. Si bien es cierto que al no limpiar el color podría usarse un único quad se ha considerado una buena práctica poder instanciar varios quads a la vez por si en algún momento se desea crear materiales que realicen algún efecto breve de aparición en la escena.

Para finalizar, la pintura que borra se puede realizar de forma muy sencilla con un tipo especial de quad de pintura. Este quad tendrá un shader que toma de entrada una textura como máscara y que, al imprimirse sobre la superficie, se encarga de limpiar el color en las zonas donde el alfa de su máscara es mayor que cero. Para poder limpiar el color, no tiene que existir ningún quad de pintura normal que intervenga en el proceso, porque entonces se limpiaría el color de la cámara, pero este quad repintaría

su área, realizando imposible el borrado. Así pues, el borrado se efectúa sobre el color de la cámara y no sobre los elementos de la escena.

Existe una alternativa de implementación usando una cámara que limpie el color. Su implementación es menos intuitiva, más costosa y genera problemas en el borrado de pintura. Esta alternativa consiste en mantener el estado de color de los quads eliminados en una textura de reserva. Para ello, se realiza un guardado de la RenderTexture de la cámara en cada cambio de superficie y se recupera más adelante, cuando se vuelve a pintar sobre tal superficie. El shader de la superficie será el encargado de realizar el blending correspondiente entre la RenderTexture de la cámara y la textura de reserva. Aquí hay un grave problema de eficiencia en el borrado de pintura ya que, para ello, es necesario realizar un guardado de los quads en la textura de reserva y borrar directamente el color de ella. Además, el proceso de borrado es independiente y su implementación difiere del proceso de pintado. No es una buena técnica a emplear. No obstante, y debido a que se realizó su implementación en un primer momento antes de utilizar el sistema de cámara que no limpia el color, se ha decidido agregar la explicación de esta implementación en el ANEXO por emplear una tecnología interesante: los Compute Shaders.

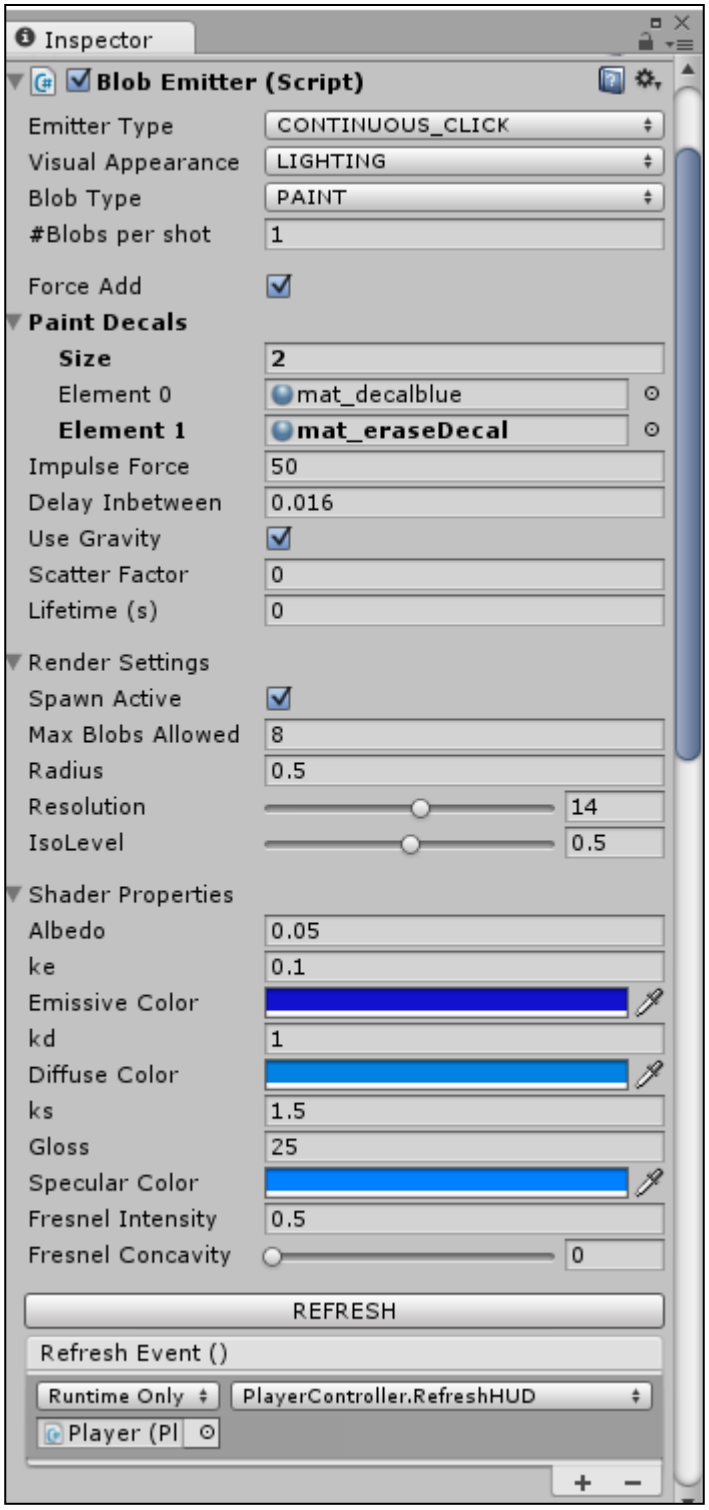
5.6 Interfaz

Hay dos formas de definir qué variables de los scripts se pueden acceder desde el editor de Unity. Una es automática, y consiste en definir en el propio script variables públicas o hacerlas serializables y Unity se encarga de exponerlas. La segunda forma trata de definir de forma explícita, en una nueva clase, qué variables se exponen y de qué forma (una caja de texto, un slider, un botón, etc). Se ha decidido utilizar esta última debido a que con esta segunda opción se pueden crear interfaces más flexibles.

En la imagen a continuación puede verse la interfaz de acceso a los parámetros del fluido. Son modificables el tipo de emisor, de blob y la apariencia. Según estos aspectos se mostrarán unas opciones u otras, dependiendo de la funcionalidad que pueda ofrecer cada uno. Por ejemplo, para el caso de mantener el click de ratón para generar el fluido que pinta su entorno, se tiene acceso a las siguientes propiedades:

- **Nº de blobs generados por disparo**.
- **Forzar inserción**: no espera a que se destruyan los blobs en caso de que se haya alcanzado el máximo permitido, sino que sustituye al más antiguo.

- **Decals de pintura**: Son los materiales que se instanciarán en la configuración cámara-lienzo para generar la pintura.
- **Fuerza de impulso**.
- **Espera intermedia**: Esto es el tiempo que tiene que transcurrir entre la instanciación de dos blobs.
- **Gravedad**: Indica si el blob es afectado por la gravedad o no.
- **Factor de dispersión**: Dado un punto en el espacio, genera los blobs alrededor de ese punto si el valor es mayor que cero, o exactamente en ese punto en caso contrario.
- **Tiempo de vida**: tiempo que tiene que pasar a partir del cual el blob se destruirá automáticamente. Si el valor indicado es 0, no se destruirá.



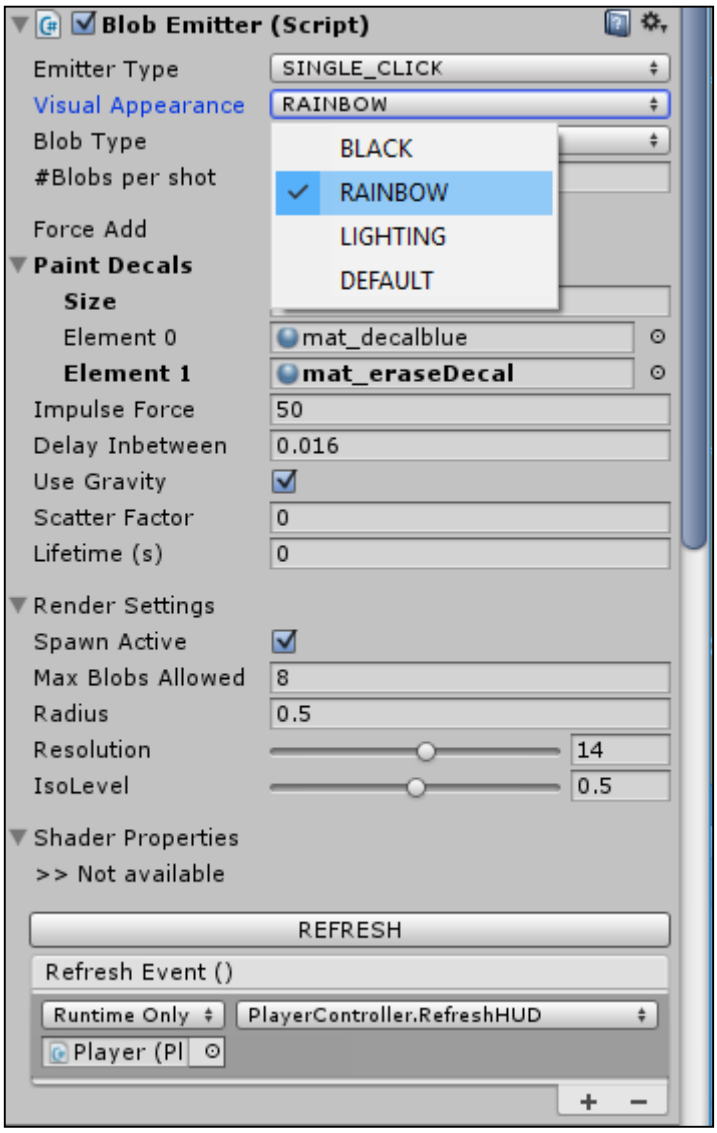
Dentro del grupo de la configuración de los aspectos del render, se tiene:

- **Spawn (instanciación) activo**: Abre o cierra el flujo de blobs. La diferencia con desactivar el script entero es que, con esta propiedad, los blobs generados y aún no destruidos siguen su curso.
- **Máximos blobs permitidos**.
- **Radio de los blobs**.
- **Resolución**.
- **IsoLevel**.

Y, finalmente, se pueden configurar las propiedades del sombreado explicadas en el apartado 5.4.

Existe un botón para actualizar el comportamiento del emisor, trasladando los cambios de las variables del emisor del fluido realizados en la interfaz al sistema en ejecución. Además, se ha definido un pequeño selector de métodos a ejecutar en el momento que se pulse el botón. Esto puede servir, por ejemplo, para realizar cambios pertinentes en el HUD del jugador, cambiando el feedback visual del tipo de blob que se ha definido recientemente.

La interfaz tiene un comportamiento dinámico. En el momento que se cambia el tipo de apariencia o de emisor, también se cambian las propiedades disponibles para su configuración. En caso de que no haya ninguna, se muestra una pequeña etiqueta de texto indicando la situación.



6. Resultados y conclusiones

En un principio se implementó el algoritmo MT5, pero esta elección dio lugar a los siguientes artefactos gráficos debidos a la ausencia de simetría entre caras de cubos adyacentes ya comentada:

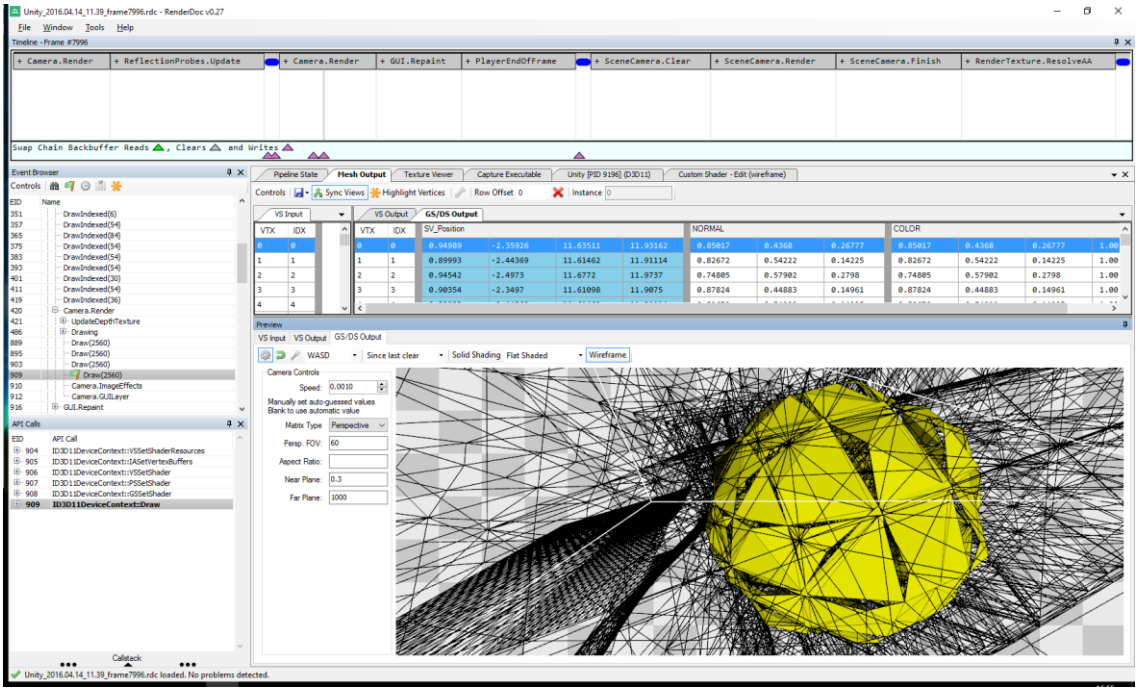


Ilustración 39: imagen del debugger de shaders RenderDoc que se puede vincular con Unity mostrando los problemas de MT5

Utilizando MT6, que cambia la configuración de la teselación para cada cubo de 5 tetraedros a 6, se ha conseguido el efecto de malla suavizada deseado.

El gran inconveniente de usar MT6 reside en la gran cantidad de triángulos que genera (la malla está demasiado teselada) [4], pero existen alternativas como RMT [11] que mejoran considerablemente el resultado. Al no existir límites en la cantidad de argumentos de entrada al shader gracias al uso de los Structured Buffers, esta optimización es posible y queda pendiente.

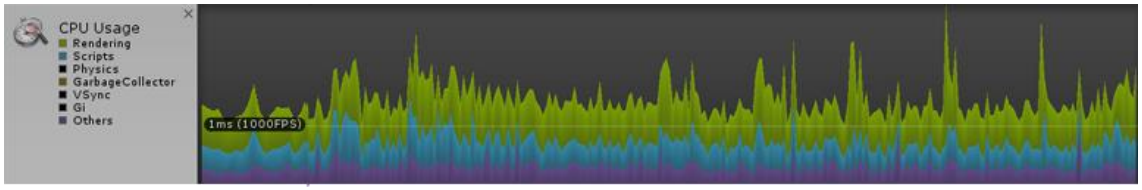
Por otro lado, hay que mejorar el tamaño de los decals de pintura que se imprimen. Actualmente no se tiene en cuenta el radio ni el isoLevel del fluido y sería conveniente hacerlo. Además, el tamaño de los decals es dependiente del tamaño y tiling de la textura base, cuando debería de ser únicamente dependiente del radio y del isoLevel. Esto es aceptable en el estado actual del desarrollo, pero no en el plugin final.

Respecto al rendimiento, se han realizado dos análisis. En ambos se ha comprobado la tasa de fps resultante de tener todas las combinaciones posibles con dos emisores predefinidos (esto es, rendimiento con ningún emisor, uno a uno, y los dos en conjunto). Los emisores se han definido pensando en la mínima calidad visual aceptable para el sistema, en una escena con una única luz direccional renderizando en tiempo real. Sus configuraciones son las siguientes:

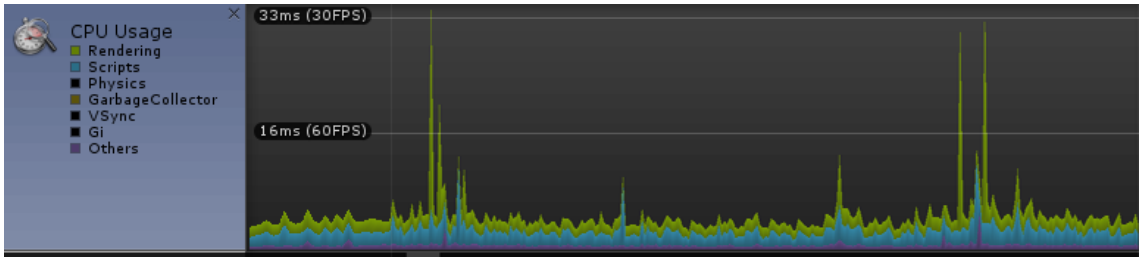
	Emisor Rojo	Emisor Azul
Tipo de emisor	PASSIVE	CONTINUOUS_CLICK
Apariencia visual	LIGHTING	LIGHTING
Tipo de blob	PAINT	PAINT
Número de blobs por disparo	2	1
Forzar inserción	No	Sí
Decals de pintura	3	1
Fuerza de impulso	10	50
Retraso intermedio	0.15	0.016
Gravedad	Sí	Sí
Factor de dispersión	1	0
Tiempo de vida (s)	2	0
Spawn activo	Sí	Sí
Máximos blobs permitidos	8	8
Radio	0.5	0.5
Resolución	10	14
IsoLevel	1	0.5

- **Análisis 1**: Máquina de bajas prestaciones (i53317U@1.7GHz, Nvidia GT 620M, 12 GB de RAM), resolución: 1366x768.

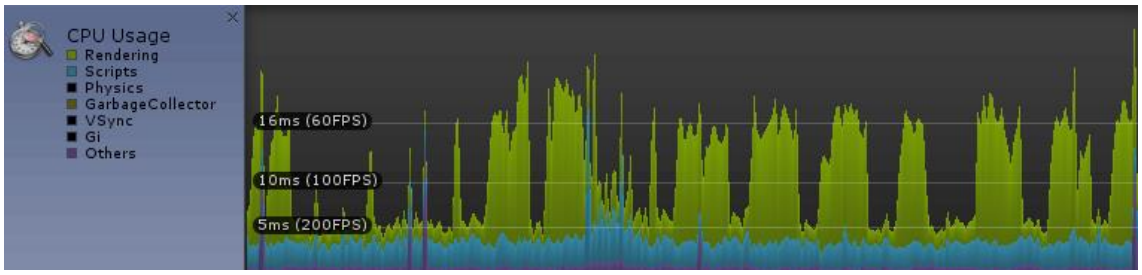
- Nada:



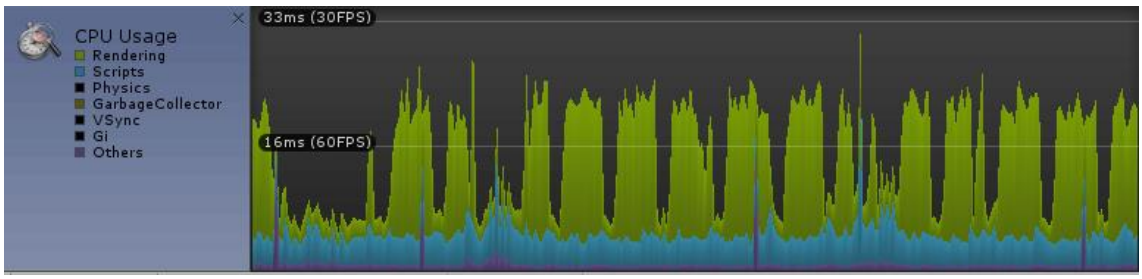
- Emisor Rojo:



- Emisor Azul:



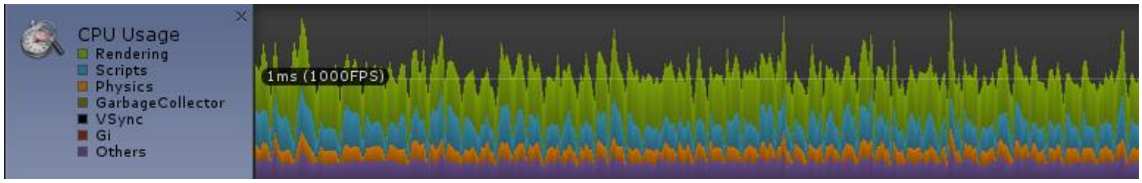
- Dos Emisores:



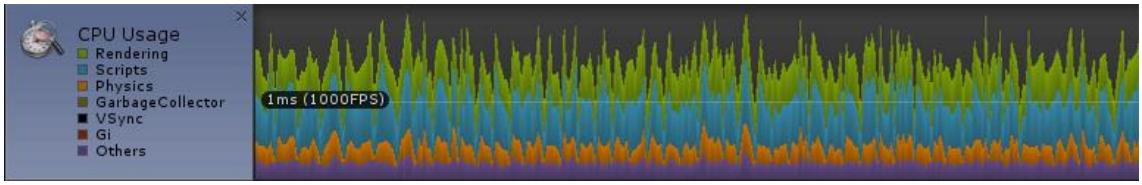
En resumen, aunque el aspecto visual no sea excelente y existan algunos picos de pocos fps, es lo bastante bueno para una máquina de estas características, y el sistema se encuentra dentro de las restricciones de eficiencia definidas, llegando incluso a estar cercano a los 60fps con un único emisor y estando sobre los 40 fps con dos emisores, dando la posibilidad de apurar la resolución para conseguir un mejor aspecto y seguir estando por encima de los 30fps.

- **Análisis 2**: Máquina de altas prestaciones (i7-3770@3.4GHz, Nvidia GTX 970, 32 GB de RAM), resolución: 1920x1080.

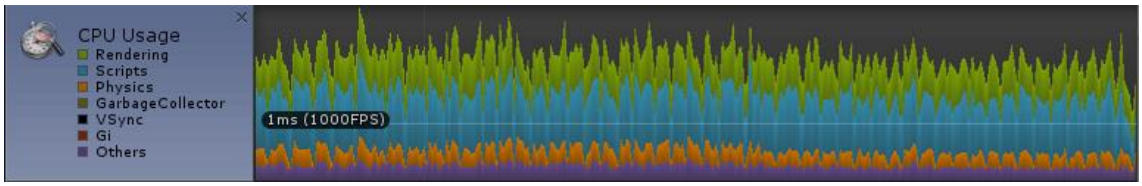
  - Nada:



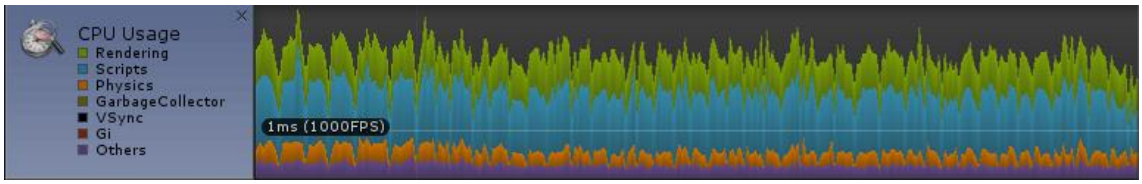
  - Emisor Rojo:



  - Emisor Azul:



  - Dos Emisores:



Como se ve, en este caso se supera con creces la especificación de la tasa de 60 fps, permitiendo aumentar la resolución de los fluidos y el número máximo de blobs permitidos para tener una mejora notable en la calidad visual. Hay que mencionar que la tarjeta gráfica utilizada es de gama media-alta y el sistema no conseguirá esta alta tasa de fps en gráficas de gama media, como por ejemplo la GTX 560. Sin embargo, realizado el análisis anterior con una tarjeta gráfica bastante inferior se puede estar seguro que superará los 30fps.

Este pequeño análisis demuestra que el requisito de eficiencia se ha cumplido y será tarea de los usuarios del plugin tunear sus parámetros para conseguir el equilibrio deseado entre la tasa de fps y la calidad del aspecto del fluido.

No obstante, se ha detectado en el desarrollo del plugin que hay algunas tarjetas gráficas, como algunas Intel integradas en el procesador, que no son capaces de

ejecutar el shader correctamente por algún motivo. En unos casos este problema se ha solucionado instalando los drivers más actualizados, pero en otros el problema persiste. Además, tanto en la tarjeta GTX 970 en la tarjeta gráfica GTX 760 la ejecución del shader causaba una excepción en los drivers, que se ha corregido realizando un pequeño cambio en la implementación del shader.

6.1 Planificación prevista vs real

La planificación pesimista inicial ha demostrado ser una muy buena aproximación del desarrollo del proyecto hasta el punto de no haber necesitado cuanto apenas el colchón de contingencias y poder utilizar todos esos días para mejorar el prototipo y redactar esta memoria. En la etapa de aprendizaje se han dedicado menos días de lo inicialmente planificado debido al trabajo desempeñado en las prácticas de empresa.

Aunque inicialmente en la planificación se asumía el trabajo constante, día a día, sin parones debido a las vacaciones de fallas ni pascuas, finalmente, esto no ha sido así y en algunos puntos del proyecto se ha sufrido algún retraso.

	Tarea	Inicio	Fin	Dur
	GameBrush	22/2/16	1/7/16	95
1	Documentación y aprendizaje	22/2/16	16/3/16	18
2	1ª Iteración	25/3/16	7/4/16	10
3	2ª Iteración	8/4/16	25/4/16	12
4	3ª Iteración	27/4/16	5/5/16	7
5	4ª Iteración	15/5/16	2/6/16	14
6	COLCHÓN DE CONTINGENCIAS	5/6/16	1/7/16	20

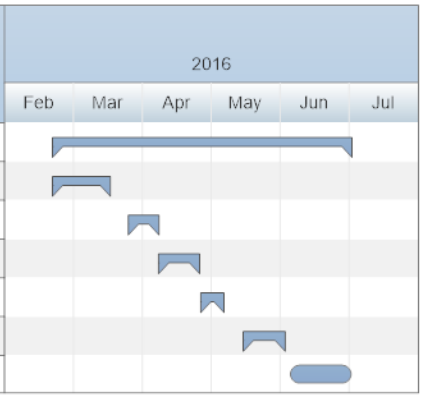


Ilustración 40: Tiempo real dedicado al proyecto

En fallas realicé un viaje que retrasó el inicio de la primera iteración por 7 días. En pascuas me tomé un descanso de 7 días, pero la planificación en este punto era demasiado pesimista y realicé todas las tareas 5 días antes.

Además, participé en una competición que retrasó 10 días el inicio de la 4ª iteración. Además, el periodo de depuración ha costado dos días más de lo inicialmente planificado.

Con todos estos problemas se han empleado un total de 10 días del colchón de contingencias, quedando aún 20 días que se han empleado para realizar pequeñas mejoras en el prototipo.

6.2 Relación con los estudios cursados

Para la realización de este proyecto han sido necesarios conocimientos de programación y diseño de software, de implementación gráfica, por no mencionar las herramientas de gestión de tiempo y trabajo. Asignaturas que me han ayudado directamente en el desarrollo de este proyecto han sido:

- **Matemática discreta, análisis matemático, programación, estructuras de datos y algoritmia**. Desde las bases de la escritura de código hasta técnicas de optimización avanzadas. Han ayudado enormemente a mantener el sistema eficiente.

- **Ingeniería de software**. Si bien mi especialización ha sido enfocada a la algoritmia, todo lo aprendido en esta asignatura me ha servido de base para documentarme por mi cuenta sobre aspectos de diseño de software orientado a objetos y a aprender sobre cuándo aplicar las optimizaciones aprendidas y cuándo no es necesario.

- **Estructura y arquitectura de computadores, computación paralela**. Me han ayudado enormemente a entender el pipeline y el modelo de paralelismo existente en las GPUs y, por lo tanto, ha agilizado el aprendizaje de los shaders.

- **Introducción a los sistemas gráficos interactivos y álgebra lineal**. Gracias a estas asignaturas he sido capaz de comprender fácilmente las transformaciones que se tienen que realizar en los shaders para conseguir el efecto deseado

- **Gestión de proyectos**. Me ha dado las herramientas para poder desarrollar mi trabajo de forma ordenada y pautada, descubriendo de forma temprana el origen de los posibles retrasos del proyecto y dándome la oportunidad de paliarlos a tiempo.

7. Trabajos futuros

Como mejoras necesarias, se tiene que corregir el problema del shader para gráficas Intel integradas. En este prototipo no es vital que funcione en esas tarjetas mientras funcione en el resto, pero en la versión final del plugin esto tiene que corregirse. Además, el sistema necesita una mejor gestión de los volúmenes de renderizado, esto es, la nube de tetraedros que computa la CPU y envía a la GPU para la generación de la malla del fluido. Actualmente se genera la nube centrada en cada blob. El problema de esto es que, si hay n blobs muy cercanos unos de otros, se calculan los vértices del espacio de los volúmenes intersectados n veces. Esto no solo supone un problema de rendimiento, siendo en algunos casos posible la eliminación de la mitad de los volúmenes de renderizado, sino que además se genera un pequeño artefacto gráfico en su intersección cuando la resolución del fluido no es muy alta.

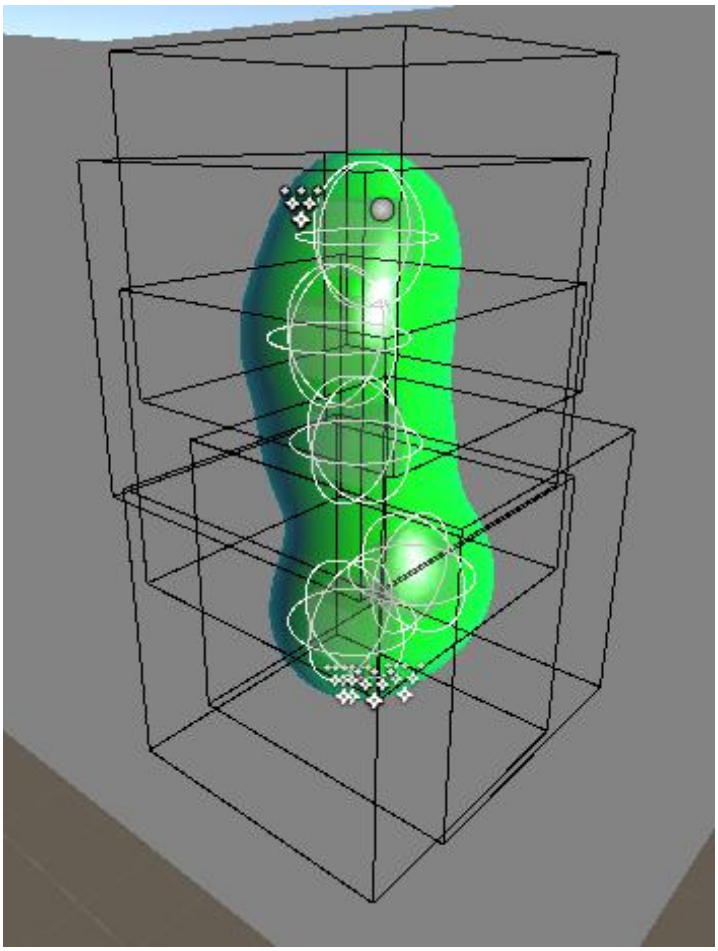


Ilustración 41: Varios volúmenes de renderizado intersectando

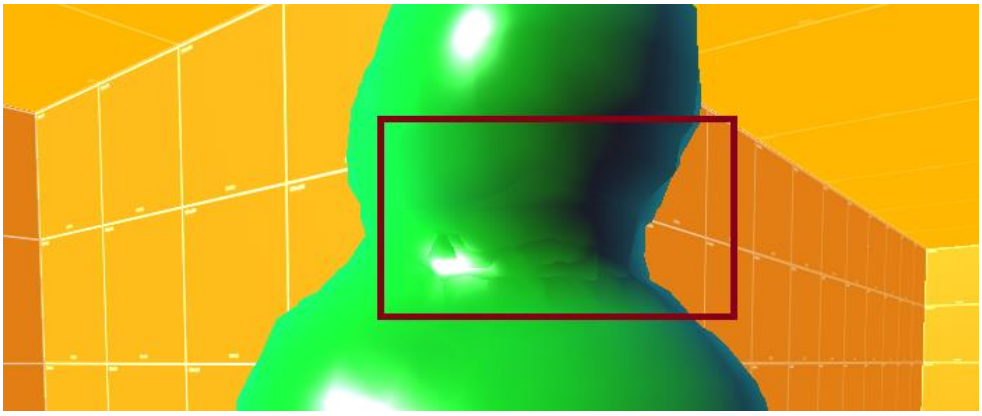


Ilustración 42: Artefacto gráfico debido a la intersección de volúmenes de renderizado

Además, en ocasiones el volumen no se centra correctamente, dejando parte del fluido fuera de él y, por lo tanto, mostrando recortes indeseables en el fluido.

Aparte de esto, va a ser necesario dedicar tiempo al testeo del shader en distintas tarjetas gráficas ya que actualmente en la programación gráfica hay cierto vínculo entre el código y el hardware sobre el que se ejecuta, por lo que resulta necesario ejecutar el shader en las gráficas más usadas a día de hoy para depurarlo y estar seguros de que funciona correctamente en todas las máquinas.

Otras mejoras necesarias a realizar, son:

- Conseguir que el plugin funcione en máquinas MAC y Linux.
- Incluir un array de posibles sonidos a emitir en el momento de una colisión del fluido.
- Conseguir que el tamaño de la pintura sea independiente de la textura base utilizada en la superficie.

Como mejoras deseables a realizar, serían interesantes:

- Cambiar el algoritmo MT6 por RMT para evitar la sobreteselación.
- Aplicar el algoritmo subdivisión Catmull-Clark [12] para suavizar la malla sin tener que aumentar la resolución del volumen de renderizado, que sigue un coste cúbico.
- Poder texturizar la superficie del fluido.

8. Agradecimientos

A Jordi de Paco y Marina González, por darme tantas oportunidades y ser tan majetes.

A Ramón Mollá, por la libertad que me ha dado en el desarrollo de este proyecto y su ayuda en todos los aspectos.

A todos aquellos a los que he dicho que hacer este proyecto era un tedio.
Era mentira.

9. Bibliografía

[1] K. Fatahalian y M. Houston, «A closer look at GPUs,» *Communications of the ACM,* vol. 51, nº 10, pp. 50-57, 2008.

[2] M. Macklin, M. Muller, N. Chentanez y T.-Y. Kim, «Unified Particle Physics for Real-Time Applications,» de *SIGGRAPH*, Vancouver, 2014.

[3] K. v. Kooten, G. v. d. Bergen y A. Telea, «Chapter 7: Point-Based Visualization of Metaballs on a GPU,» de *GPU Gems 3*, Massachusetts, Addison-Wesley Professional, 2007.

[4] C. D. Hansen y C. R. Johnson, The Visualization Handbook, Ámsterdam: ELSEVIER, 2005.

[5] R. C. Martin, Agile Software Development, Principles, Patterns, and Practices, Nueva York: Pearson Education, 2002.

[6] M. Fowler, Refactoring. Improving the Design of Existing Code, Massachusetts: Addison-Wesley, 1999.

[7] R. C. Martin, «The Open-Closed Principle,» de *C++ GEMS*, SIGS Publications Group, 2000, pp. 97-112.

[8] Y. Uralsky, «Practical Metaballs and Implicit Surfaces,» de *Game Developers Conference*, San Jose, California, 2006.

[9] J. Patera, «Iso-Surface Extraction and Approximation Error,» Department of Computer Science and Engineering, University of West Bohemia in Pilsen, 2004.

[10] A. Lauritzen, «Deferred Rendering for Current and Future Rendering Pipelines,» de *SIGGRAPH*, Los Angeles, 2010.

[11] G. Treece, R. Prager y A. Gee, «Regularised marching tetrahedra: improved iso-surface extraction,» University Engineering Department, Cambridge, 1998.

[12] E. Catmull y J. Clark, «Recursively generated B-spline surfaces on arbitrary topological meshes,» *Computer-Aided Design,* vol. 10, nº 6, pp. 350-355, 1978.

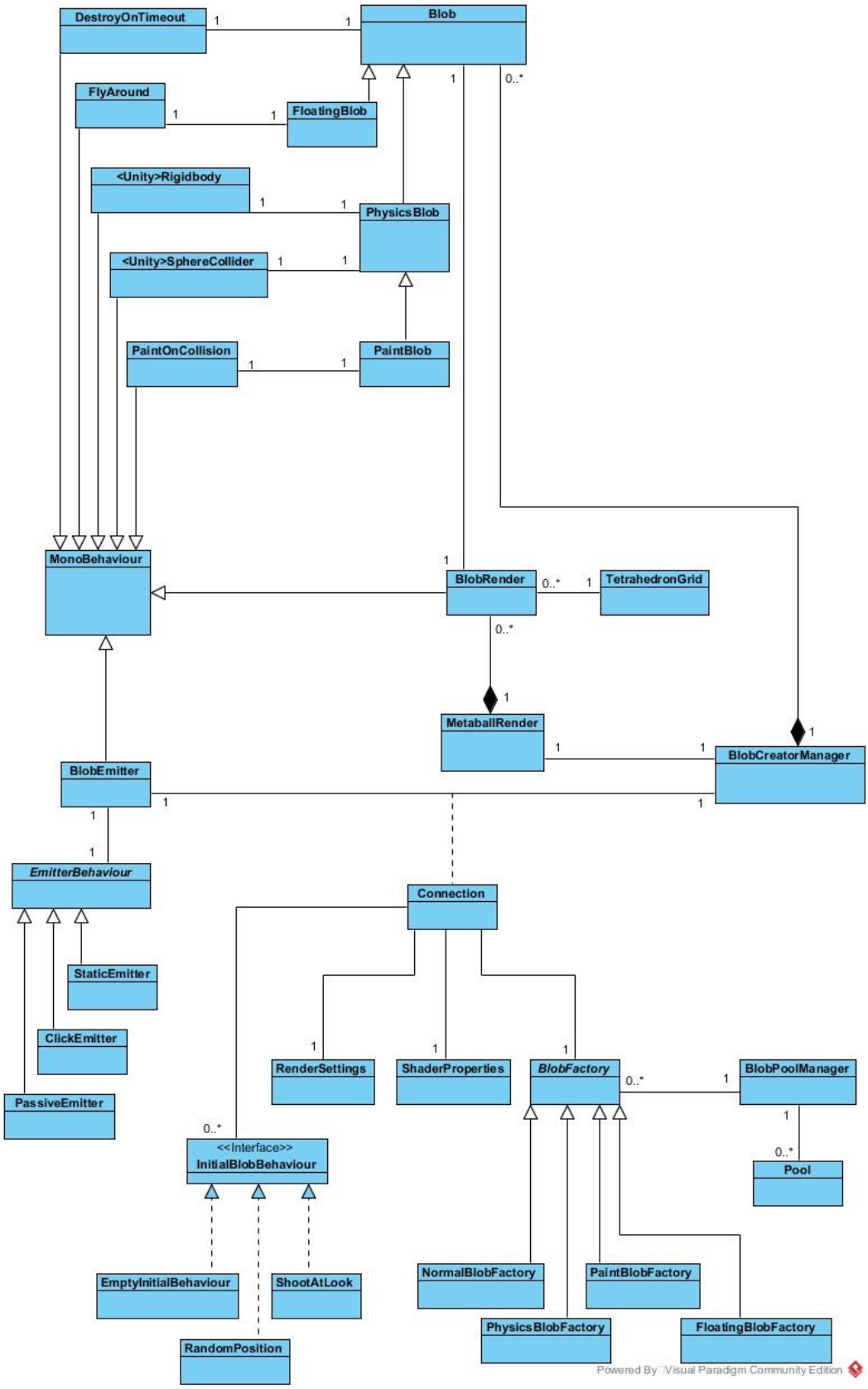
10. ANEXO

10.1 Glosario de términos

Bitmap	Mapeado de un dominio concreto a bits.
Blending	Operación de mezcla de colores (aditivo, multiplicativo, etc.)
Cg	Lenguaje de Nvidia para escribir shaders multiplataforma con la misma sintaxis que HLSL.
Component	Script de Unity3D que hereda de MonoBehaviour (o ScriptableObject), dándole de esta forma acceso al game loop.
Direct3D	API gráfico de DirectX.
DirectX	Colección de APIs multimedia para sistemas Windows.
FPS	Marcos por segundo, del inglés *frames per second*.
Game loop	Bucle principal de cualquier videojuego, encargado de recoger las entradas y procesar los resultados de todos los sistemas, mostrando por pantalla el proceso de rasterizado.
GameObject	Estructura en Unity que almacena *Components*
HLSL	Lenguaje de escritura de shaders empleado en Direct3D.
HUD	Del inglés, *Heads-Up Display*, hace referencia a la información del sistema de juego que se encuentra en la pantalla.
Inspector	Sección de la interfaz de Unity dedicada a la configuración de los parámetros de los scripts vinculados a un GameObject.
OpenGL	API gráfico multiplataforma.
Pipeline	Proceso segmentado en etapas conectadas en las que la salida de cada etapa es la entrada de la siguiente.
Raster	Proceso de transformación de una imagen vectorial en un bitmap de píxeles, típicamente para dibujar en la pantalla.
Render	Proceso de cálculo vectorial sobre la geometría de los elementos que se van a dibujar.
RenderTexture	Textura especial empleada en Unity3D utilizada para imprimir sobre ella lo que se visualiza en una cámara dada.
Shader	Pequeño programa que se ejecuta en la GPU explotando el paralelismo SIMD para obtener efectos gráficos.

Shaderlab	Lenguaje propio de Unity3D para definir shaders.
SIMD	Tipo de paralelismo *Single Instruction Multiple Data* (una instrucción, varios datos)
Teselado	Proceso por el cual una figura geométrica se regulariza utilizando polígonos sencillos, típicamente triángulos.
Tiling	Se trata de un parámetro de dos dimensiones que identifica cómo se repite una textura sobre una superficie (tanto en la dirección x como en la y).

10.2 Diagrama de clases completo:



10.3 Sistema de pintura descartado

Se ha acabado con un shader para renderizar las superficies que utiliza tres texturas diferentes para poder realizar el pintado:

1. La textura base, la imagen de la superficie.
2. La RenderTexture que renderiza los quads vivos, guardando toda la información de la pintura de la superficie.
3. La textura de reserva del color que generaban los quads antes de ser descartados.

Cuando se cambie de superficie impactada, se realiza un guardado de los quads destinados a pintar la anterior superficie en su textura de reserva y a la cámara se le configura la nueva RenderTexture asociada a la nueva superficie. De esta forma, se pueden pintar tantas superficies como existan.

Como era de esperar, esta solución no es perfecta. Tiene un problema grave de eficiencia en el momento de trasladar el contenido de la RenderTexture a la textura de reserva. Para mantener la escala de las manchas de pintura generadas, la RenderTexture tendrá el mismo tamaño que la superficie sobre la que se aplica. Esto se consigue realizando un pequeño cálculo del tamaño de la textura base y el tiling que tiene definido. El problema viene cuando la RenderTexture tiene un tamaño superior a 128x128 píxeles y se realizan cambios de superficie constantemente. En este contexto, sigue existiendo un cuello de botella. Se puede construir un pequeño conjunto de configuraciones de cámara y lienzo, y realizar una gestión sobre ellas, pero esto no soluciona el problema, únicamente lo hace menos grave si hay menos superficies impactadas que configuraciones cámara y lienzo. También se puede reducir el área de textura que se tiene que guardar en la textura de reserva realizando un seguimiento del subárea de la imagen que contiene a todos los quads, pero aun así el peor caso sigue siendo inviable.

Fijándonos en la naturaleza del cómputo, se ve que para realizarlo únicamente se necesitan tres valores para obtener el valor del nuevo color que se guardará en la textura de almacenaje. Para cada pixel de la nueva textura de reserva se lee un pixel de la textura con los colores de la RenderTexture, se lee otro con el valor actual de la textura de reserva y se hace una pequeña operación de blending entre ambos colores. Esto se repite así tantas veces como píxeles tenga la textura reserva, que coincide con las dimensiones de la RenderTexture. Como se puede apreciar, este proceso es

fácilmente paralelizable con un paralelismo SIMD, que es justamente el tipo de paralelismo existente en una GPU.

En el proceso, la CPU enviaría a la GPU las tres texturas y esta, al terminar, devolvería la textura con los nuevos colores que la CPU asignaría a su lugar.

Con la CPU el proceso es pixel tras pixel, mientras que con la GPU el proceso se realiza a la vez, tardando en procesar un número arbitrario de píxeles en, idealmente, el mismo tiempo que tarda en procesar uno solo. Sin embargo, para poder realizar este cálculo en la GPU es necesario el uso de los Compute Shaders.

10.3.1 Compute Shader

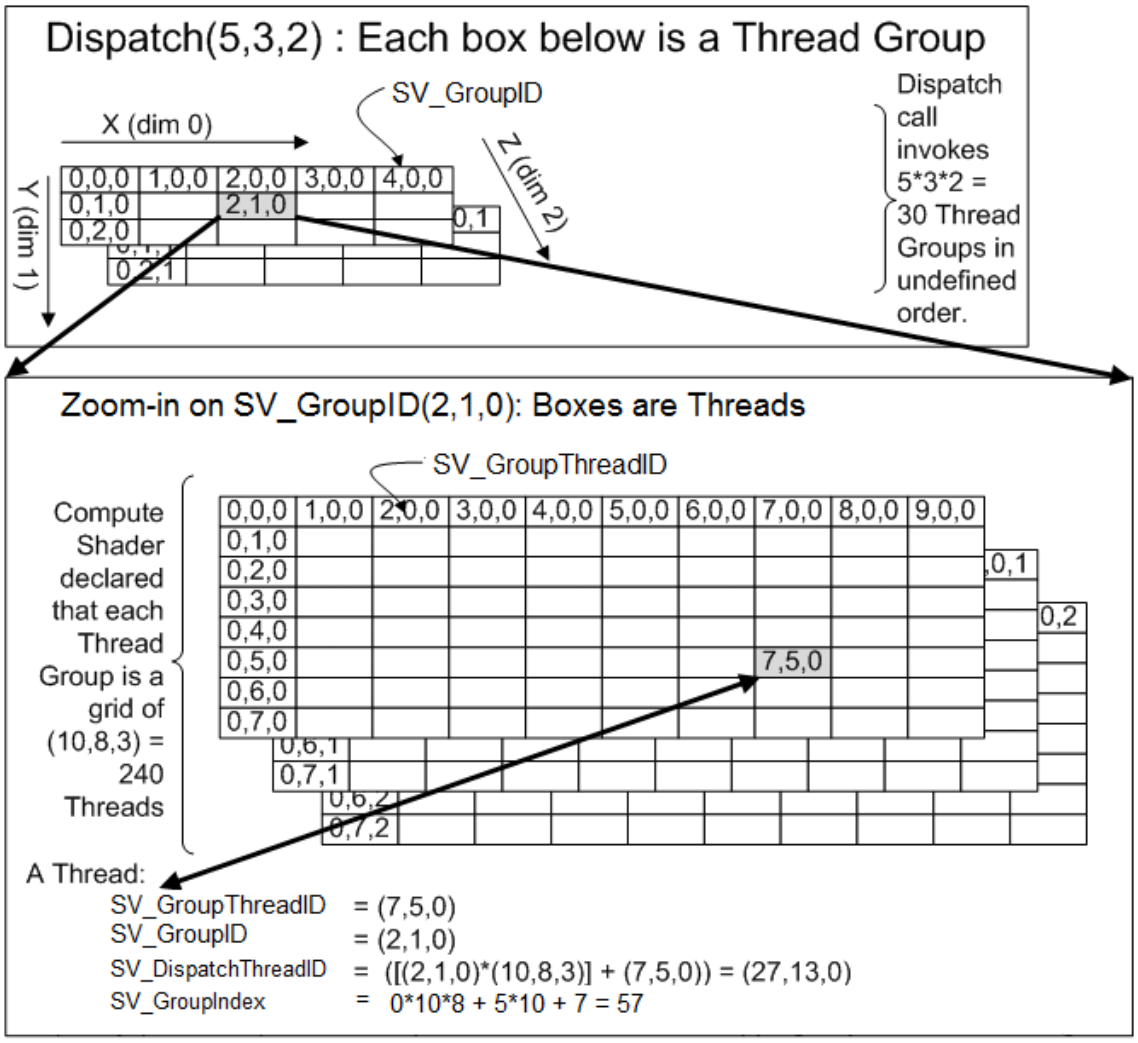
Un Compute Shader es un tipo de shader especial que no interviene en los procesos gráficos. Se usa para realizar cálculos sobre información arbitraria, de cualquier tipo. Aunque puede utilizarse directamente para el proceso de renderizado, los Compute Shader típicamente se usan para realizar procesos relacionados con este proceso de forma indirecta.

Mientras que los otros tipos de shaders tienen definidas unas entradas y salidas, así como una frecuencia de ejecución fijada por el estado del pipeline gráfico que ocupa, un Compute Shader no. Está dirigido completamente por la CPU y se utilizan principalmente para aprovechar la capacidad de computo en paralelo de la GPU para implementar tareas paralelizables. Este tipo de shaders está disponible a partir de las implementaciones de Direct3D 11 y de OpenGL 4.3.

Para arrancar un Compute Shader es necesario realizar una llamada al método Dispatch:

```
void Dispatch(
  [in] UINT ThreadGroupCountX,
  [in] UINT ThreadGroupCountY,
  [in] UINT ThreadGroupCountZ
);
```

Cada uno de los argumentos señala el número de grupos de hilos que se desean lanzar para ejecutar el shader. Esta llamada internamente genera una serie de identificadores que puede emplear el shader para identificar qué hilo o conjunto de hilos está en ejecución:



En cada ComputeShader es necesario definir en su cabecera cuántos hilos representa un grupo. En nuestro caso, la orden Dispatch se lanza con los argumentos `(renderTexture.ancho/8, renderTexture.alto/8, 1)`.

Y en el Compute Shader se define el número de hilos que se crean por grupo con la expresión `[numthreads(8,8,1)]`. De esta manera se está definiendo que cada grupo contiene un total de 8x8x1 hilos. Nótese que la llamada a Dispatch se ejecuta con los argumentos ancho/8 x alto/8 x 1. El total de hilos lanzados a la GPU será de:

```
8 x 8 x ancho/8 x alto/8 = ancho x alto
```

Como se ve, se lanzan tantos hilos como pixeles existen en la imagen. Por lo que, si hay tantos hilos como píxeles disponibles, el proceso de cálculo será muy rápido y el coste vendrá dado por la comunicación entre CPU y GPU.

10.4 El equipo: Deconstructeam

Deconstructeam es un pequeño equipo de desarrollo de videojuegos independientes. Cuenta con más de 16 juegos creados en competiciones que consisten en desarrollar un videojuego en unos pocos días. Uno de ellos, Gods Will Be Watching, obtuvo el 2º puesto y acabó obteniendo el apoyo de la editora Devolver Digital para la financiación y posterior distribución en la plataforma de distribución digital Steam, entre otras, de una versión mucho más grande y ambiciosa. Gods Will Be Watching ha recibido numerosos premios, entre ellos al mejor videojuego independiente, por Fun&Serious, y al mejor debut, por Gamelab.

El objetivo de Deconstructeam es buscar nuevas formas de crear experiencias narrativas interactivas, tratando de crear videojuegos que vayan más allá de la diversión y el mero entretenimiento. Es por este motivo por el que se ha desarrollado la tecnología mostrada en el presente texto.

Personalmente he realizado las prácticas de empresa a través del SIE en la empresa de este equipo, conocida como Gods Will Be Watching S.C. (código de convenio RR31551), durante los meses comprendidos entre febrero y junio, ambos inclusive. Este trabajo final de grado surge como resultado de esta colaboración.

deconstructeam.com