Universidad Zaragoza

Proyecto Fin de Carrera

SIMULACIÓN DE SISTEMAS MEDIANTE PARTÍCULAS UTILIZANDO GPU's

Autor

Adrián Pascual Sancho

Director/es y/o ponente

Francisco José Serón Arbeloa
Eduardo Mena Nieto

Departamento de Informática e Ingeniería de Sistemas
Área de Lenguajes y Sistemas Informáticos
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

2015

SIMULACIÓN DE SISTEMAS MEDIANTE PARTÍCULAS UTILIZANDO GPU's

RESUMEN

El problema gravitatorio de los N cuerpos es, tal vez, el problema no resuelto más antiguo y a la vez más fecundo en la historia de la ciencia. Su origen se remonta a la necesidad del hombre antiguo de medir el paso del tiempo para anticipar migraciones de animales y, posteriormente, los ciclos agrícolas.

En este proyecto se ha creado un programa que implementa uno de los métodos de resolución que existen para el problema de los N cuerpos, concretamente, el método Partícula-Partícula. Se ha abordado prácticamente todo lo que OpenGL, librería base del proyecto, permite hacer sobre la tarjeta gráfica, desde el computo de los cálculos que mueven cada planeta hasta la renderización de los mismos en un mundo 3D.

El objetivo de este proyecto consiste en estudiar el comportamiento de una GPU trabajando con partículas. Se ha tomado como escenario los planetas y su gravitación, y se quería averiguar cuál era el número límite de planetas que se podían simular a la vez manteniendo unos FPS aceptables y utilizando todo el potencial de la tarjeta gráfica.

En las páginas de este documento se recoge todo el proceso de preparación, desarrollo, generación de resultados y pruebas de rendimiento. Como se podrá comprobar, los resultados obtenidos se enmarcan dentro de lo que se esperaba teóricamente, por lo que el objetivo ha sido cumplido.

TABLA DE CONTENIDOS

1. INTRODUCCIÓN

1.1. OBJETIVOS

Este documento recoge la memoria del Proyecto Fin de Carrera de Adrián Pascual Sancho. Este PFC se enmarca dentro del mundo de las tarjetas gráficas y gráficos 3D.

El objetivo principal del presente proyecto es el estudio del comportamiento de una GPU trabajando con una carga gráfica y computacional elevada. Para ello, se ha tomado como escenario el problema gravitacional de los N cuerpos.

El problema gravitatorio de los N cuerpos es, tal vez, el problema no resuelto más antiguo y a la vez más fecundo en la historia de la ciencia. Su origen se remonta a la necesidad del hombre antiguo de medir el paso del tiempo para anticipar migraciones de animales y, posteriormente, los ciclos agrícolas.

En este proyecto, se va a implementar un algoritmo básico de resolución de este problema. Se va a desarrollar una simulación de la gravitación universal, realizando todos los cálculos dentro de la GPU, y con la renderización, aunque básica, de los planetas involucrados. Estos puntos son independientes del modelo de GPU utilizado.

Cómo se podrá comprobar a lo largo del trabajo, el algoritmo implementado presenta una complejidad computacional elevada, del orden de complejidad cuadrático ($O(n^2)$). Para evitar que el tiempo de ejecución se incremente demasiado, se van a aplicar ciertas optimizaciones, tanto a las fórmulas con las que se actualizan los datos, como al número de partículas que tiene que tratar en cada iteración.

Se pretende partir del problema de los dos cuerpos y llegar hasta un sistema con un número N de partículas, a partir de la cual, la GPU seleccionada no consiga alcanzar unos FPS aceptables, entendiendo como aceptables, alrededor de 15 FPS. Este número N es depende del modelo de GPU utilizado. Para este proyecto, se va a utilizar la Nvidia GeForce GTX 660M, por lo que todos los resultados serán específicos de esta tarjeta gráfica.

La metodología a seguir consistirá en:

- Analizar los algoritmos existentes para la resolución del problema de los N cuerpos.

- Realizar una implementación en un entorno NVIDIA y analizar sus prestaciones y resultados obtenidos.

- Integrar todo lo anterior en un programa en 3D escrito en C++ y utilizando la librería OpenGL 4.x.

- Obtener un límite máximo de partículas, a partir del cual, la GPU no consiga obtener unos FPS mínimos para la correcta visualización del programa.

1.2. DESCRIPCIÓN DE LA ESTRUCTURA DE LA MEMORIA

- Métodos de Resolución del problema gravitacional de los N cuerpos (sección 2):

Clasificación y análisis de las diferentes aproximaciones existentes para la resolución del problema gravitacional de los N cuerpos y desarrollo del algoritmo principal de la simulación.

- La GPU (sección 3):

Presentación del funcionamiento de las tarjetas gráficas actuales, así como, de la librería OpenGL, encargada de los cálculos y la generación de la escena 3D.

- Preparación de la simulación (sección 4):

Explicación de todas las decisiones tomadas en relación al diseño e implementación del algoritmo utilizado en la simulación.

- Simulación del Sistema Solar (sección 5):

Recorrido por las diferentes versiones del algoritmo implementado, concluyendo con la simulación del Sistema Solar completo en la GPU. Presentación y explicación de los resultados obtenidos.

- Simulación aleatoria (sección 6):

Explicación de los cambios realizados sobre el algoritmo de la sección anterior, así como de los resultados obtenidos.

- Conclusiones (sección 7):

Valoración del trabajo realizado, presentación del diagrama temporal y opinión personal.

2. MÉTODO DE RESOLUCIÓN PARTÍCULA-PARTÍCULA

Para simplificar la simulación, se considera una estrella como si fuera un punto de masa cuyas únicas propiedades a tener en cuenta son la masa y la atracción gravitacional. Teniendo esto en mente, se puede llamar al problema clásico gravitacional de N cuerpos como el problema de determinar el comportamiento de una colección de N puntos de masa cuando están en movimiento bajo fuerzas gravitacionales recíprocas de acuerdo con las leyes de Newton del movimiento.

La evolución de los grupos de estrellas o galaxias, considerados como puntos de masas, y el desarrollo de estructuras de espiral y barril en una galaxia, son problemas que pueden aproximarse con la simplificación anterior.

En este capítulo, se van a presentan los tipos de aproximaciones al problema que existen, así como los métodos matemáticos de resolución de cada uno de ellos, finalizando el apartado explicando, más en detalle, el método de resolución "Partícula – Partícula", que va a ser el que se va a utilizar para las simulaciones de este proyecto.

2.1. CLASIFICACIÓN DE LAS APROXIMACIONES

El problema de los N cuerpos se puede dividir en 3 clases de aproximaciones, dependiendo de la importancia que tengan las colisiones binarias en la evolución del sistema. Estas aproximaciones se denominan: *collisional-dominated systems*, *collisionaless systems* y cúmulo globular.

Si se define que la ratio de colisión binaria (v_D) (capítulo 11 de [1]) mide el alcance en el que la órbita de una estrella situada en un campo gravitacional aparece perturbada por la presencia de otra estrella cercana, entonces el tiempo de colisión binario ($T_D = v_D^{-1}$) en un sistema de puntos de masas en 3 dimensiones viene definido por:

$$T_D = \frac{v^3}{8\pi n G^2 m^2 H \ln\left(\frac{Dv^2}{2Gm}\right)} \tag{2,1}$$

n: $densidad\ volumétrica$
m: $masa$
v $velocidad\ relativa$
D: $distancia$
G: $constante\ de\ gravitacion\ universal$
H: $constante\ que\ vale\ 0.4\ aproximadamente$

Sabiendo que n es proporcional a N^{-1} y que m es proporcional N, se puede reducir la anterior ecuación a $T_D \propto N$, en consecuencia $v_D = T_D^{-1} \propto N^{-1}$.

En grupos compuestos por pocos centenares de estrellas, la órbita de cualquiera de ellas dependerá principalmente de la posición y masa precisas de las estrellas vecinas locales. Este tipo de sistemas son llamados *collisional-dominated systems* (sistemas dominados por las colisiones) debido a estas interacciones binarias.

Por otra parte, si el sistema está compuesto por un número N muy grande de estrellas, como podría ser una galaxia, que cuenta con un N del orden de 10^{11}, el tiempo para que una órbita estelar sea perturbada un ángulo de 9 grados en alguna dirección es del orden de 100 rotaciones de la galaxia. Esto representa un límite de tiempo en torno al cual los efectos de las colisiones pueden considerarse como insignificantes. Por este motivo, este tipo de sistemas se denominan *collisionaless systems* (sistemas sin colisiones). La evolución de estos sistemas está determinada por la densidad de masa de todo el sistema y no por las masas individuales de cada estrella.

Existe una tercera agrupación de estrellas, denominada cúmulo globular, que está compuesto por un número de estrellas comprendido entre 10^4 y 10^6 estrellas. Este es el grupo más complicado de simular ya que N no es lo suficientemente grande como para ignorar las colisiones ni tan pequeño como para calcular la órbita de cada estrella de forma precisa.

2.2.MÉTODOS DE RESOLUCIÓN

Las técnicas de simulación de partículas intentan modelar sistemas de muchos cuerpos mediante la resolución de las ecuaciones de movimiento de un conjunto de partículas usadas para representar el sistema. Para cada uno de los tipos de aproximaciones citados se puede definir un método de resolución, como podemos observar en el capítulo 1 de [1]. Para agrupaciones pequeñas de estrellas, el método "Partícula-Partícula" (PP) es el más adecuado, para agrupaciones intermedias se usa el método "Partícula-Partícula – Partícula-Malla" (P^3M o PPPM) y para agrupaciones grandes se emplea el método "Partícula-Malla" (PM).

La técnica PP para simular los *collisional-dominated systems* se ha utilizado muchos desde los años 60. La fuerza en una estrella se calcula mediante la suma de las interacciones con las otras estrellas. Para obtener la fuerza en las N estrellas, son necesarias entorno a $5N^2$ operaciones. Hablando de agrupaciones menores a 4000 cuerpos, este método presenta importantes ventajas respecto a los demás ya que cada fuerza es tan precisa como la precisión aritmética del ordenador utilizado.

El método PM, utilizado en la simulación de los *collisionaless systems*, fue introducido a finales de los años 60 usando una modificación de un programa diseñado para la simulación de gas plasma. Esta técnica trata a la fuerza como un campo cuya magnitud se aproxima utilizando una malla, por tanto, no trabaja con la masa de cada partícula en concreto sino con una densidad de masa calculada a partir de varias partículas cercanas. Es por este motivo que la principal característica de este método es su velocidad. Con estas densidades, se calculan los valores de la malla, los cuales permiten, mediante su interpolación, obtener las fuerzas y potencias en cada partícula. Debido al uso de las densidades, el método PM es inaceptable para el estudio de interacciones entre partículas cercanas.

Por último, el método P^3M sirve para paliar el principal fallo de la técnica PM, el cálculo de fuerzas entre partículas cercanas. La fuerza se divide en dos partes, una de variación rápida y de corto alcance, que se calcula mediante la suma directa de fuerzas usando el método PP, y otra de variación lenta y de largo alcance, cuyo valor se obtiene usando el método PM. El principal problema de un algoritmo P^3M es que suele ocurrir que la parte del sumatorio directo termine dominando sobre la otra, consiguiendo lo contrario de lo que se buscaba.

2.3.ALGORITMO BASE DEL MÉTODO PP

Como ya se ha comentado anteriormente, para el proyecto planteado, se ha elegido el método "Partícula-Partícula", que a pesar de ser el más sencillo, tanto conceptual como computacionalmente, es el que más precisión obtiene en un ámbito de pocos cuerpos como es el Sistema Solar.

En un algoritmo que implementa un método PP, el estado del sistema en un instante t está definido mediante la posición y la velocidad de cada partícula.

$$x_i(t), v_i(t); i = 1..N_p \qquad (2,2)$$

N_p: número de partículas
x_i: *posición en el instante i*
v_i: *velocidad en el instante i*

Dicho algoritmo, que podemos encontrar en el capítulo 1 de [1], consta de en un bucle principal (ver imagen 2.1), que se denomina *timestep loop*, que actualiza los valores indicados utilizando las ecuaciones de movimiento (ecuación 2.2) y la fuerza de interacción entre partículas (ecuación 2.3).

Timestep loop of PP method

1. Compute forces.
   Clear force accumulators
   for $i = 1$ *to* N_p *do*
   $\mathbf{F}_i := 0$
   Accumulate forces
   for $i = 1$ *to* $N_p - 1$ *do*
   for $j = i+1$ *to* N_p *do*
   Find force $\mathbf{F}_{ij}$ of particle j on particle i
   $\mathbf{F}_i := \mathbf{F}_i + \mathbf{F}_{ij}$
   $\mathbf{F}_j := \mathbf{F}_j - \mathbf{F}_{ij}$
2. Integrate equations of motion.
   for $i = 1$ *to* N_p *do*
   $\mathbf{v}_i^{\text{new}} := \mathbf{v}_i^{\text{old}} + \frac{\mathbf{F}_i}{m_i} DT$
   $\mathbf{x}_i^{\text{new}} := \mathbf{x}_i^{\text{old}} + \mathbf{v}_i DT$
3. Update time counter.
   $t := t + DT$

Imagen 2.2.1 Algoritmo PP básico.

En este proyecto, se va a usar este algoritmo para calcular la evolución de los astros del Sistema Solar, y posteriormente la evolución de los planetas colocados de forma aleatoria en un espacio determinado, partiendo de una posición inicial elegida manualmente. Por tanto, la fuerza de interacción entre las partículas implicadas será la definida por la ley de gravitación universal, la cual predice que la fuerza ejercida entre dos cuerpos de masas m_1 y m_2 separados una distancia r es proporcional al producto de sus masas e inversamente proporcional al cuadrado de la distancia, es decir:

$$F = G\frac{m_1 m_2}{r^2} \tag{2,3}$$

F: $módulo\ de\ la\ fuerza$
G: $constante\ de\ gravitación\ universal = 6.6784 * 10^{-1}\ \ Nm^2Kg^{-2}$
m_i: $masa\ del\ planeta\ i$
r: $distancia\ entre\ los\ planetas$

Aplicando esta ley a nuestro caso de N partículas y simplificando la fórmula (consultar el capítulo 31 de [4]), la fuerza total F_i en el cuerpo i vendrá dada por la ecuación 2.3.

$$\overrightarrow{F_i} = Gm_i \sum_{1 \le j \le N} \frac{m_j \overrightarrow{r_{ij}}}{\left(\left\|r_{ij}\right\|^2 + \varepsilon^2\right)^{2/3}} \; ; \; \overrightarrow{r_{ij}} = x_j - x_i \tag{2,4}$$

ε^2: $factor\ de\ suavizado$
$\overrightarrow{r_{ij}}$: vector que va desde la partícula i a la partícula j
x_i: $posicion\ de\ la\ partícula\ i$
m_i: $masa\ de\ la\ partícula\ i$

2.4.OPTIMIZACIONES

Al bucle citado (*timestep loop* básico) se le van a añadir ciertos cambios de forma que la precisión en el cálculo de la posición y velocidad se vea mejorada y admita el poder ejecutarlo computacionalmente de forma paralela.

Para calcular la fuerza total en la partícula i, es necesario sumar todas las fuerzas que cada una de las demás partículas ejercen sobre ella. La aproximación más básica reside en implementar un bucle anidado para recorrer todas las partículas, y por cada una, recorrer todas las demás, calculando así la fuerza total (imagen 2.2), lo cual genera un algoritmo de orden de complejidad[1] O (n^2).

[1] El orden de complejidad mide la eficiencia de un algoritmo cuando el número de datos N tiende a infinito.

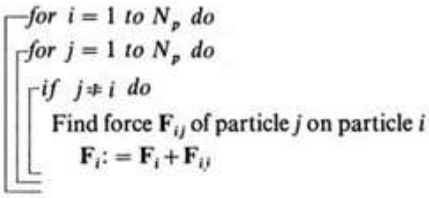


Imagen 2.2.2 Bucle sin optimizar para calcular las fuerzas.

El algoritmo definido en el *timestep loop*, a la vez que se calcula la fuerza total en la partícula i, se va acumulando la fuerza f_{ij} en el sumatorio de la partícula j, de esta forma, se evita tener que calcular f_{ji} ya que $f_{ij} = f_{ji}$ (imagen 2.3). Así, se pasa de tener un algoritmo del orden de complejidad O (n^2) a uno de O (n*log n). Para hacer una idea de la mejora lograda, teniendo 1000 partículas, el bucle sin optimizar tendría que realizar 1000000 iteraciones para poder calcular la fuerza en cada una de ellas, en cambio, el bucle optimizado necesitaría solo $\sum_{k=1}^{1000}(1000 - k)$ = 499500 iteraciones, menos de la mitad que el anterior caso.

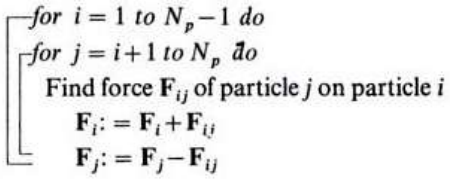


Imagen 2.2.3 Bucle para calcular las fuerzas optimizado para una ejecución secuencial.

Esta optimización es perfecta para una ejecución secuencial del algoritmo. En cambio, en una ejecución en paralelo, en la que cada hilo o *thread* va a ejecutar este bucle para calcular la fuerza neta sobre un único planeta, no es recomendable su uso. Los hilos se ejecutan sin un orden determinado, por lo que el hilo que calcule la fuerza sobre la partícula i no tiene forma de saber si el hilo que calcula la fuerza sobre la partícula j ha incrementado ya sus respectivas fuerzas netas con f_{ij} o no. El querer ir actualizando la fuerza de las partículas de los demás hilos conllevaría generar problemas de concurrencia innecesarios[2], en consecuencia, para la ejecución paralela, el bucle utilizado es el mostrado en la imagen 2.2.

Ahora que el algoritmo ya es paralelizable, se procede a añadir más exactitud en los cálculos. En el algoritmo clásico la forma en que se obtiene la nueva posición y velocidad en cada iteración del bucle es muy simple y acaba provocando problemas de precisión. Como solución, se ha utilizado el "método del salto de rana", más conocido con su nombre en inglés *Leapfrog Integration* [3]. Con éste método, las ecuaciones de movimiento quedan de la siguiente forma:

$$x_i^{NEW} := x_i^{OLD} + v_i^{OLD} DT + \frac{1}{2} a_i^{OLD} DT^2 \qquad (2,5)$$

[2] Como por ejemplo una condición de carrera (*race condition*) ocurre cuando dos o más procesos acceden un recurso compartido sin control, de manera que el resultado combinado de este acceso depende del orden de llegada

$$v_i^{NEW} := v_i^{OLD} + \frac{1}{2}\left(a_i^{OLD} + a_i^{NEW}\right)DT \qquad (2,6)$$

Hay que resaltar que para el cálculo de la nueva velocidad se necesita haber calculado previamente la nueva aceleración, es decir, la nueva fuerza en cada partícula en base a las nuevas posiciones. En consecuencia, es necesario variar el orden en que se ejecuta el algoritmo, de forma que la fuerza se actualice después de que los planetas se hayan movido y estén ya en sus nuevas localizaciones y antes de calcular la velocidad que les hará avanzar en la siguiente iteración.

En la imagen 2.4 ha sido modificada manualmente para añadir estos cambios nombrados. En ella, se puede apreciar el bucle *timestep loop* final.

Timestep loop of PP method

1. Update pos.

   for $i = 1$ *to* N_p *do*

   $\mathbf{x}_i^{new} := \mathbf{x}_i^{old} + \mathbf{v}_i^{old} DT + \frac{1}{2}\mathbf{a}_i^{old} DT^2$

2. Compute forces.

   Clear force accumulators

   for $i = 1$ *to* N_p *do*

   $\mathbf{F}_i := 0$

   Accumulate forces

   for $i = 1$ *to* N_p *do*

   for $j = 1$ *to* N_p *do*

   if $j \neq i$ *do*

   Find force $\mathbf{F}_{ij}$ of particle j on particle i

   $\mathbf{F}_i := \mathbf{F}_i + \mathbf{F}_{ij}$

3. Update vel.

   for $i = 1$ *to* N_p *do*

   $\mathbf{a}_i^{new} := \frac{\mathbf{F}_i}{m_i}$

   $\mathbf{v}_i^{new} := \mathbf{v}_i^{old} + \frac{1}{2}\left(\mathbf{a}_i^{old} + \mathbf{a}_i^{new}\right) DT$

4. Update time counter and ac.

   $t := t + DT$

   $\mathbf{a}_i^{old} := \mathbf{a}_i^{new}$

Imagen 2.2.4 Algoritmo PP optimizado.

3. LA GPU

Las tarjetas gráficas (GPUs) actuales, consisten en un gran número de procesadores programables denominados *shader cores*, los cuales ejecutan programas llamados *shaders*. Cada *core* tiene un rendimiento relativamente bajo: procesa una única instrucción del *shader* en uno o más ciclos de reloj y, normalmente, carecen de características de procesadores más avanzados como podrían ser la ejecución fuera de orden, la predicción de saltos, el superescalado…

Sin embargo, cada GPU cuenta con una gran cantidad de *cores*, que van desde unas pocas decenas a unos pocos miles, y juntos pueden llevar a cabo una cantidad inmensa de trabajo.

3.1. PIPELINE GRÁFICA

La mayoría de los sistemas gráficos siguen el paradigma de lo que se denomina *pipeline gráfica,* que consiste en un número de etapas, cada una representada bien por un *shader*, o bien por funciones fijas (consultar el capítulo 3 de [2]). Existen dos tipos de *pipeline*: la *rendering pipeline* y la *compute pipeline*.

La *pipeline* de renderizado es la *pipeline* clásica, cuya finalidad es la de renderizar o visualizar objetos 3D (ver imagen 3.1). La *compute pipeline* ha sido integrada recientemente en las tarjetas gráficas y tiene como objetivo permitir al usuario ejecutar cualquier operación o cálculo matemático usando los procesadores de la tarjeta.

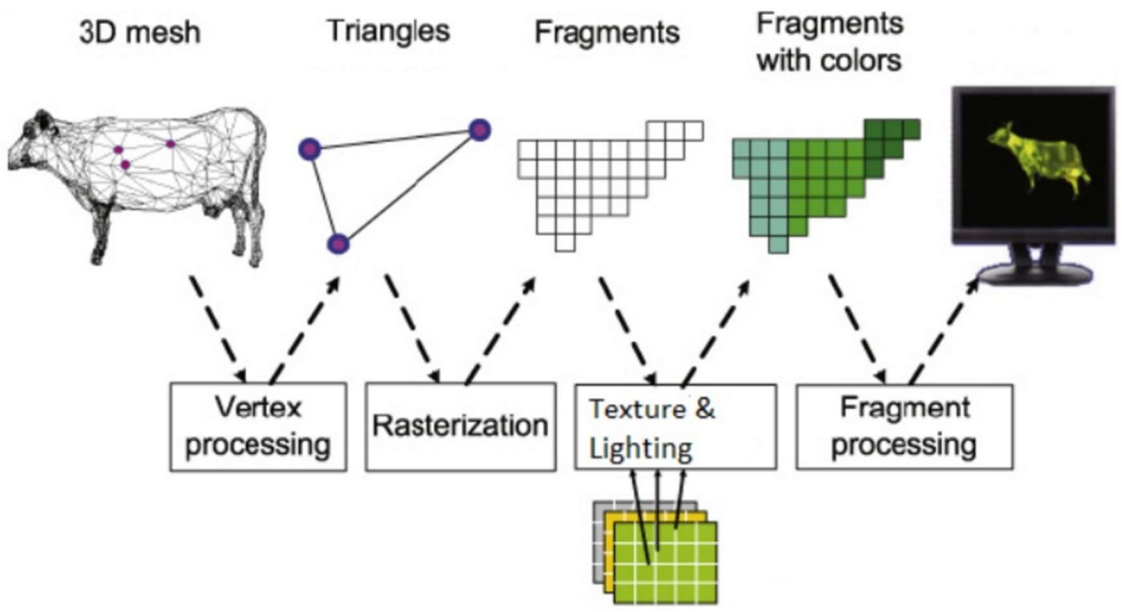


Imagen 3.1 Proceso de renderizado desde que un objeto 3D entra a la pipeline hasta que se muestra en la pantalla.

En este capítulo se van a presentar los distintos *shaders* que el programador tiene a su disposición (para un recorrido más en detalle de las *pipelines*, ver el apéndice D):

A continuación, se van a presentar las distintas etapas programables de ambas *pipelines*:

- *Vertex shader* (VS): Primera etapa programable de la *pipeline* de renderizado. El objetivo de esta etapa es el de procesar cada vértice de forma individual.

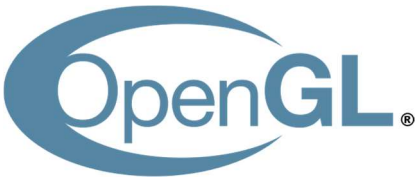
- *Tessellation Control Shader* (TCS) y *Tessellation Evaluation Shader* (TES): La teselación es el proceso de romper primitivas de alto orden, en muchas primitivas más pequeñas y simples por medio de subdivisiones recursivas.

- *Geometry shader* (GS): Esta etapa permite obtener nueva geometría a partir de la geometría original, agregando o sustrayendo vértices.

- *Fragment shader* (FS): Es la última etapa de la *pipeline* de renderizado. Su función es la de aplicar color a los píxeles que se mostrarán por pantalla.

- *Compute shader* (CS): La *compute pipeline* es una *pipeline* relativamente nueva, fue incorporada en la versión 4.3 de OpenGL, cuya función es presentar la GPU como una unidad de cálculo matemático de carácter general con una gran potencia. Consta de tan solo un *shader,* denominado *compute shader*.

3.2. OPENGL



OpenGL (*Open Graphics Library*) [3] es una especificación estándar que define una API (*Application Programming Interface*) escrita en C multilenguaje y multiplataforma para escribir aplicaciones que produzcan gráficos 2D y 3D. La API consiste en varios centenares de procedimientos y funciones diferentes que pueden usarse para dibuja escenas tridimensionales complejas a partir de primitivas geométricas simples siguiendo la *pipeline* explicada anteriormente. Fue desarrollada inicialmente por Silicon Graphics Inc. (SIG) en 1992 y se usa ampliamente en CAD, realidad virtual, representación científica y desarrollo de videojuegos entre otros campos.

El objetivo de OpenGL es el de proporcionar una capa de abstracción entre la aplicación y el sistema gráfico que se encuentra por debajo (ver imagen 3.2). Esta capa debe ocultar las diferencias entre las GPUs y los rasgos específicos de cada sistema, como la resolución de la pantalla, la arquitectura del procesador, el sistema operativo instalado entre otros. Por otro lado, el nivel de abstracción debe ser lo suficientemente bajo como para que el programador tenga el suficiente acceso al hardware que hay por debajo y pueda aprovechar toda su potencia.

[3] OpenGL: https://es.wikipedia.org/wiki/OpenGL, a fecha de septiembre 2016.

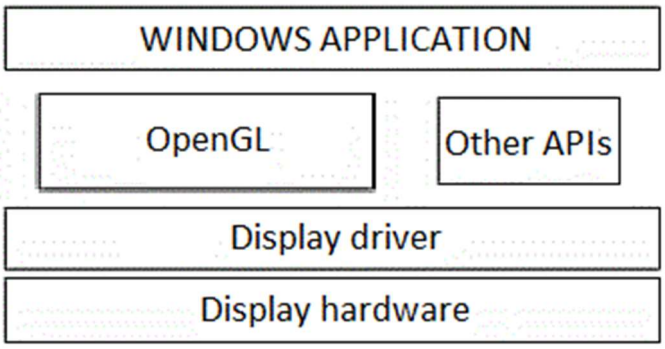


Imagen 3.2 Situación de la API de OpenGL en las capas del sistema.

3.1.1. PROGRAMACIÓN CON OPENGL

Un programa típico de OpenGL está formado por tres partes bien diferenciadas: la inicialización del contexto[4] de OpenGL, la inicialización de los recursos (explicados seguidamente) y el bucle principal. La imagen 3.3 muestra un esquema de esta estructura.

En primer lugar, hay que inicializar el contexto en el que se va a correr la aplicación. Dado que OpenGL es solo una biblioteca destinada a trabajar sobre la tarjeta gráfica, es necesario encontrar otra librería que permita crearlo. Como mínimo, ésta debe permitir la creación y el manejo de ventanas, así como proporcionar un método para procesar la interacción con el usuario. Existen varias librerías posibles, como Freeglut (obsoleta), SFML, SDL… Para este proyecto se ha elegido GLFW, una librería en C especialmente diseñada para trabajar con OpenGL. A diferencia de las anteriores, está solo cuenta con lo absolutamente necesario: creación de ventanas y la administración de la entrada de usuario. Además, ofrece un control bastante grande sobre la creación del contexto OpenGL.

En segundo lugar, hay que inicializar todos los recursos empezando por la librearía GLEW, librería multiplataforma escrita en C/C++ destinada a ayudar en la carga y consulta de extensiones de OpenGL. Una vez inicializada, es necesario declarar las variables y buffers que nuestra aplicación utilizará para comunicarse con la GPU. En este proyecto, se han creado hasta 4 programas, 2 de ellos destinados a ejecutar una *pipeline* de renderizado y otros 2 a una *compute pipeline*. También se ha creado un VAO (*Vertex Array Object*), un objeto de OpenGL que almacena toda la información relativa a los datos necesaria para nutrir al *vertex shader* de vértices, y hasta 3 VBO (*Vertex Buffer Object*) y 4 SSBO (*Shader Storage Buffer Object*), buffers que almacenan los datos que van a ir leyendo los *shaders* a lo largo de la pipeline (los datos que almacenen y la estructura delos buffers están descritos en el apartado 5.3). Además, es en este punto en el que se escribe en los buffers los datos iniciales.

[4] El contexto de OpenGL representa varias cosas. El contexto almacena todo el estado asociado a la instancia de OpenGL que se está ejecutando, la ventana donde se va a renderizar los objetos 3D, la interacción con el usuario por teclado o ratón, el *framebuffer*…

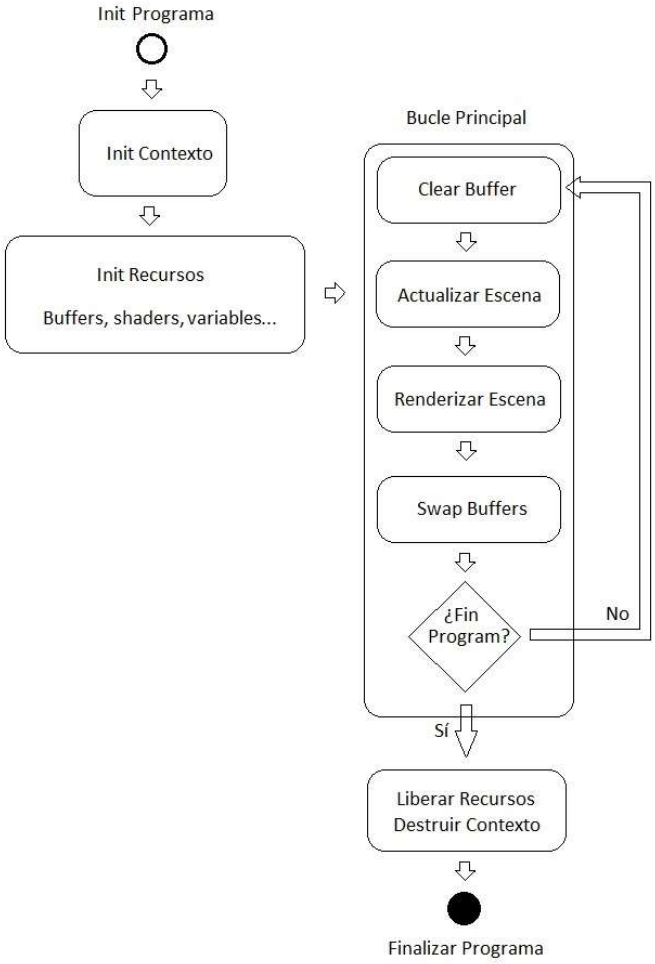


Imagen 3.3 Estructura básica de un programa de OpenGL.

Por último, es necesario construir el bucle principal de la aplicación. Este bucle consta de dos acciones: la primera, actualizar los datos almacenados en los buffers (si procede), y la segunda, renderizar los objetos 3D en la ventana. Es en este momento cuando se ejecutan las *pipelines* vistas anteriormente, la *compute pipeline* ya que su único propósito es el llevar a cabo cálculos, está ligada a la acción de actualizar, mientras que la *rendering pipeline* está ligada a la de renderizar. Este bucle hay que declararlo explícitamente mediante alguna estructura de bucle como, por ejemplo, un *do-while*.

4. PREPARACIÓN DE LA SIMULACIÓN

En este apartado se presentan todas las decisiones que se han tomado para la construcción del programa que ha permitido alcanzar los objetivos de este proyecto, las funciones implementadas y los resultados obtenidos para cada uno de las simulaciones realizadas consistentes en un escenario conocido como es el del Sistema Solar y el otro escenario consistente en sistema aleatorio desconocido.

4.1. PROGRAMAS UTILIZADOS

Como OpenGL es una API multilenguaje, ha sido necesario escoger el lenguaje en el que implementar el proyecto. Se ha seleccionado C++ debido a la buena integración con OpenGL. Recuérdese que la librearía de OpenGL está escrita en C, en caso de querer utilizarla en otro lenguaje es necesario descargar un paquete para enlazar ese código en al lenguaje seleccionado. Esta técnica es lo que se conoce como *language binding*. Para evitarlo, se ha elegido C++, lenguaje que extiende a C y que, además, permite la programación orientada a objetos, con todas las ventajas que presenta como puede ser la abstracción o la reusabilidad.

Por la comodidad que supone, se ha querido desarrollar el proyecto en un IDE (*Integrated Development Environment)* y de entre los que existen, se ha escogido el Visual Studio 2013 (VS-2013) Community como entorno de desarrollo debido a que es gratuito, viene con un compilador C++ y un *debugger* ya incluido, y cuenta además con muchas funciones, *plugins* y utilidades que facilitan y agilizan la creación de código. Uno de los *plugins* a destacar es el NVIDIA Nsight, plugin que permite ver el rendimiento de un programa de gráficos, ver el contenido inicial de los buffers de la GPU, detalles de la tarjeta gráfica, entre otros aspectos.

Con el VS 2013 se puede crear el programa en sí y visualizar todo el escenario en movimiento en 3D. Para verificar si los resultados eran correctos[5], se decidió utilizar dos programas externos que han permitido comprobar el buen funcionamiento del código de forma simple y visual. Estos programas son el MS Excel y el GNU Octave. El MS Excel, de la suite ofimática Microsoft Office, permite cargar datos en tablas y representar éstos en gráficos 2D. Además, ha sido el principal protagonista para el desarrollo de los primeros algoritmos. El GNU Octave es un programa libre para realizar cálculos numéricos, que cuenta con unos gráficos 2D y 3D mucho más potentes que el Excel. Asimismo, admite la ejecución de scripts permitiendo automatizar la ejecución de ciertas funciones.

4.2. ENTRADA Y SALIDA DEL ALGORITMO

Dado que el número de partículas a simular no va a ser el mismo de una versión a otra, se ha pensado en almacenar los datos iniciales en un fichero externo al programa, de forma que un cambio en las partículas de entrada no suponga modificación alguna de código.

[5] GNU Octave: https://www.gnu.org/software/octave/doc/v4.0.1/index.html

Se ha elegido el tipo de fichero CSV como fichero tanto para almacenar los datos de entrada como los resultados. Éste presenta la ventaja de que es un formato abierto muy sencillo cuyo propósito es representar los datos en forma de tabla. Otro factor importante a la hora de decidir usar este tipo de fichero es que tanto el lenguaje C++, el MS Excel como Octave lo aceptan de forma nativa.

4.3.ESTRUCTURA DE LOS BUFFERS DE LA GPU

Para la realización de este proyecto se han utilizado tanto VBO como SSBO. En los VBO se han almacenado datos que van a ser constantes a lo largo de toda la simulación, los cuales son los vértices de las esferas de los planetas, los vectores normales a esos vértices, y el orden en que consumirlos. Estos buffers van al *vertex shader* como datos de entrada y éste puede ir tratando los datos que contienen uno a uno conforme le llegan.

Un caso especial es el del buffer que almacena el orden de los vértices, llamado *index buffer*. No se puede acceder a su contenido dentro del *shader*. Su única función es la de indicar al *vertex fetching* en qué orden debe enviar los vértices del buffer al *vertex shader* para generar la geometría deseada.

Los SSBO son buffers de propósito general que utiliza el *compute shader* como entrada y salida de datos. Los datos que almacenan son las posiciones, velocidades, aceleraciones, masas, radios, grupo y el número de planetas. Como ya se ha visto anteriormente (ver imagen 4.13), estos buffers son accesibles desde ambas *pipelines*, así, todos los resultados del algoritmo PP ejecutado en el compute *shader* pueden ser directamente leídos en la *pipeline* de renderizado. Este es el caso de las posiciones y los radios.

En total, se han creado 4 VBOs y 2 SSBOs para la simulación del Sistema Solar y 3 VBOs y 4 SSBOs para la simulación aleatoria. La única diferencia que entre ambas simulaciones es que, en la primera, los radios son una constante durante todo el programa, por eso son un VBO, en cambio, en la simulación aleatoria se añade la detección de colisiones y la posterior unión de los planetas colisionados, por lo que el radio puede variar (ver capítulo 6). Además, derivado también de este cambio, el número de planetas puede disminuir, por lo que se almacena en un nuevo buffer.

4.4.DIAGRAMA DE CLASES

La imagen 4.1 muestra el diagrama completo del sistema.

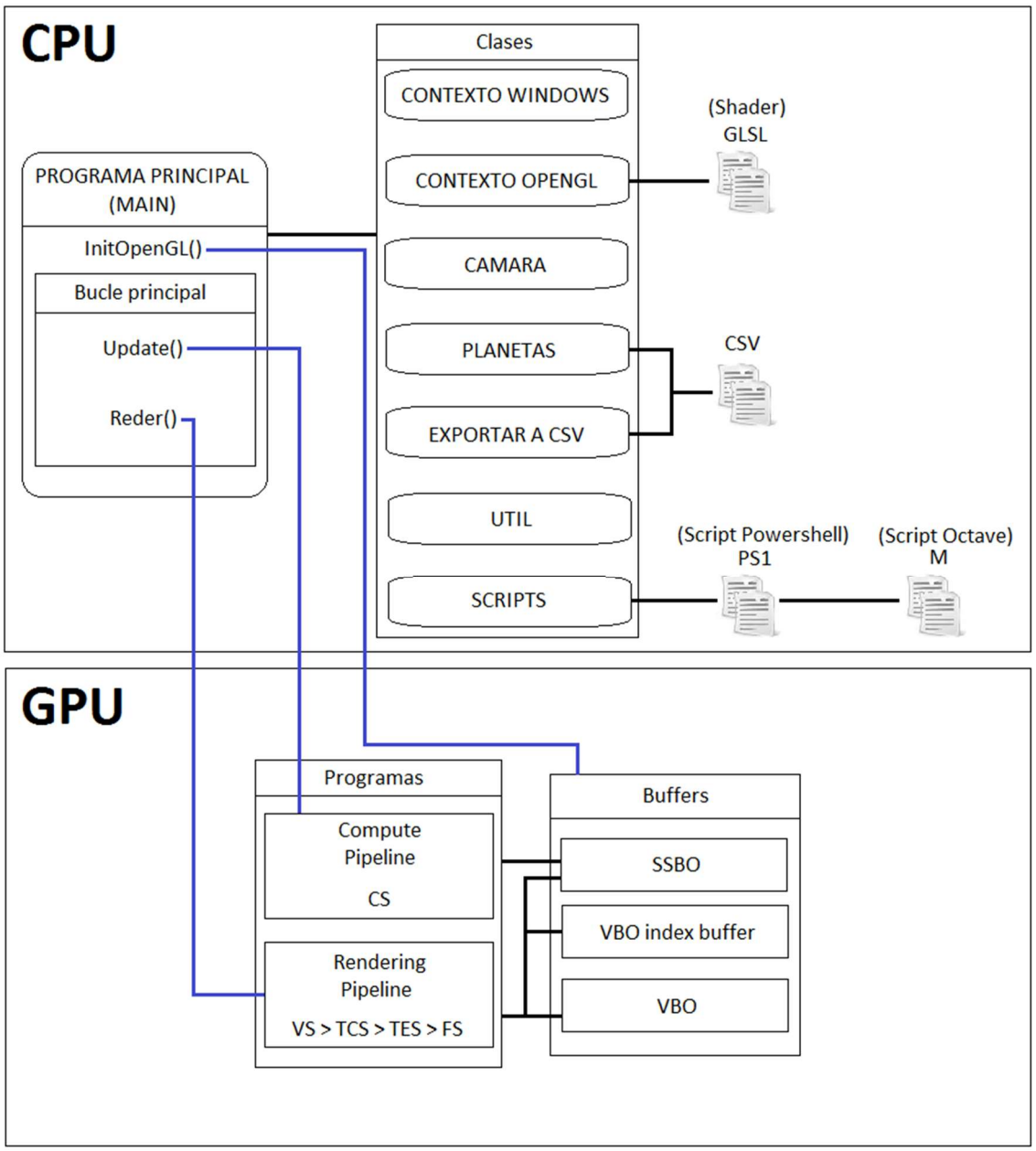


Imagen 4.1 Diagrama completo del programa.

4.5.DECISIONES OPENGL

No todos los astros del Sistema Solar tienen la misma forma. Por lo general, su geometría se aproxima a la de una esfera, aunque existen algunas excepciones como por ejemplo Deimos, el satélite de Marte, que presenta una forma muy irregular. Dado que el objetivo del proyecto no es el renderizado exacto de cada astro, se ha decidido representar a cada uno de ellos por una esfera de mayor o menor tamaño en función del radio medio del planeta.

Se ha decidido que las esferas fuesen de radio 1. Cada planeta tiene un radio medio distinto a los demás, esto implica que habría que calcular una esfera diferente para cada

uno de ellos y enviarla a los buffers definidos en la memoria de la GPU para que ésta pudiera procesarlos. Además de la cantidad de espacio de memoria que haría falta para poder almacenar todos esos vértices, hay que tener en cuenta que los cálculos que generan la esfera se realizan sobre la CPU, lo que generaría una carga computacional muy grande que ralentizaría el programa. La solución más óptima era definir una única esfera de radio 1, cuyo centro se situara en el origen de coordenadas, y enviarla a la GPU, de forma que fuera el TES quien la adaptara a las características de cada astro, aplicando las matrices de transformación adecuadas. Con esto se consigue no tener que modificar la información relativa a los vértices (posición, vectores normales y coordenadas de textura, que en este proyecto no se utilizan) de los *VBO* con cada planeta.

Además, como todos comparten la misma geometría, se puede utilizar un método denominado *Instanced Rendering*, mediante el cual, se informa a OpenGL, en una única llamada de dibujado, que se desea dibujar varias copias de la misma geometría. Esto evita el tener que realizar una llamada de dibujado por cada astro que se quiera renderizar. De esta manera, se pasa de invertir la mayor parte del tiempo en llamadas a funciones de dibujar de OpenGL a invertirlo en el renderizado en sí.

Siguiendo esta filosofía de adaptar la geometría inicial en los *shaders*, se ha implementado un método de LOD (*Level of Detail*) [2] de forma que la geometría que recibe la *pipeline* gráfica es una esfera con tan pocos vértices que pierde cualquier característica típica de esfera. Esta geometría con tan pocos polígonos es ideal para representar planetas tan lejanos que se muestran como si fueran puntos en la ventana de visualización. Para los planetas más cercanos, se utiliza el proceso de teselación para generar más vértices con los que poder ir dando ese carácter esférico al objeto. La imagen 4.2 muestra este LOD.

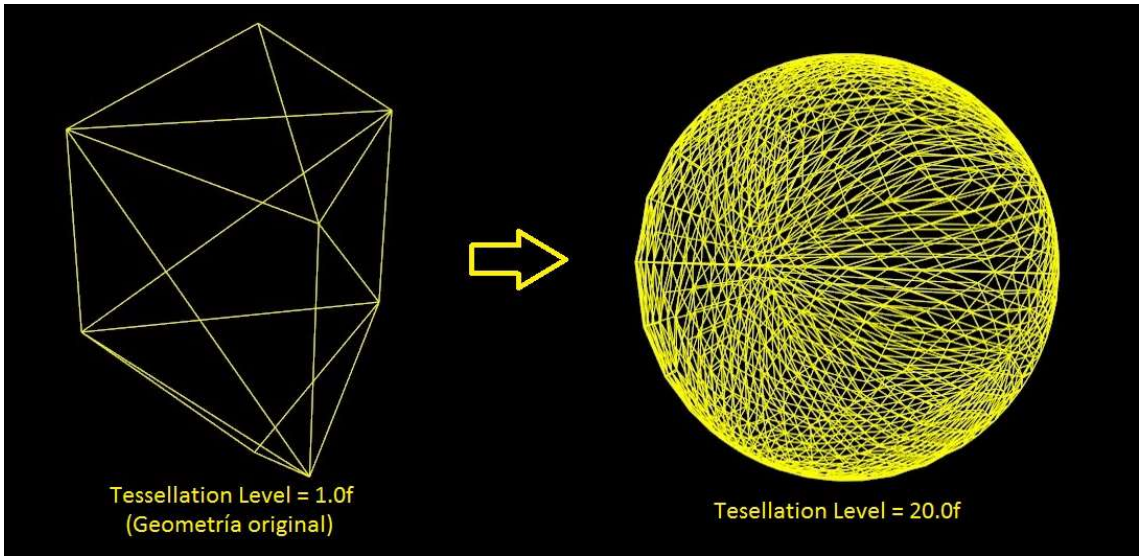


Imagen 4.2 LOD (Level of Detail) de los planetas (izquierda - planeta lejano, derecha - planeta cercano).

Dependiendo de la distancia a la que se sitúen los planetas de la cámara tendrán más vértices o menos, así pues, se logra que aquellos planetas muy cercanos se vean como esferas propiamente dichas y, conforme se alejen, irán perdiendo definición. Éste método tiene una influencia directa en los FPS (*Frames Per Second*) de la aplicación, otorgando una gran mejora de rendimiento.

Otra decisión tomada en relación con OpenGL ha sido la de desactivar el *depth test*. El *depth test* es uno delos test que aplica OpenGL prácticamente al final de la *rendering pipeline*, después de que el *fragment shader* se haya ejecutado. Resumiendo, el *depth test* permite distinguir qué objetos están situados más cerca de la cámara y cuales están más lejos, de esta manera, si se da el caso que un objeto lejano quiere dibujarse en un mismo pixel que un objeto cercano, será el más cercano el que terminará siendo mostrado (ver el apéndice D para una información más detallada).

El desactivar el *depth test*, se ha decidido después de observar que los planetas situados muy lejos de la cámara, dependiendo de su posición, iban despareciendo y reapareciendo. El problema radica en un fallo en la precisión de los cálculos que realiza el *depth test* para determinar si un polígono se va a mostrar o no (ver apéndice C para más información).

4.6.CARACTERÍSTICAS DE LA TARJETA GRÁFICA

Para la realización de todas las simulaciones se ha utilizado una tarjeta gráfica Nvidia GeForce GTX 660M con las siguientes características:

GPU	Núcleos CUDA	384 núcleos
	Frecuencia del reloj	835 MHz
	Tasa de relleno de texturas	30.4 GTexel/s
Memoria	Frecuencia del reloj	2000 Mbps
	Interfaz de memoria	128-bit GDDR5
	Ancho de banda máximo	64 GB/s

Cuadro 4.1 Características tarjeta gráfica Nvidia GeForce GTX 660M.

Para poner en contexto la tarjeta gráfica, Nvidia nombra las tarjetas gráficas según su serie (en este caso es la serie 600 Mobile) y en cada generación aumenta la serie. Nvidia saca una nueva generación cada año. Actualmente, acaba de salir al mercado la serie 1000. En la imagen 4.3 se compara una de las tarjetas de serie 1000 con una tarjeta de serie 600[6]. Se puede apreciar que desde que se empezó el proyecto han salido tarjetas gráficas mucho más potentes (recordar que en este proyecto se está usando una serie 600 Mobile, cuyas características y prestaciones son peores que la serie 600 normal).

[6] Comparación obtenida de http://hwbench.com/vgas/geforce-gtx-1060-vs-geforce-gtx-660, a fecha de septiembre del 2016

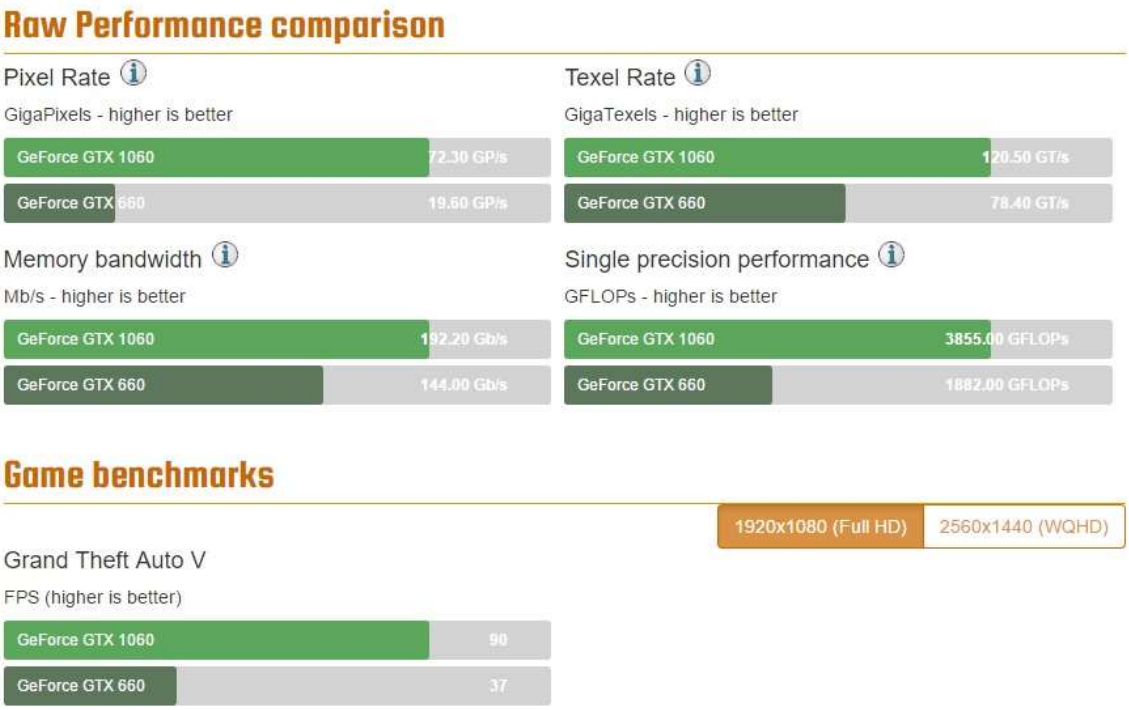


Imagen 4.3 Comparación tarjetas gráficas GeForce GTX 660 y GTX 1060.

5.SIMULACIÓN DEL SISTEMA SOLAR

5.1.OBJETIVO

La primera simulación que se va a realizar para verificar el correcto funcionamiento del sistema diseñado. Ha sido elegido el Sistema Solar, ya que cumple una serie de requisitos relacionados todos con el conocimiento profundo que se tiene del mismo.

Las principales características son: en primer lugar, es un sistema estable, es decir, ninguna partícula se aleja tanto como para anular las fuerzas que le ejercen todas las demás y desplazarse hasta el infinito. Además, contiene una gran variedad de partículas, variedad en tamaño, en masa y en el eje sobre el que describen su trayectoria. Por último, es un sistema del que se conoce abundante información, como las trayectorias de cada partícula, sus dimensiones, las fuerzas a las que se ven sometidas, las velocidades a las que se mueven, etc.

Si el algoritmo implementado es capaz de reproducir el comportamiento de dicho sistema, se considerará un algoritmo adecuado.

5.2.PROBLEMA DE LOS 2 CUERPOS

El objetivo de empezar por resolver el problema de los dos cuerpos reside en obtener un algoritmo base que funcione correctamente. Por simplicidad, y dado que la carga computacional no es muy grande al tener únicamente dos cuerpos, se ha elegido trabajar directamente sobre MS Excel, utilizando el sistema de macros en Visual Basic que viene con él. De esta forma, se consigue tener una primera aproximación del algoritmo PP sin preocuparse de todo lo que un programa OpenGL conlleva (creación de ventanas, contexto, etc.).

Se ha elegido que uno de los cuerpos sea el Sol y el otro sea La Tierra. En este algoritmo, los datos iniciales son conocidos y están almacenados en una página del mismo documento Excel, por tanto, no es necesario realizar ningún cálculo previo para obtenerlos.

El algoritmo utilizado para realizar la prueba es el algoritmo PP básico, sin utilizar el integrador de Leapfrog, y los resultados que se obtienen son los esperados: La Tierra logra dar una vuelta completa alrededor del Sol sin problema, describiendo una trayectoria elíptica con las posiciones esperadas.

Una vez logrado lo más básico, ahora hay que avanzar en la complejidad del sistema y lograr que no solo La Tierra gire alrededor del Sol, sino que el sistema Sol – Tierra - Luna sea un sistema estable y que sus trayectorias sean las esperadas. Una vez que esto se ha conseguido, el evidente paso siguiente es lograr reproducir el comportamiento de todo el Sistema Solar.

Para alcanzar los nuevos objetivos se añadió el integrador de *Leapfrog* para evitar posibles problemas de precisión en los cálculos. El resultado obtenido concuerda con lo que la teoría indicaba.

Tanto la Luna como La Tierra describen la misma trayectoria alrededor del Sol. Si se acerca la vista a un tramo de esa trayectoria, se puede ver como la Luna no está superpuesta a La Tierra, sino que va oscilando alrededor de ella (ver imagen 5.1). Como se puede apreciar, los resultados son todos correctos. Se procede a implementar un programa que mueva el Sistema Solar completo.

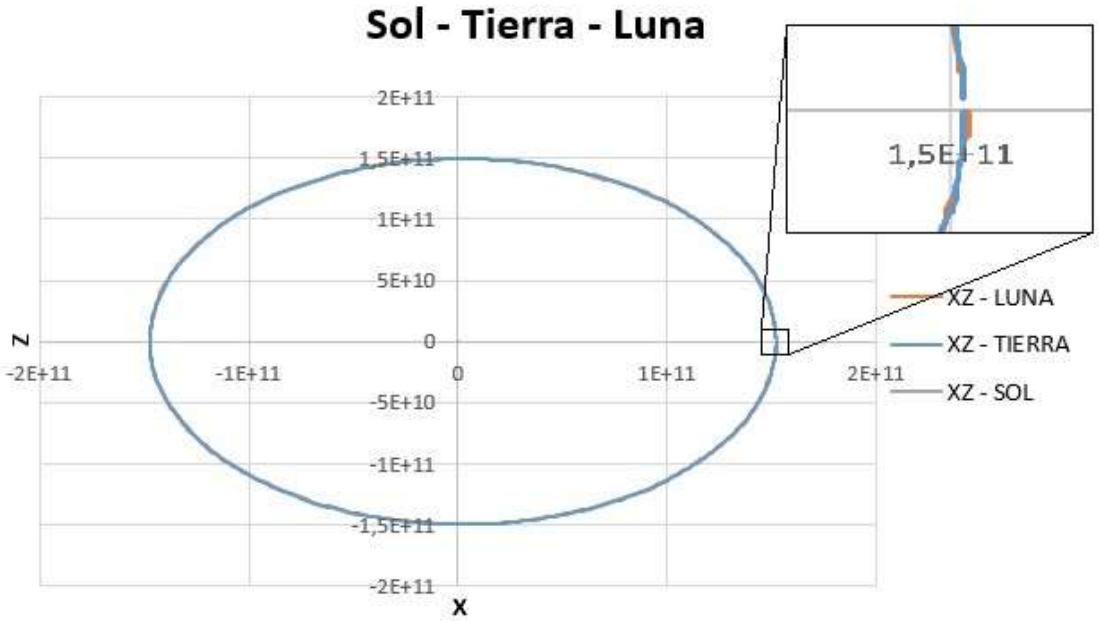


Imagen 5.1 Sistema Sol - La Tierra – Luna.

5.3.SISTEMA SOLAR

El algoritmo "Partícula - Partícula" es un algoritmo de integración temporal que requiere las condiciones iniciales adecuadas para integrar una ecuación diferencial ordinaria de 2º orden, por lo tanto, es necesario nutrirle de unas posiciones y velocidades iniciales a partir de las cuales calcular las siguientes.

Aunque se conoce perfectamente la trayectoria de la gran mayoría de los astros del Sistema Solar, es necesario elegir un punto en concreto de la misma para utilizarlo como inicio del algoritmo.

Se sabe que las trayectorias que describen los planetas alrededor del Sol (o de su planeta en el caso de los satélites) son trayectorias elípticas con más o menos excentricidad, así que se pueden usar las reglas matemáticas de la elipse para hallar los datos necesarios de cada uno. Tomando los ápsides de cada planeta y la inclinación de su trayectoria respecto al plano de la eclíptica, se puede calcular cualquier posición de su recorrido. En este caso, se ha elegido tomar el punto más alejado de su trayectoria (el apoápside) como posición inicial. En cuanto a su velocidad, se ha elegido que los planetas describan una trayectoria anti horaria vista desde posiciones de Y positivas.

Las fórmulas serían las siguientes (ver apéndice A para ver cómo se han obtenido):

$$a = \frac{Periápside + Apoápside}{2} \tag{5,1}$$

$$V_0 = \sqrt{2GM_{Eje}(\frac{1}{Apoápside} - \frac{1}{2a}) + V_{Eje}} \tag{5,2}$$

$$\overrightarrow{V_0} = (0, 0, -V_0) \tag{5,3}$$

$$X_0 = (Apoápside * \cos\varphi\,, Apoápside * \sin\varphi\,, 0) + X_{Eje} \tag{5,4}$$

M_{Eje}: masa de su planeta eje
ϕ: inclinación orbital

La cantidad de astros con los que se va a trabajar a partir de este punto hace que el utilizar Excel para desarrollar el algoritmo ya no sea una opción viable. Hay que recordar que Excel, aunque disponga de herramientas para implementar programas, es un sistema ideado para trabajar con hojas de cálculo. La carga computacional que supone el realizar todos los cálculos del algoritmo PP a la vez que inserta los resultados en tablas, hace que el programa se sature y el tiempo que tarda en terminar una iteración completa del *timestep loop* es muy elevado.

Se procedió a crear un programa OpenGL en C++ completo. Es en este momento cuando al algoritmo PP se le añade el sistema de entrada y de salida de datos por medio de un fichero CSV.

5.1.1. VERSIÓN 1: CPU

En esta primera versión, toda la carga computacional se concentraba en la CPU y la GPU se limita a renderizar los resultados. La idea principal es crear un marco sobre el que trabajar posteriormente.

Todas las funciones que implementan el algoritmo "Partícula – Partícula" se van a ejecutar en el procesador. Esto permite realizar una ejecución paso a paso con la que se puede depurar fácilmente el código. Además, se implementan las funciones que controlan la ventana y sus eventos y se inicializan los recursos necesarios para trabajar con OpenGL. Este paso es necesario para tener una base libre de errores con todos los recursos que se van a necesitar.

5.1.2. VERSIÓN 2: CPU-GPU

Con esta versión, la tarjeta gráfica adquiere más protagonismo al implementar todos los cálculos dentro del *compute shader*. Aun así, el procesador sigue manteniendo considerable carga debido a que en cada iteración se introducen los datos antiguos al

SSBO, se devuelven los datos actualizados a la CPU y, posteriormente, se vuelven a enviar a la GPU como parámetros de entrada del *vertex shader* para su renderizado.

Esta versión no fue la definitiva, siguió siendo una versión cuyo objetivo era comprobar que todo funciona correctamente y que todos los cálculos eran válidos. El coste de sacar los resultados de la GPU a la CPU en cada iteración es muy elevado y repercute enormemente en los FPS de la aplicación, pero es el único modo de comprobar que los datos con los que trabaja la tarjeta gráfica son correctos, ya que no se puede acceder directamente desde el procesador a su memoria.

5.1.3. VERSIÓN 3: GPU

Por último, esta versión concentra toda su actividad en la tarjeta gráfica. El procesador se limita a crear la ventana, tratar los eventos de teclado y ratón y a realizar las llamadas de las funciones de OpenGL en cada iteración.

Toda la información necesaria para el algoritmo (posiciones, velocidades, aceleraciones, masas y radios de los planetas) está almacenada en los buffers de la tarjeta gráfica, y son estos buffers los que se van enlazando como entradas y/o salidas de los diferentes shaders conforme se van necesitando. No es necesario llevar ninguna información a la CPU, ya se sabe que los valores con los que trabaja el algoritmo son correctos gracias a las versiones anteriores.

5.4. RESULTADOS OBTENIDOS

Desde el punto de vista de la ejecución, el resultado alcanzado es bueno. A pesar de que no se ha podido reproducir todas las trayectorias de los cuerpos de forma exacta, sí que se ha logrado que el sistema fuera estable en su mayor parte. De un total de 141 cuerpos que estaban involucrados en la simulación, 49 de ellos se pierden en el infinito, o lo que es lo mismo, un 35% aproximadamente de los cuerpos no se estabilizan.

Se está trabajando con números muy grandes, valores del orden de 10^{10} en distancias y de 10^{25} en masas, por lo que es probable que algunos cálculos no sean lo suficientemente precisos. Además, las fórmulas para obtener la posición y la velocidad, a pesar de utilizar el integrador de Leapfrog, no son las más exactas para este tipo de cálculos. Recordemos que la Mecánica Celeste trabaja en este tipo de problemas y los observatorios astronómicos, así como la NASA, utilizan métodos numéricos bastante sofisticados.

Otro dato a tener en cuenta son las condiciones iniciales del algoritmo. El algoritmo PP es un algoritmo recursivo, necesita unas condiciones iniciales a partir de las cuales iniciar el proceso. Dado que no se ha podido encontrar la posición exacta, ni la velocidad ni aceleración, de todos los cuerpos para un instante i determinado, es posible que la elegida para iniciar la simulación en este proyecto no sea la adecuada y genere fallos en los resultados. Por todo ello, el hecho de que el 35% delos cuerpos no se consiga estabilizar, se puede considerar como un resultado indicativo de que el algoritmo PP implementado es válido.

En las imágenes conseguidas con los gráficos MS Excel (ver imágenes 5.2 y 5.3), se puede ver cómo la trayectoria de los planetas es correcta casi perfectamente, y cómo la trayectoria de los satélites, si bien no es la que debiera en muchos casos, por lo menos

es estable. Aquellos satélites que no se estabilizaban se han decidido quitar de los gráficos por visibilidad.

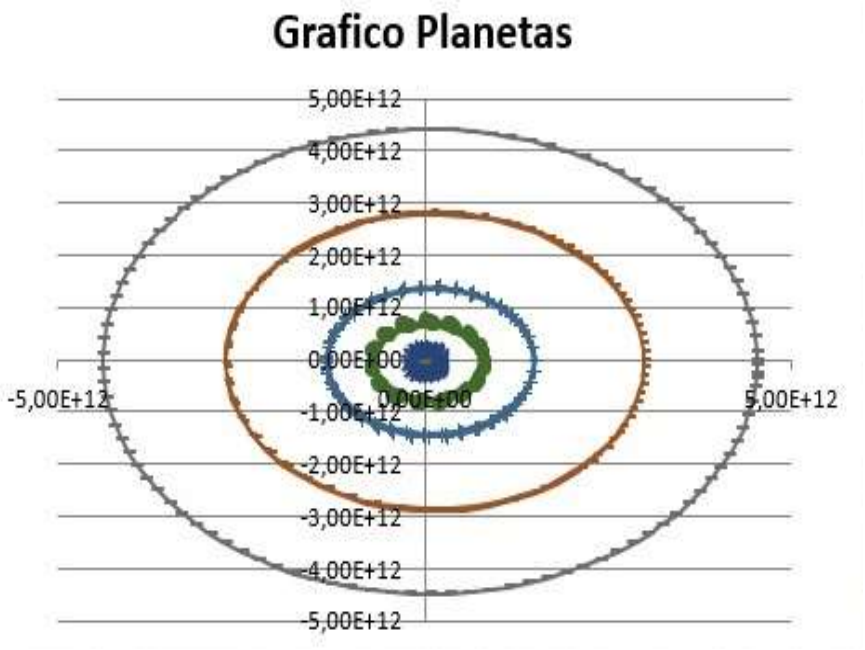


Imagen 5.2 Trayectoria recorrida por los planetas.

Imagen 5.3 Trayectoria recorrida por los satélites.

La imagen 5.4 está obtenida con Octave utilizando los mismos datos que para las imágenes 5.2 y 5.3. Ésta permite ver la evolución del escenario de una forma más completa gracias a la vista 3D.

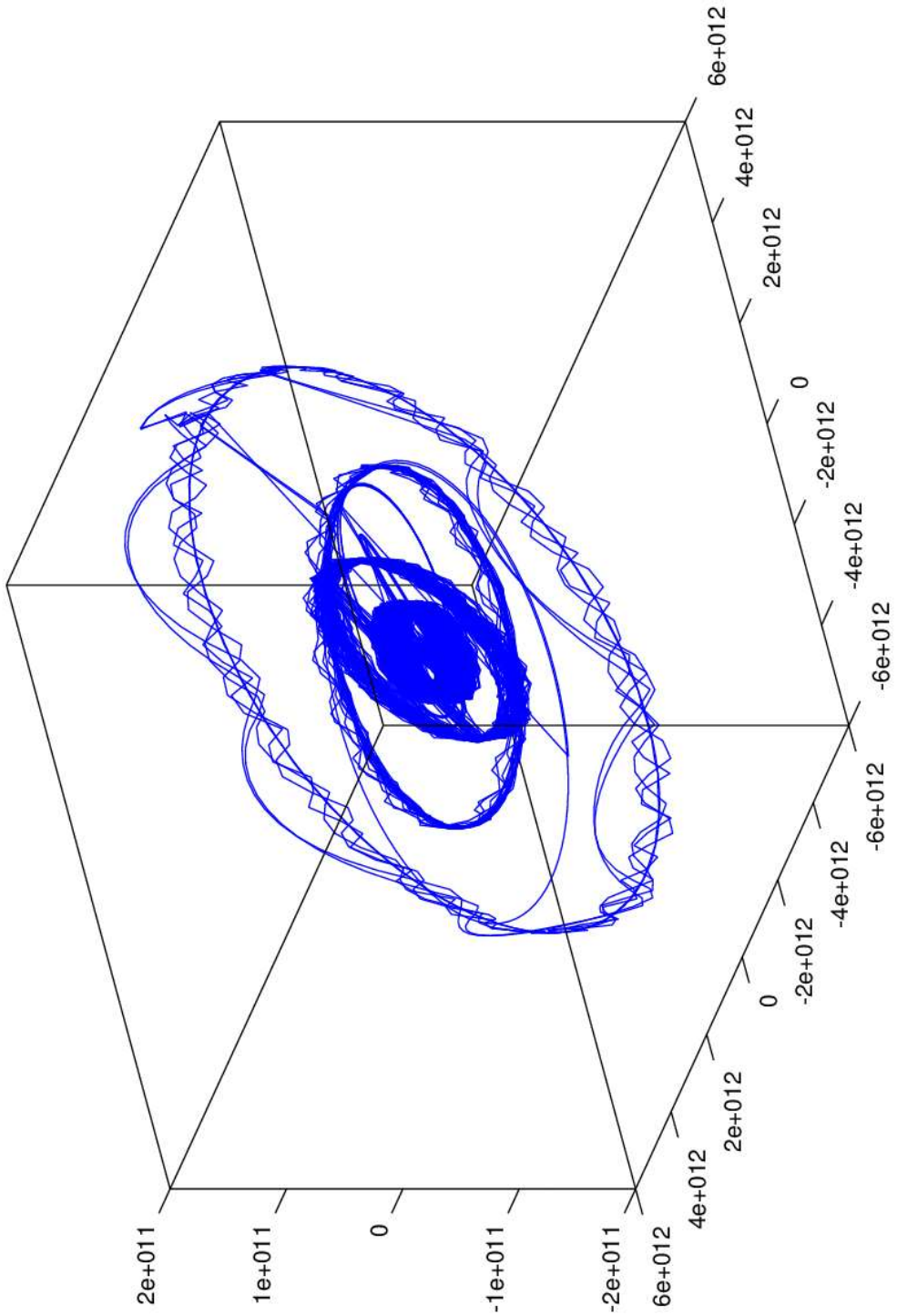


Imagen 5.4 Captura final de la simulación del Sistema Solar.

Dado que la cantidad de planetas que se van a mover es relativamente pequeña, incluso la versión primera es capaz de renderizar todo con unos buenos FPS. A continuación, se presentan algunas capturas del renderizado 3D de este sistema.

Utilizando el *plugin* Nvidia Nsight, se consigue mostrar, además, algunos datos del rendimiento de la simulación. Como era de esperar, la versión que utiliza la CPU (ver

imagen 5.5) para realizar los cálculos del algoritmo es algo más lenta que las demás, aun así, se estabiliza en unos 46 FPS, manteniéndose por encima de los 25 FPS que es lo que define un renderizado en tiempo real[7]. La versión final presenta una mejora importante de rendimiento, llegando a las 60 FPS (ver imagen 5.6).

Imagen 5.5 Captura de la ejecución de la versión CPU de la simulación del Sistema Solar, en la que se pueden ver varios planetas y satélites (a destacar el Sol y Júpiter rodeado de sus satélites). Se muestran algunos datos del rendimiento como los FPS estabilizados en 46 FPS (derecha).

Imagen 5.6 Captura de la ejecución de la versión final (GPU) de la simulación del Sistema Solar, en la que se pueden ver varios planetas y satélites (a destacar el Sol y Júpiter rodeado de sus satélites). Se muestran algunos datos del rendimiento como los FPS estabilizados en 60 FPS (derecha).

[7] La velocidad mínima a la que una consecución de imágenes es considerada como un movimiento fluido por el ojo humano es de 25 FPS aproximadamente.

Es interesante observar en qué acciones se distribuye el trabajo de la GPU. En la imagen 5.7 se están observando 4 medidas: *Geom busy*, *shader busy*, *texture busy* (este se puede omitir ya que no se han utilizado texturas en todo el proyecto, por lo que siempre va a ser nulo) y *gpu idle*.

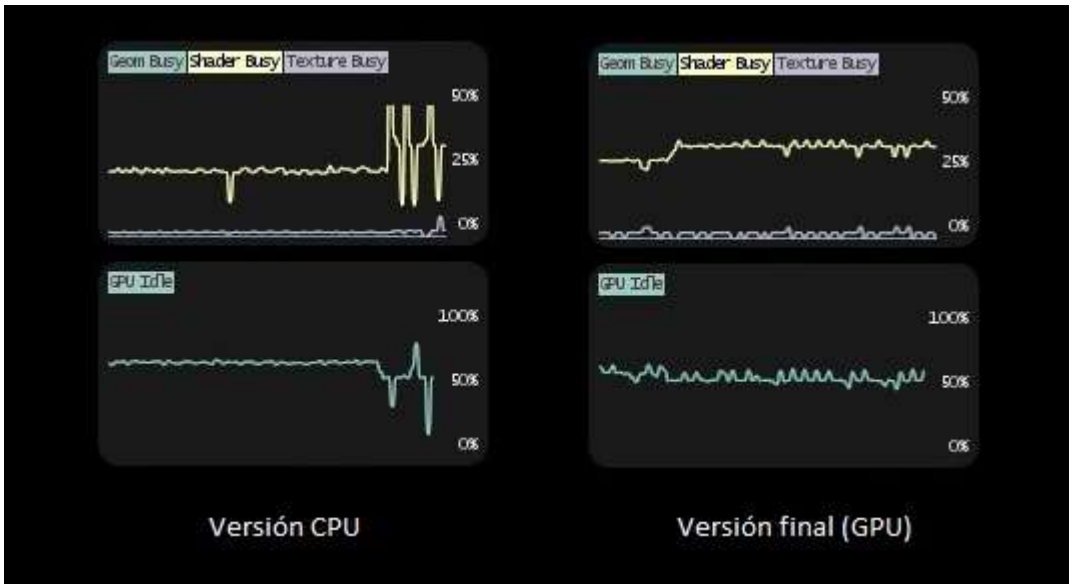


Imagen 5.7 Distribución de las acciones de la GPU en la simulación del Sistema Solar.

El *geom busy* mide la cantidad de trabajo que la GPU dedica a mover los vértices desde la unidad que los distribuye hasta los *shaders* donde serán transformados. Llamadas de dibujado con una larga ristra de vértices, o bien un gran número de llamadas consecutivas con un pequeño número de vértices, pueden ser causas potenciales de que esta medida se incremente. En el caso de este proyecto, a pesar de que ambas versiones presentan un *geom busy* pequeño, el de la versión de la CPU es inferior que el de la versión final. Hay que recordar que, en la versión final, todos los cálculos se realizan en el *compute shader* dentro de la GPU, limitando el uso de la CPU a la preparación e invocación de las *pipelines* gráficas, por lo tanto, la cantidad de datos que se tienen que mover a la unidad que ejecuta los *shaders* es mucho mayor (no solo hay que mover los vértices, sino también las posiciones, velocidades y aquellas características necesarias para ejecutar el algoritmo PP).

El *shader busy* mide la cantidad de trabajo que la GPU dedica a la ejecución de todos los *shaders* que se han definido en el programa. Para este proyecto, se ha definido un *shader* adicional en la versión final, el *compute shader*, de ahí que sea mayor que en la versión de CPU.

Por último, el *gpu idle*, mide la cantidad de recursos que no están siendo utilizados en la GPU. En ambas versiones, aproximadamente la mitad de la GPU no está siendo utilizada. Esto quiere decir que la tarjeta gráfica no ha llegado a su límite todavía, es capaz de realizar simulaciones con un número mayor de partículas. Al realizar la simulación aleatoria (capítulo 6), se ha establecido la condición de que esta medida sea prácticamente nula.

Para finalizar, se presenta una captura que muestra la utilización tanto de la GPU como de la CPU con la versión final (ver imagen 5.8).

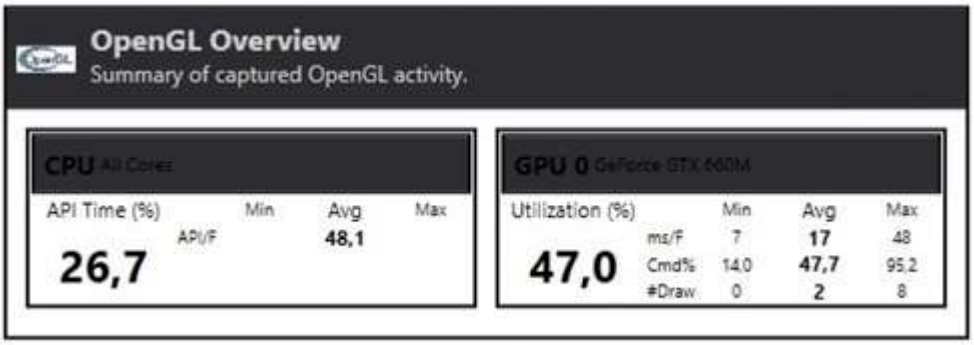


Imagen 5.8 Rendimiento Sistema Solar.

Tal como se puede apreciar, y en concordancia con la medida *gpu idle* anteriormente mencionada, la GPU utiliza aproximadamente la mitad de su potencial. Cabe destacar los datos recopilados sobre la CPU. El tiempo que ha consumido en llamadas a la API de OpenGL con respecto al tiempo total de la aplicación es solo una cuarta parte. Existen muchas llamadas a funciones de OpenGL, por ejemplo, las destinadas a configurar funciones no programables de la pipeline, las destinadas a crear y manejar los buffers y los *shaders* o las destinadas a enviar datos de la memoria del programa a la GPU, que tienen que realizarse obligatoriamente. La implementación de la técnica *Instanced Rendering* (explicado ene l capítulo 4.5) ha permitido reducir considerablemente el número de llamadas de dibujar que se hubieran hecho sin ella.

6.SIMULACIÓN DE UN SISTEMA ALEATORIO

6.1.OBJETIVO

Llegados a este punto, lo que se quiere lograr es conocer el límite de trabajo de la GPU en este tipo de problemas, es decir, conocer cuál es la configuración del problema que hace que la representación gráfica baje de unos FPS aceptables, La velocidad a la que una secuencia de imágenes deja de ser vista como un movimiento fluido por el ojo humano es de entre 25 y 30 FPS, para este trabajo, se ha dado un poco de margen y se ha considerado como aceptable una velocidad de alrededor de 15 FPS.

Partiendo del algoritmo implementado para el escenario anterior, se va a crear uno ligeramente diferente, de forma que permita la simulación de un número de planetas del orden de 10^3, además de una serie de mejoras que se van a detallar en el apartado siguiente.

6.2.CAMBIOS EN EL ALGORITMO

Además de las mejoras con las que ya cuenta, se añaden las siguientes: detección de colisiones entre planetas, fusionado de planetas colisionados y restringir el cálculo de las fuerzas a los planetas cercanos.

Dado que cabía la posibilidad de que dos o más planetas chocasen, se ha implementado un sistema para detectar colisiones y unificar aquellos planetas que colisionen. La gran cantidad de planetas que se simulan no permite generar un sistema de colisiones que recree fielmente la realidad, por ello se ha simplificado el problema teniendo en cuenta solo las colisiones directas entre dos planetas.

La colisión y posterior unión de dos planetas sí que se ha intentado hacer de una forma realista, utilizando para ello, se han utilizado las siguientes fórmulas que calculan la nueva masa y el nuevo radio:

$$M_{NEW} = M_1 + M_2 \tag{6,1}$$

$$V_{NEW} = V_1 + V_2;\ V = \frac{4}{3}\pi R^3 \rightarrow \frac{4}{3}\pi(R_{NEW})^3 = \frac{4}{3}\pi(R_1)^3 + \frac{4}{3}\pi(R_2)^3 \rightarrow$$
$$\rightarrow R_{NEW} = \sqrt[3]{{R_1}^3 + {R_2}^3} \tag{6,2}$$

$$V: volumen$$

Para obtener el nuevo vector velocidad simplemente se aplica una suma vectorial de las velocidades de los dos planetas colisionados, y la nueva posición será el punto medio de la línea que une sus posiciones. Uno de los planetas actualizará sus características

siguiendo estas fórmulas, el otro en cambio, pondrá su masa a 0 para dar a entender que ha chocado contra otro y que ya no es válido. Estos planetas anulados se irán moviendo al final del buffer de la GPU para mantener un cierto orden. Además, habrá que recorrer el buffer para llevar la cuenta del número de planetas que siguen activos, dato necesario a la hora de utilizar la función de dibujado de OpenGl. Para evitar que este recuento suponga una carga computacional adicional, será realizada por los hilos que actualizan planetas que ya han sido eliminados a la vez que los demás hilos están ejecutando el algoritmo PP.

Por último, se ha implementado una organización estructural de los planetas realizando agrupaciones dependiendo de la zona del espacio en la que se sitúan. La idea es que como la fuerza de la gravedad decrece con el cuadrado de la distancia (decrecimiento muy rápido) no todos los planetas afectan a todos los demás de una manera intensa, por ello se restringen los cálculos de manera que el algoritmo solo se aplicará a aquellos planetas que residan en el mismo grupo, siendo cada grupo independiente de los demás.

Una última modificación, derivada de la agrupación de los planetas es que éstos se ordenan por grupo dentro del buffer. De esta forma, a la hora de calcular las fuerzas que ejerce cada planeta sobre cada uno de los demás planetas de su grupo, el algoritmo no tendrá que recorrer todo el buffer para buscarlos.

Se espera que con estas optimizaciones se mejore el rendimiento general del programa y se admitan más planetas manteniendo la misma tasa de FPS que se hubiera encontrado sin ellas.

6.3.POSICIONES ALEATORIAS

Uno de los factores clave para este escenario es que las posiciones de cada planeta se recalculan con cada ejecución del programa. Esto es una condición necesaria porque el algoritmo implementado no debe depender de unos datos iniciales determinados. Existen librerías en C++ que ya cuentan con un generador de números aleatorios, pero dada la magnitud de los valores con los que se trabaja en este proyecto, no daba unos resultados óptimos, por lo que se ha construido un script para Octave que los calcula utilizando las diferentes distribuciones numéricas que dispone, en concreto, para generar las posiciones y los radios de los planetas se ha utilizado una distribución uniforme (ver imágenes 6.2 y 6.3), mientras que el cálculo de las masas se ha realizado siguiendo una distribución beta (ver imagen 6.1).

Para esto último se ha elegido una distribución no uniforme por el hecho de que, si todos los planetas cuentan con una masa similar, la densidad de masa que rodea a cada planeta será más o menos la misma en todas las direcciones, por lo que el planeta en cuestión no se vería apenas perturbado. Con esta distribución, nos aseguramos de que haya muchos planetas con masas relativamente pequeñas y pocos con masas grandes evitando el efecto anterior.

Se ha tenido que ajustar los límites entre los que se sitúan cada uno de estos valores para llegar a un compromiso entre las distancias entre los planetas y las masas, de forma que, las masas sean lo suficientemente grandes como para llegar a atraer a todos los planetas independientemente de su posición, pero no tanto como para generar una fuerza de tal

magnitud que los lanzara fuera del volumen estudiado. En este sentido, la distancia media entre planetas es de $1.9 \cdot 10^{19}$ u.m. y la masa media se ha establecido en 10^{50} para un espacio de tamaño $3 \cdot 10^{19}$ x $3 \cdot 10^{19}$ x $3 \cdot 10^{19}$ u.m.

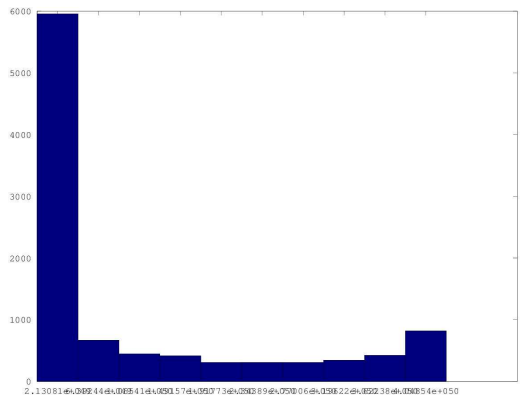


Imagen 6.1 Distribución beta – Masas.

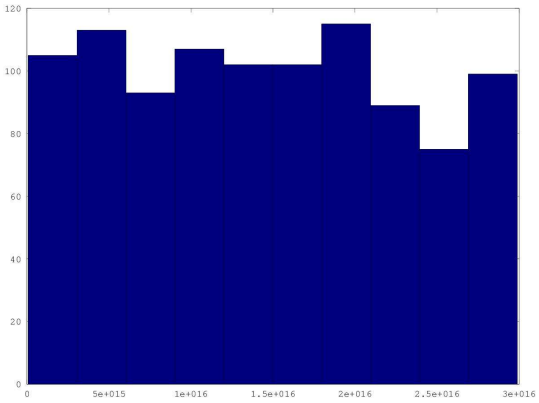


Imagen 6.2 Distribución uniforme – Radios.

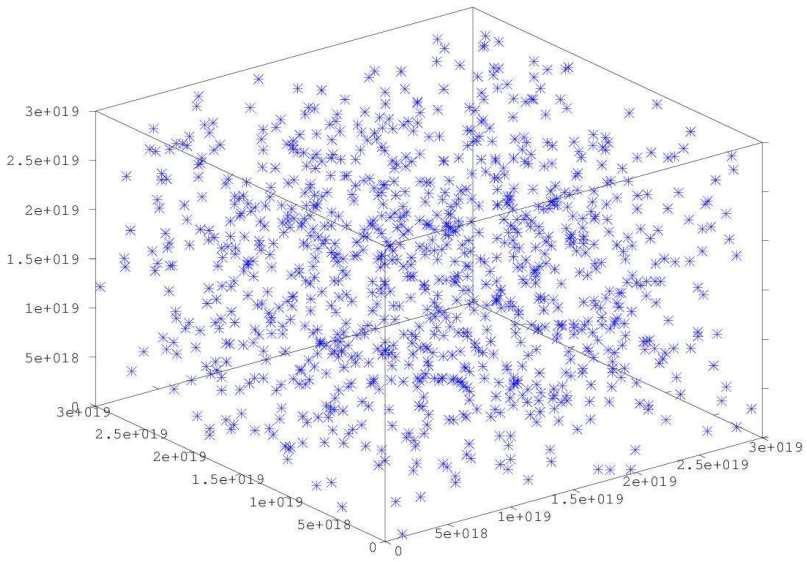


Imagen 6.3 Distribución uniforme – Posiciones.

El script de Octave se invoca en dos pasos: primero, la ejecución de un script de Powershell[8], y segundo, la ejecución del script en cuestión. Esta forma de ejecución no es necesaria, pero se ha implementado con la intención de que el script de Powershell actúe a modo de *switch*, permitiendo al programa principal elegir qué script de Octave lanzar y con qué parámetros de entrada.

Los nuevos valores generados no se pueden devolver directamente al programa principal, así que, son almacenados en un fichero CSV para que éste los lea y pueda iniciar la ejecución el algoritmo PP.

[8] La función system () permite la ejecución de un comando en el procesador de comandos del sistema. A través de ella, se puede lanzar el script de Powershell.

6.4.RESULTADOS OBTENIDOS

En primer lugar, se ha lanzado la simulación con las opciones de detección de colisiones y unión de planetas colisionados, así como la de división de los planetas por grupos, desactivadas para poder comparar la ganancia de cada una. Con todo desactivado, los planetas se comportan como se esperaba: a partir de su posición inicial, se ven atraídos hacia el centro, que es donde tiende a concentrarse la mayor parte de la masa, y pasadas unas iteraciones, algunos planetas terminan estabilizándose describiendo órbitas entorno a otros pocos planetas (ver imagen 6.4). Al igual que ocurre en el escenario del Sistema Solar, no todos logran estabilizarse, también existen algunos que se desplazan hacia el infinito, pero como se puede ver en las capturas, estos solo representan un pequeño porcentaje.

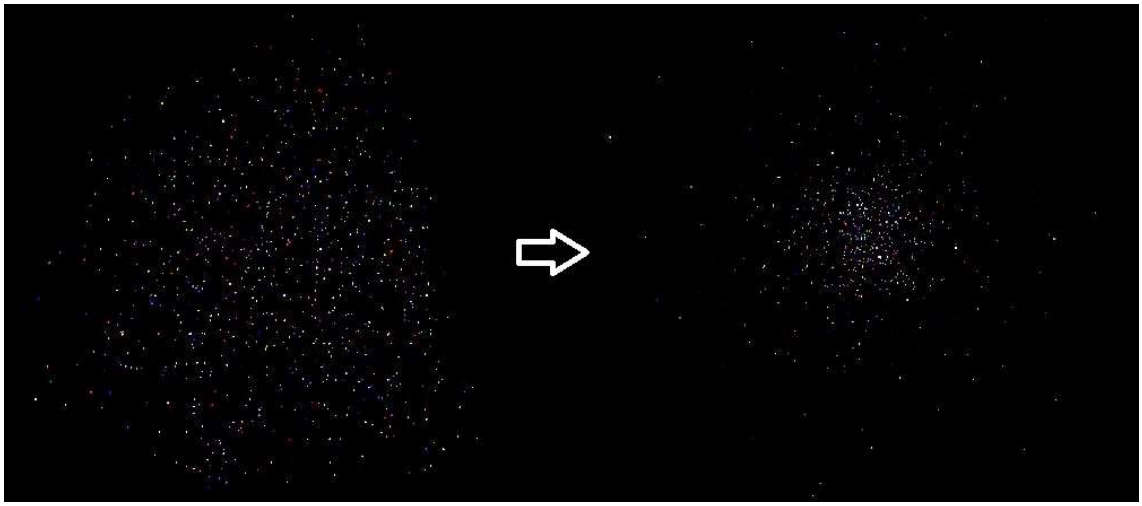


Imagen 6.4 Inicio y fin planetas aleatorios.

Al activar las opciones de colisión y unión, se ha visto que, sobre todo en las primeras iteraciones, cuando los planetas tienden a concentrarse en el centro del espacio, se pueden producir colisiones y el número de planetas en simulación decae. Pero por lo general, conforme pasa el tiempo, cada vez existen menos choques, así que esta optimización tampoco ha terminado siendo muy influyente en la ejecución del programa. Al activar la optimización de dividir los planetas en grupos en función de su posición, sí que se ha notado como se gana una pequeña mejora de rendimiento, entorno a unos 5 FPS, algo que no es muy relevante.

En conclusión, se puede decir que, al final, las mejoras implementadas no han conseguido la ganancia esperada posiblemente debido a que el número de objetos y el tamaño del universo recreado es pequeño (recuérdese que se está trabajando con una GPU con las características especificadas en el apartado 4.4). A pesar de ello, el algoritmo responde muy bien ante el enorme aumento en el número de planetas respecto al escenario anterior. Se han conseguido unos FPS en torno a 30 simulando 1000 planetas, es decir, el número máximo que la GPU utilizada es capaz de renderizar en tiempo real está en 1000 aproximadamente (ver imagen 6.5).

Imagen 6.5 Captura de la ejecución de la simulación aleatoria. Se muestran algunos datos del rendimiento como los FPS estabilizados en 33 FPS (derecha).

Para ponerlo en contexto y poder tener una mejor idea de la potencia computacional con la que se ha trabajado, el hecho de que el programa funcione a una velocidad de 30 FPS quiere decir que está generando un *frame* cada 30 ms, es decir, el programa realiza una iteración del *timestep loop* (por diferenciarla de otras iteraciones, se va a denominar como it_T) cada 0,03 segundos. En cada una de ellas, el programa mueve 1000 planetas, lo que implica que está calculando la fuerza neta en cada uno de ellos, es decir, está ejecutando el bucle que calcula la fuerza gravitatoria que ejercen todos los planetas sobre cada uno de los demás (recuérdese el capítulo 2). Si tenemos 1000 planetas, para calcular la fuerza sobre cada uno de ellos es necesario realizar 1000000 iteraciones del bucle que calcula las fuerzas (it_F). La duración del parpadeo del ojo humano está registrada en unos 0.3 o 0.4 segundos, 10 veces más que los segundos necesarios para realizar una it_T completa. En conclusión, el algoritmo es capaz de realizar 10 it_T cada una de ellas compuesta de 10^6 it_F, haciendo un total de 10^8 iteraciones en el tiempo en que dura un parpadeo.

El que se haya obtenido 30 FPS en la simulación, no significa que la GPU haya llegado a su límite, de hecho, como prueba final del proyecto, se ha incrementado dicho número hasta 2000 planetas y se han obtenido un rendimiento alrededor de los 10 FPS y una utilización que ronda el 99% como se puede comprobar en la imagen 6.6.

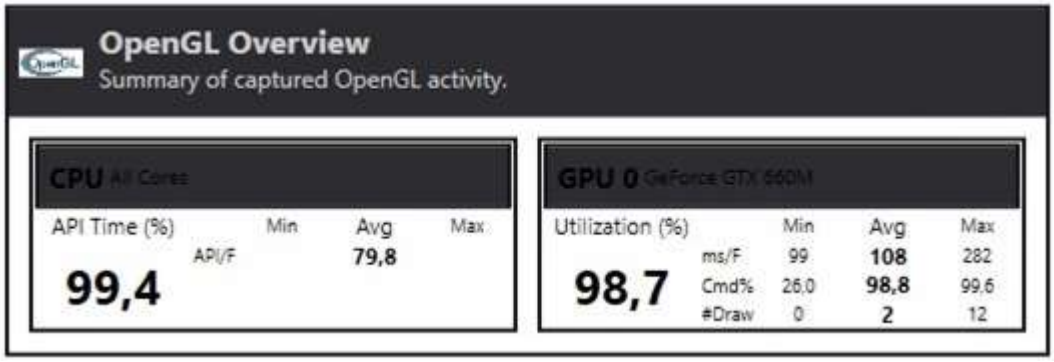


Imagen 6.6 Rendimiento sistema aleatorio.

Es interesante fijarse en el tiempo que la CPU destina a hacer llamadas a la API de OpenGL. Recordar que, en la simulación del Sistema Solar, este valor era de entorno al 25%, en cambio, ahora nos encontramos con casi el 100% a pesar de que la estructura del programa no se ha modificado, las funciones de OpenGL que son invocadas son prácticamente las mismas. El motivo de este aumento radica en el hecho de que, en esta simulación, la CPU tiene que mandar muchos más datos a la GPU.

Para finalizar, la imagen 6.7 muestra 2000 planetas formando un sistema estable.

Imagen 6.7 Captura final sistema aleatorio.

7. CONCLUSIÓN

Como se expone en el apartado de la introducción, el objetivo último del trabajo era el estudiar el comportamiento de una GPU trabajando con partículas. Se ha tomado como escenario para el desarrollo el problema de los N cuerpos por su carga tanto gráfica como computacional.

El algoritmo elegido para la simulación de dicho escenario ha sido el algoritmo "Partícula - Partícula" que, tal y como se ha expuesto al principio del documento, funciona bastante bien para un número de cuerpos pequeño, estableciendo el límite en torno a los 4000 cuerpos.

Partiendo del algoritmo básico, optimizado para una ejecución secuencial, se ha obtenido uno válido para ser ejecutado en paralelo, sin riesgo a que pueda generar problemas de concurrencia de ningún tipo. Además, se han mejorado las ecuaciones de movimiento (posición y velocidad) mediante el uso del integrador de *Leapfrog*, logrando más precisión en los cálculos.

Cabe resaltar de nuevo la elevada complejidad computacional que presenta el algoritmo "Partícula –Partícula" implementado. Para cada iteración del bucle principal, tiene que calcular todas las fuerzas de cada uno de los cuerpos de la simulación con todos los demás. Se está hablando de un algoritmo de orden de complejidad cuadrático (al duplicar el número de cuerpos involucrados en los cálculos, se cuadruplica el tiempo de ejecución). Con la intención de intentar mejorar esta situación, se han implementado optimizaciones, como la de restringir el cálculo de fuerzas de un cuerpo a los cuerpos cercanos.

Se han desarrollado dos simulaciones: la simulación del Sistema Solar y la simulación aleatoria. En la primera, el objetivo era integrar ese algoritmo en un programa de OpenGL y poder visualizar, en un mundo 3D, los resultados que se iban generando. Tras observar que todo funcionaba de forma correcta, y que las trayectorias de los planetas y satélites involucrados en la simulación eran, si bien no exactas, bastante fieles a la realidad, se procedió a iniciar la segunda simulación, cuyo objetivo era incrementar el número de planetas hasta que la GPU no pudiera mantener unos FPS mínimos, estableciendo el límite alrededor de 15 FPS.

Una vez estudiados los resultados obtenidos en cada apartado, se puede afirmar que, dentro de las limitaciones del proyecto, el objetivo último de todo el trabajo se ha cumplido. Utilizando una GPU modelo Nvidia GeForce GTX 660M, se ha logrado crear un escenario con un máximo de 2000 cuerpos, manteniendo unos FPS alrededor de 10.

7.1. DIAGRAMA TEMPORAL

La realización del diagrama temporal supone un problema. Ha sido un proyecto muy entrecortado por motivos laborales y personales, por lo que establecer unos límites en horas y fechas supone una tarea complicada.

De todas formas, aunque las fechas pueden no ser las exactas, la distribución de la carga sí que es fiel a la realidad. La imagen 7.1 muestra el diagrama de Gantt de este proyecto.

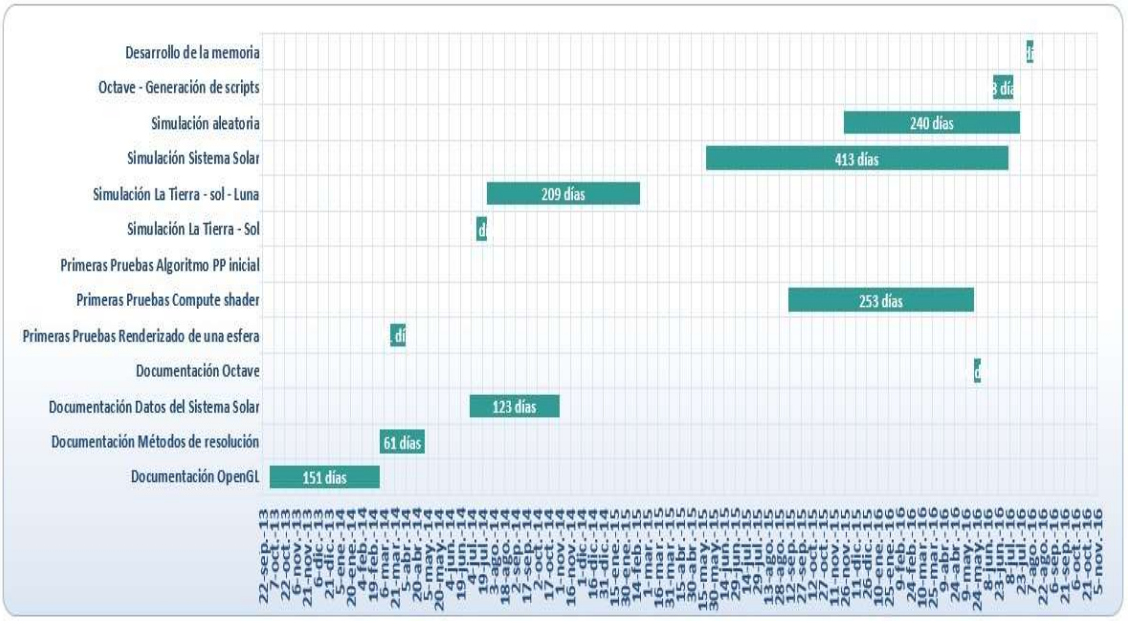


Imagen 7.1 Diagrama de Gantt.

7.2. POSIBLES AMPLIACIONES

Como posibles mejores al trabajo realizado, se podrían mejorar, en primer lugar, las fórmulas de movimiento (posición y velocidad). Se ha visto que los cálculos no son lo suficientemente exactos y esto repercutía en que algunos cuerpos del Sistema Solar se perdían en el infinito.

Por otro lado, los resultados obtenidos son específicos de la GPU Nvidia GeForce GTX 660M como ya se ha comentado. Sería recomendable apuntar como una posible ampliación, una comparación entre esta GPU y una actual, como la GTX 1060. Es de esperar que el tiempo de ejecución mejore, y en consecuencia, que el número de cuerpos N simulados se incremente.

Como aplicación directa de este proyecto, se sitúan las simulaciones en campos como la astronomía, la mecánica de fluidos o la electroestática entre otros. Otra posible aplicación son los videojuegos. Algunos juegos cuentan con explosiones, sistemas de partículas o directamente, la base del juego es la atracción gravitatoria entre los objetos, para estos aspectos se podría aplicar el trabajo realizado también de forma directa.

7.3. OPINIÓN PERSONAL

Ha sido un proyecto muy rico en conocimientos. Se ha tocado prácticamente todo lo que se podía tocar con OpenGL, permitiéndome aprender muchos aspectos que en la carrera no me han presentado (o no han hecho demasiado hincapié) y que, de no ser por esto, no

hubiera aprendido. Además, ha sido un proyecto muy amplio como se puede apreciar, en el sentido de que abarca una gran variedad de tecnologías y programas aparte de OpenGL (Excel, Octave, Powershell).

Hay que destacar la dificultad de ciertas partes. En primer lugar, programar en GLSL (lenguaje utilizado en los *shaders* de OpenGL) es muy complicado. Hay que ver la GPU como una caja negra, puedes conocer lo que entra y lo que sale, pero no hay una forma sencilla de ver lo que ocurre por dentro. En el momento en el que algo falla dentro de un *shader*, descubrir lo que ocurre y arreglarlo es una tarea compleja. La información que recibes sobre el error es mínima, insuficiente, en la mayoría de las ocasiones, para hacerte una idea de lo que está fallando. Tampoco se dispone de un *debug* con el que ir ejecutándolo instrucción a instrucción hasta dar con el fallo. También hay que tener en cuenta, que la situación laboral en la que estaba no me permitía dedicar las horas que el proyecto necesitaba para llevarlo al día. Todo esto ha influido en su duración final.

A pesar de todos los problemas surgidos y la duración del trabajo, ha merecido la pena todo el esfuerzo. Es un trabajo que visualmente es bonito. Las simulaciones en 3D que se crean (el Sistema Solar dando vueltas, el desplazamiento de los planetas al ser atraídos por todos los demás, la agrupación de los mismos en torno a unos pocos planetas con masas, en comparación, más grandes) no se hubieran conseguido con otro proyecto. Creo que esa es la principal característica de un programa de gráficos, no solo construyes un programa que funciona, sino que, además, puedes verlo con tus propios ojos.

8. BIBLIOGRAFÍA

[1] Eastwood, R. W. (1988). Computer Simulation using Particles. New York: Taylor & Francis Group, LLC.

[2] Graham Sellers & Richard S. Wright, J. &. (2014). *OpenGL Superbible. Comprehensive Tutorial and Reference (6º ed.).* Crawfordsville, Indiana: Pearson Education, Inc.

[3] *Leapfrog integration*. (septiembre de 2016). Obtenido de https://en.wikipedia.org/wiki/Leapfrog_integration

[4] Nguyen, H. (2007). *GPU Gems 3.* USA: Addison-Wesley Professional.

9.APENDICES

APENDICE A: OBTENCIÓN DE LAS ECUACIONES DE MOVIMIENTO

Sabiendo los ápsides de la trayectoria de un planeta y su inclinación respecto de la eclíptica, es posible calcular cualquier posición de su recorrido. En este proyecto, se van a calcular las posiciones en uno de sus ápsides ya que los cálculos se simplifican, en concreto, se ha elegido el apoápside.

Dado que las trayectorias que se describen son elípticas, se van a utilizar las fórmulas matemáticas de la elipse. Una elipse es una curva plana y cerrada, simétrica respecto a dos ejes perpendiculares entre sí: el semieje mayor y el semieje menor.

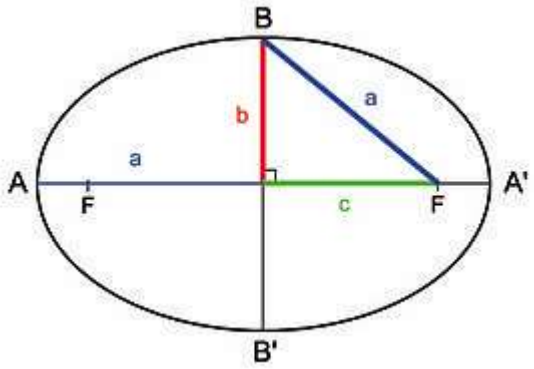


Imagen A.1 Elipse.

Las fórmulas que definen la posición y la velocidad de un objeto que describe una trayectoria elíptica son las siguientes:

$$\frac{(x - x_0)^2}{a^2} + \frac{(y - y_0)^2}{b^2} = 1 \qquad \text{(A,1)}$$

$$v = \sqrt{2\mu\left(\frac{1}{r} - \frac{1}{2a}\right)} + v_{Eje} \qquad \text{(A,2)}$$

$$\mu = G * M_{Eje} \qquad \text{(A,3)}$$

G: $constante\ de\ gravitacion\ universal = 6{,}67428 * 10^{-11}\frac{Nm^2}{Kg^2}$

M_{Eje}: $Masa\ del\ planeta\ eje$

r: Distancia entre el planeta en cuestión

(x_0, y_0): Centro de la elipse

Si se contrasta la elipse con la trayectoria de un planeta, se puede ver que las siguientes fórmulas son válidas:

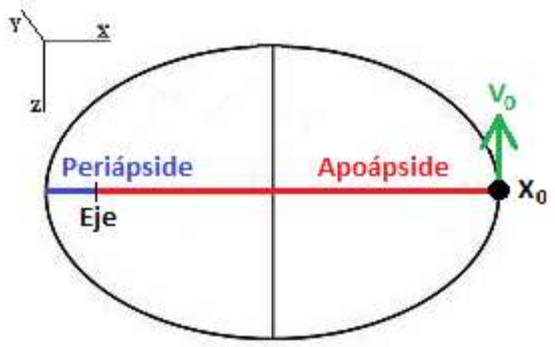


Imagen A.2 Trayectoria de un planeta.

$$a = \frac{Periápside + Apoápside}{2} \tag{A,4}$$

$$V_0 = \sqrt{2GM_{Eje}\left(\frac{1}{Apoápside} - \frac{1}{2a}\right) + V_{Eje}} \tag{A,5}$$

$$\overrightarrow{V_0} = (0, 0, -V_0) \tag{A,6}$$

$$X_{Apoápside} = (Apoápside, 0, 0) \tag{A,7}$$

Una vez conseguida la posición en el apoápside del planeta en cuestión, se procede a calcular su posición inicial utilizando para ello la inclinación de su órbita valiéndonos de las reglas trigonométricas.

$$X_0 = (Apoápside * \cos\varphi\,, Apoápside * \sin\varphi\,, 0) + X_{Eje} \tag{A,8}$$

ϕ: inclinación orbital

APENDICE B: INTEGRADOR DE LEAPFROG

En matemáticas, el método del salto de rana o *Leapfrog Integration* [5] es un método simple para la resolución numérica de ecuaciones diferenciales de la forma:

$$\ddot{x} = F(x) \tag{B,1}$$

O bien, expresado de otra manera, ecuaciones del tipo:

$$\dot{v} = F(x), \dot{x} \equiv v \tag{B,2}$$

El método consiste en actualizar las posiciones $x(t)$ y velocidades $v(t) = \dot{x}(t)$ en etapas intercaladas, es decir, la posición se actualiza en el paso *i* mientras que la velocidad lo hará en el $i + \frac{1}{2}$. De esta forma, el sistema quedaría expresado como sigue:

$$x_i = x_{i-1} + v_{i-1/2} DT \tag{B,3}$$

$$a_i = F(x_i) \tag{B,4}$$

$$v_{i+1/2} = v_{i-1/2} + a_i DT \tag{B,5}$$

x_i: *Posición en el paso i*
a_i: *Aceleración en el paso i*
$v_{i+1/2}$: *Velocidad en el paso* $i + 1/2$
DT: Tiempo que transcurre entre cada paso

El hecho de que la posición y la velocidad no se actualicen en el mismo paso añade una dificultad a la hora de resolver el sistema. Se puede aplicar un cambio para que ambos se actualicen a la vez siempre que el *DT* se mantenga constante. Con dicho cambio, el sistema quedaría:

$$x_{i+1} = x_i + v_i DT + \frac{1}{2} a_i DT^2 \tag{B,6}$$

$$v_{i+1} = v_i + \frac{1}{2}(a_i + a_{i+1}) DT \tag{B,7}$$

El método *Leapfrog Integration* presenta importantes ventajas con respecto a otros métodos de resolución de ecuaciones diferenciales, como podrían ser los métodos de Euler o Runge-Kutta, cuando es aplicado a problemas mecánicos.

El primero es que es reversible en el tiempo. El método permite avanzar *n* pasos hacia adelante, retroceder otros *n* pasos y llegar al mismo punto inicial del sistema. El segundo es su naturaleza simpléctica, la cual implica que conserva la energía de los sistemas dinámicos, característica especialmente importante a la hora de simular órbitas dinámicas. Es por esto último por lo que uno de los usos más importantes de este método sea en las simulaciones gravitacionales.

APENDICE C: DEPTH TEST

El *depth test* es una operación que se procesa por cada fragmento que el *fragment shader* emite. Es un test cuya finalidad es la de, dada una serie de fragmentos que quieren dibujarse sobre el mismo pixel, cuál de ellos es el que se debe terminar dibujando. Para ello, utiliza un recurso llamado Z-Buffer.

El Z-Buffer es un buffer que almacena información relativa a la profundidad de cada fragmento. Este buffer se crea automáticamente al crear el contexto de ventanas y se suelen almacenar en registros de 24 bits.

El *depth test* consiste en una comparación entre el valor de profundidad (Z) del fragmento actual y el valor que existe en el Z-Buffer. Si este fragmento falla el test, es descartado y no llegará a dibujarse, por el contrario, si lo pasa, el buffer se actualizará con el valor del fragmento en cuestión. De esta forma, el programa es capaz de determinar qué fragmento debe mostrar en cada caso.

Cuando el desarrollador establece una perspectiva, establece un volumen de visualización delimitado por un plano más cercano a la cámara, denominado zNear, y otro lejano, zFar (ver imagen C.1). Dependiendo de la diferencia que exista entre ambos, se pueden dar casos en los que este *depth test* no funcione correctamente debido a un fallo en la precisión de los cálculos.

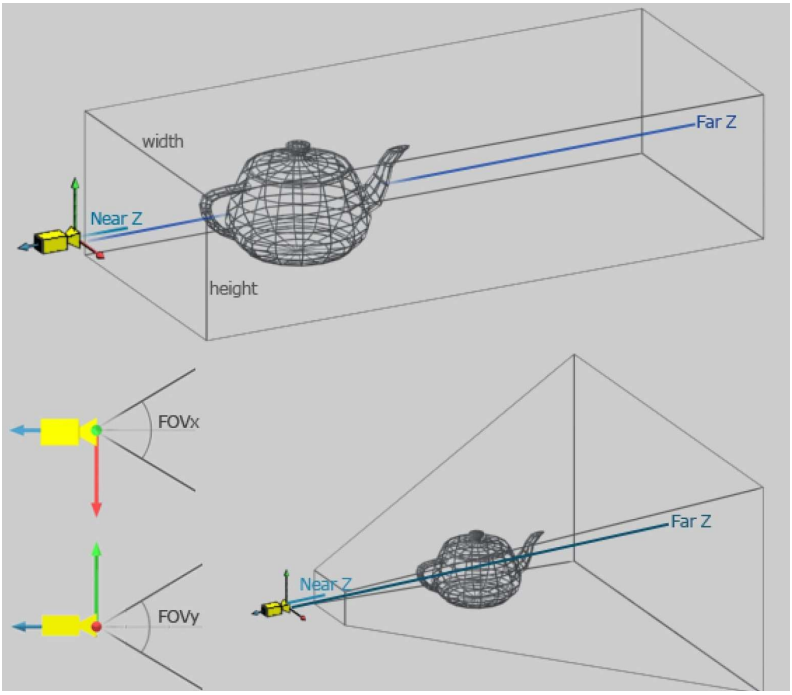


Imagen C.91.1 Volumen de visualización con sus límites zNear y zFar.

Los valores de z van a estar representados entre zNear y zFar, Después de aplicar la transformación de perspectiva, el nuevo valor de z, ya normalizado, está definido por[9]:

[9] Fórmulas Z-Buffer: https://en.wikipedia.org/wiki/Z-buffering, a fecha de septiembre de 2016.

$$z' = \frac{zFar + zNear}{2 * (zFar - zNear)} + \frac{1}{z}\left(\frac{-zFar * zNear}{zFar - zNear}\right) + \frac{1}{2} \qquad \text{(C,1)}$$

Multiplicando ahora por $S = 2^d - 1$, donde d es el número de bits que utiliza el Z-buffer, que por lo general es de 24 bits, y redondeando el resultado, se obtiene la ecuación:

$$z' = floor((2^d - 1) * \left(\frac{zFar + zNear}{2 * (zFar - zNear)} + \frac{1}{z}(\frac{-zFar * zNear}{zFar - zNear})\right) + \frac{1}{2}) \qquad \text{(C,2)}$$

A partir de esta fórmula, se puede obtener la precisión del Z-buffer, simplemente invirtiendo y derivando, obteniendo:

$$z = \frac{-S * zFar * zNear}{z' * (zFar - zNear) - zFar * S} \qquad \text{(C,3)}$$

$$S = 2^d - 1$$

A través de ella, se establece una precisión no lineal del Z-buffer. Entorno al plano más próximo a la cámara, la precisión es muy alta, en cambio, cerca del plano lejano, la precisión es muy pequeña. Esta precisión mantiene una relación directa con la ratio $zNear/_{zFar}$, de forma que, cuanto más pequeño sea su valor, más pequeña será la precisión a lo lejos. De hecho, el establecer un zNear muy cercano a la cámara, es un error muy común que ocasiona un renderizado con errores.

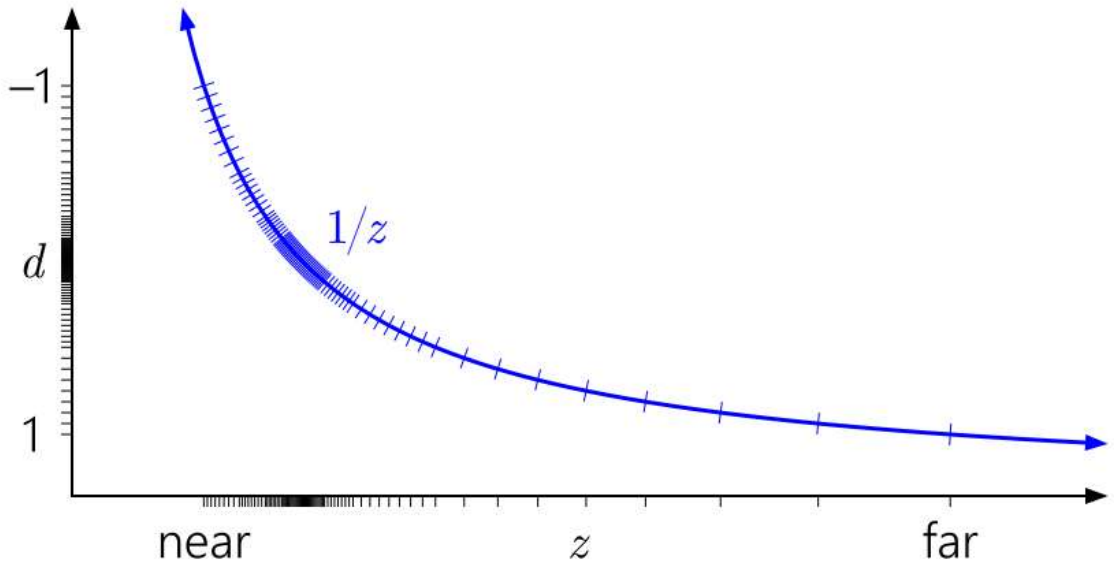


Imagen C.2 Precisión Z-buffer.

En este proyecto se ha sufrido este error. El plano cercano está situado muy cerca de la cámara, y el plano lejano se sitúa en un valor de z muy grande debido a las distancias entre los planetas que se están tratando, por lo que, a pesar de que en los planetas muy cercanos no se aprecia problema alguno, en los más lejanos se puede observar como aparecen y desaparecen.

La solución que se ha establecido en este proyecto ha sido la de desactivar el *depth test*, priorizando la visibilidad de los cálculos matemáticos ante el renderizado exacto.

APENDICE D: PIPELINE GRÁFICA

En este apéndice, se realiza un recorrido por las *pipelines* que existen (*rendering pipeline* y *compute pipeline*), así como una explicación más detallada de las principales etapas de las mismas.

RENDERING PIPELINE

La imagen D.1 muestra una *pipeline* de renderizado simplificada.

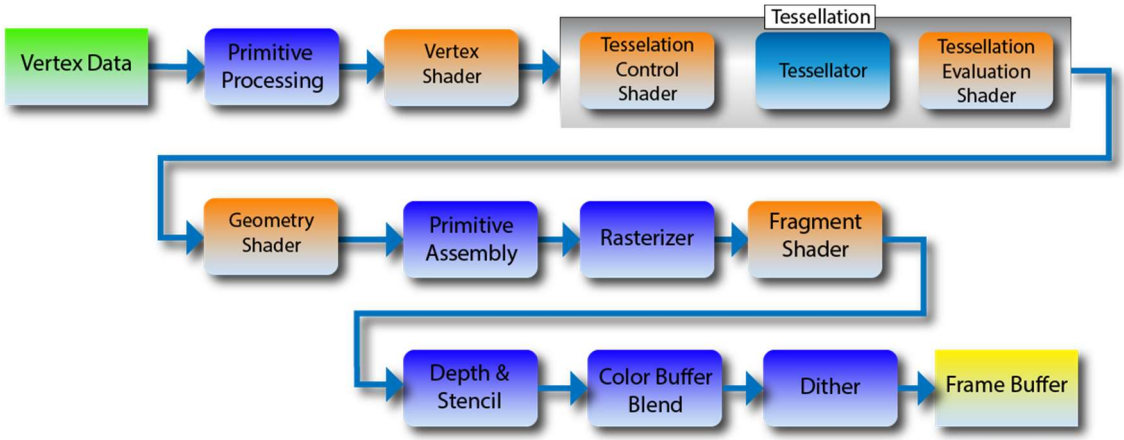


Imagen D.1 Pipeline de renderizado.

Los bloques que se muestran en naranja indican las etapas que son programables. Éstas ejecutan el código que el programador les suministra a través de los shaders. Las etapas que se muestran en azul indican funciones fijas. En la práctica, todas o la mayoría de estas últimas etapas terminan pudiendo ser implementadas en un *shader*, la diferencia es que no es el programador quien lo suministra, sino más bien el fabricante de la tarjeta gráfica a través de *drivers*, *firmware* u otro tipo de software similar.

- *vertex fetching*:

La *pipeline* empieza con una etapa no programable. Esta etapa dicta cómo serán introducidos los vértices a la *pipeline* y los prepara para mandárselos a la primera etapa programable, el *vertex shader*.

A pesar de ser una función fija y, por tanto, no programable, existen funciones[10] que el programador pude utilizar para elegir ciertos parámetros.

- *Vertex shading*:

El *vertex shader* es la primera etapa programable en la *pipeline* y se distingue de los demás *shaders* en que es el único que debe estar presente obligatoriamente, aunque si

[10] La API de OpenGL ofrece, por ejemplo, las familias de funciones glVertexAttrib*() y glDraw*() que el programador invoca para establecer en donde están almacenados los datos, en qué modo (punto, triángulos o *patches* (polígonos de cualquier número de vértices)), etc.

no está presenta el *fragment shader*, no se podrá ver nada en la pantalla. Este *shader* se alimenta de los datos que genera la *fixed-function vertex fetching*.

Recibe como entrada un flujo de vértices y es el encargado de manejar el procesamiento de éstos de forma individual. Las tareas que suele desempeñar son el cálculo de la posición de los vértices, el cálculo de la iluminación *per-vertex* y cualquier otro cálculo necesario para las etapas posteriores. Como salida, genera un vértice por cada uno que él recibe y los ensambla formando primitivas geométricas como el punto, segmentos de líneas o polígonos (ver imagen D.2).

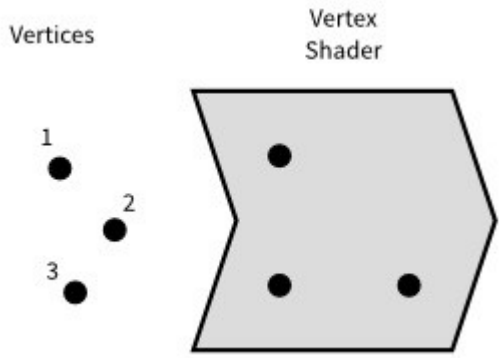


Imagen D.2 Entrada y salida del vertex shader.

- <u>*Tessellation*</u>:

La teselación es el proceso de romper primitivas de alto orden, denominadas *patches*, en muchas primitivas más pequeñas y simples (ver imagen D.3). Estos *patches* están formados por varios vértices denominados puntos de control. Por lo general, suelen estar formados por tres o cuatro.

Este proceso es opcional y consta de tres partes: *tessellation control shader* (TCS), *tessellation engine*, la cual no es programable, y *tessellation evaluation shader* (TES).

El TCS es una etapa programable, situada inmediatamente después del *vertex shader*, cuya finalidad es la de establecer el nivel de teselación que será utilizado por el *tessellation engine* para generar los nuevos vértices.

El TES es la última etapa programable de este proceso. Recoge todos los vértices generados por el *tessellation engine*, así como todos los datos que le ha enviado el TCS, y se encarga de dar los valores finales a esos vértices, como, por ejemplo, su posición.

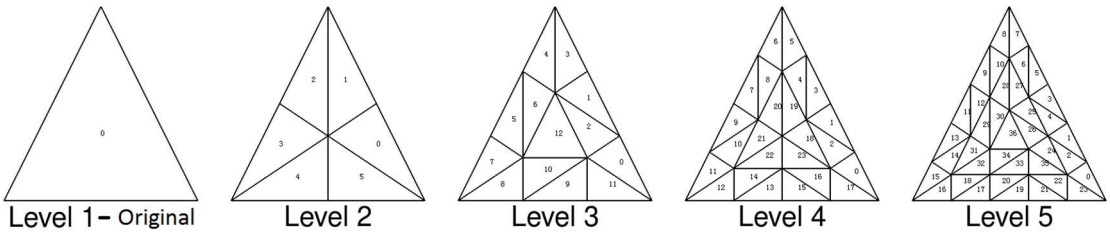


Imagen D.3 Resultado del proceso de teselación en diferentes niveles.

- *Geometry shading*:

El *geometry shader* es también una etapa en la que se pueden modificar la geometría de las primitivas generadas por el *vertex shader* o por el proceso de teselación.

Su función es similar a la de *tessellation*, genera múltiples primitivas a partir de una inicial. La principal diferencia entre ambos procesos reside en que la teselación implementa una subdivisión recursiva automática en función de las opciones que se hayan elegido en el TCS, en cambio, el *geometry shader* puede emitir o eliminar vértices a deseo del programador.

La imagen D.4 muestra algunos ejemplos de lo que se puede lograr usando el geometry shader.

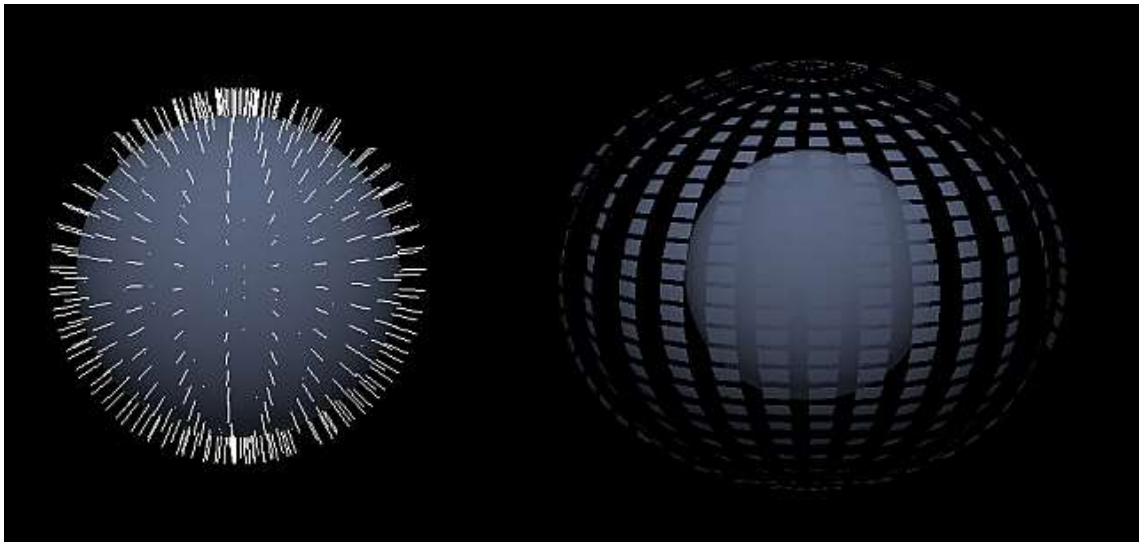


Imagen D.4 Tomando como base una esfera, se utiliza el geometry shader para dibujar los vectores normales (izquierda) y para generar caras nuevas y desplazarlas en la dirección de su vector normal (derecha).

- *Primitive assembly*, *clipping*, *culling* y *rasterización*:

En este punto se pasa a ejecutar unas etapas no programables de la *pipeline* que, partiendo de la representación en vértices de nuestra escena conseguida en etapas anteriores, realizan un conjunto de tareas para obtener una serie de pixeles que necesitan ser coloreados y pintados en la pantalla.

La primera de estas tareas es el ensamblado de primitivas. En esta etapa se agrupan los vértices en líneas o triángulos (también ocurre si la salida deseada son puntos, pero ese caso es trivial) (ver imagen D.5).

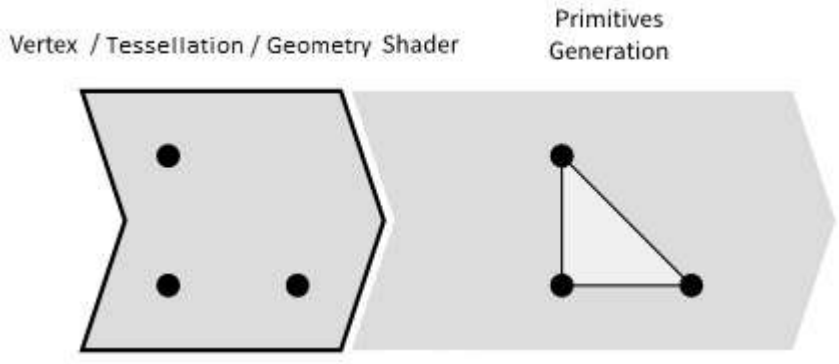


Imagen D.5 Primitive assembly.

Una vez agrupados en primitivas, éstas inician el proceso de *clipping*. El *clipping* es el proceso de determinar cuáles de las primitivas están completa o parcialmente dentro del *viewport* o región que se va a mostrar en pantalla y, por tanto, van a llegar a las etapas posteriores de la pipeline, y cuáles van a situarse fuera y no serán mostradas. En ocasiones, solo una porción de la primitiva se sitúa dentro del *viewport*, en este caso, se va a recortar de tal forma que la nueva primitiva este totalmente dentro del *viewport* (ver imagen D.6).

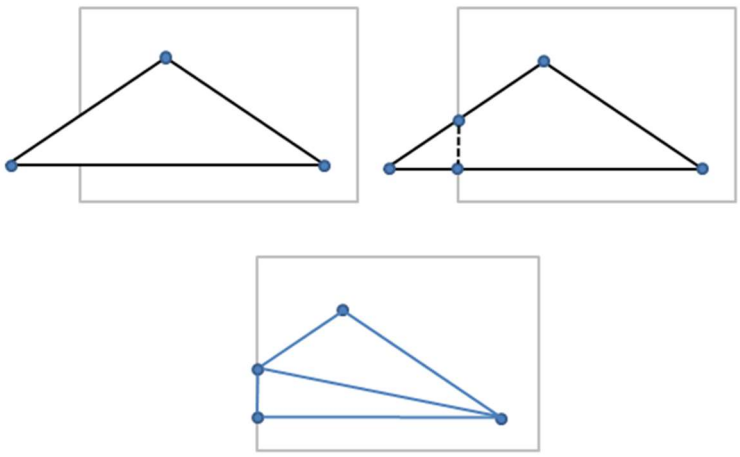


Imagen D.6 Triángulo recortado (clipped) por estar parcialmente fuera del viewport.

Opcionalmente, antes de pasar a la siguiente etapa, las primitivas pueden pasar por una etapa denominada *culling*. En ella, se determina si la primitiva está mirando a la cámara o no, es decir, si el vector normal de la primitiva apunta en dirección a la cámara. En caso de que no esté mirando a la cámara, la primitiva será descartada con el *culling* (ver imagen D.7).

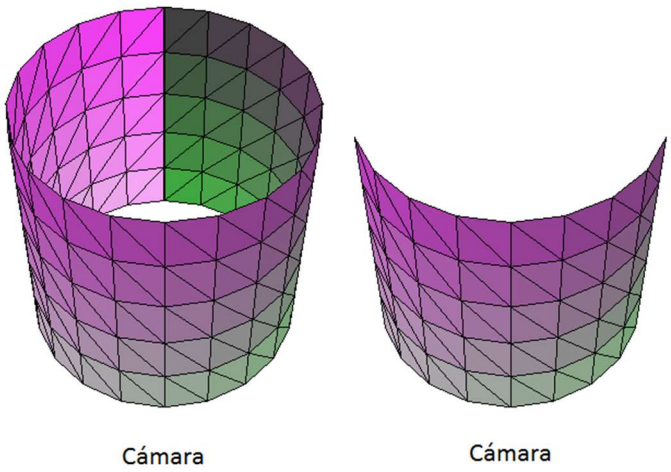


Imagen D.7 Efecto del backface culling, aquellas caras que no están mirando a la cámara se descartan (izquierda - original, derecha - culling aplicado).

Por último, las primitivas llegan a la etapa de rasterización. En este proceso se determina qué conjunto de pixeles se encuentra cubierto por una primitiva geométrica (ver imagen D.8).

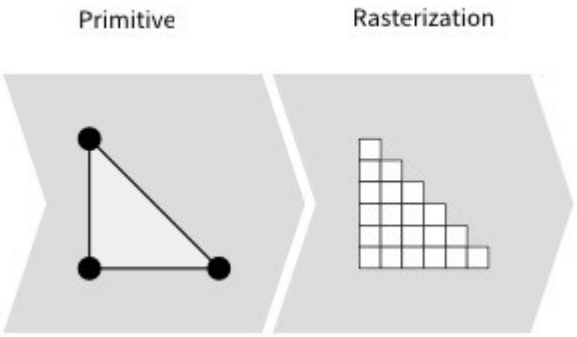


Imagen D.8 La primitiva está formada por uno o más vértices que no están alineados con la malla de pixeles. Con la rasterización se obtiene un fragmento que sí que está alineado.

- *Fragment shading*:

El *fragment shader* es la última etapa programable. Este *shader* recibe como entrada un conjunto de pixeles o fragmentos a los que hay que especificar un color para poderlos mostrar en pantalla (ver imagen D.9). Es en esta etapa donde se llevan a cabo todos los cálculos que van a determinar dicho color. El proceso puede ser tan sencillo como especificar un color sólido para todo fragmento que entre, o tan complejo como el tener que calcular las características de la iluminación o la aplicación de texturas o materiales.

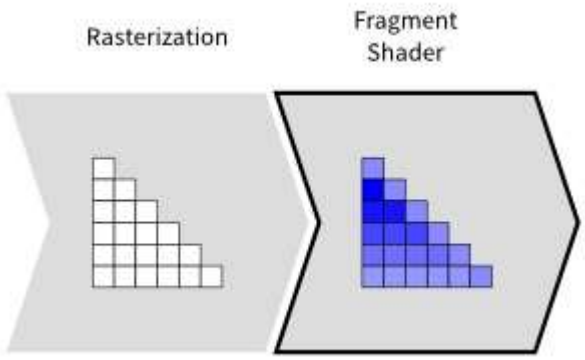


Imagen D.9 Entrada y salida del fragment shader.

- *Depth*, *stencil* and *blending*:

Para finalizar el proceso de renderizado, los fragmentos que se obtienen del *fragment shader* van a pasar una serie de test y transformaciones antes de ser almacenados en el *framebuffer*[11] para ser mostrados por pantalla.

En primer lugar, se les aplica el llamado *depth test*. Existe la posibilidad de que dos o más fragmentos se quieran dibujar en la misma región de la pantalla, en tal caso, hay que determinar cuál de ellos será el que finalmente se muestre. Para ello está el depth *test*. En este, se comparan las coordenadas Z del espacio de coordenadas de la pantalla de cada fragmento y en base a ella se selecciona uno u otro. Es el programador el que puede elegir la regla mediante la que se compara, por defecto, pasan el test aquellos fragmentos más próximos a la cámara (ver imagen D.10).

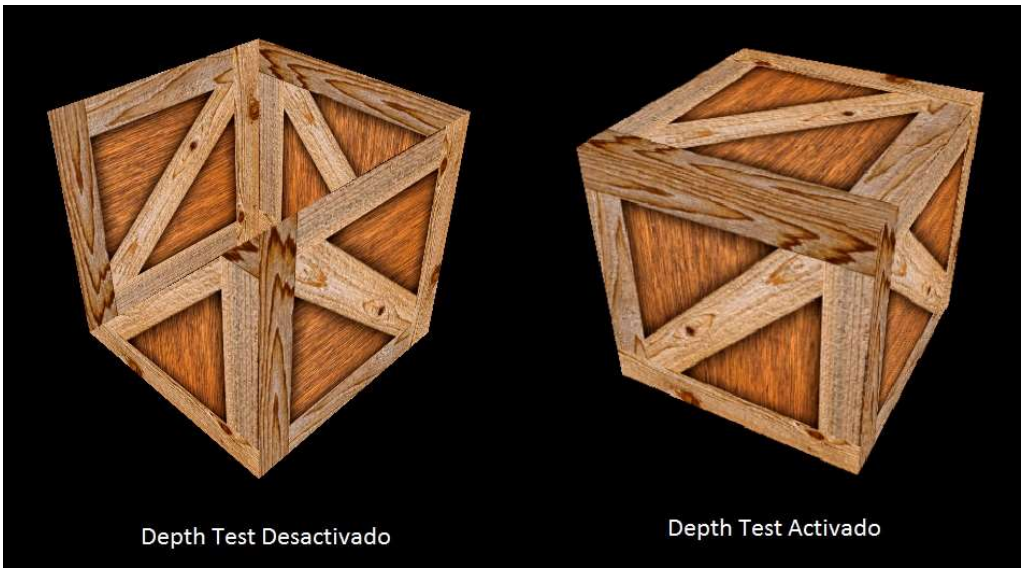


Imagen D.10 Con el depth test desactivado (izquierda), las caras del cubo se dibujan conforme van siendo tratadas, sobrescribiendo las caras anteriores si se superponen. Con el depth test activado (derecha), el cubo se dibuja correctamente, las caras que están siendo tapadas por otras caras más cercanas, no se visualizan.

[11] El *framebuffer* es el buffer donde se almacena temporalmente una imagen (*frame*) a la espera de ser enviada al monitor o a un dispositivo.

Otro test que también puede ser aplicado es el *stencil test*. Su funcionamiento es similar al del test de profundidad, pero, en esta ocasión, los fragmentos se comparan contra los valores que residen en el *stencil buffer,* previamente establecidos por el usuario. Un ejemplo podría ser el utilizar una imagen de máscara, con todo negro excepto una zona, y cargarla en el *stencil buffer*. Bajo este caso, solo aquellos pixeles que residen en la zona que no es negra pasarán el test y se mostrarán en la pantalla, como se puede ver en la imagen D.11.

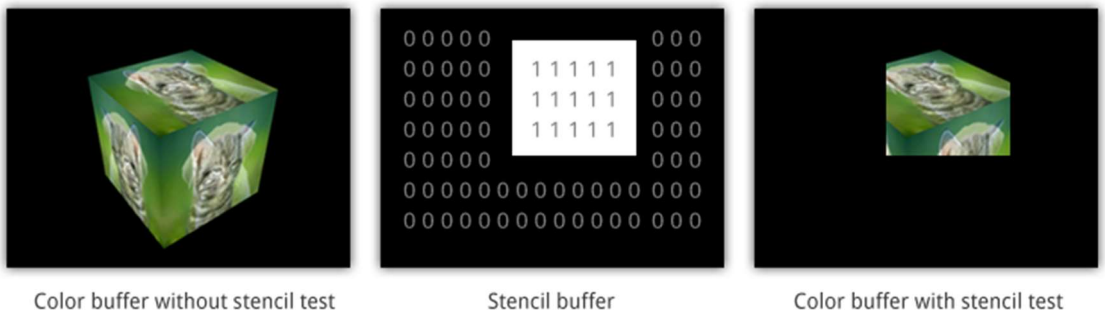


Imagen D.11 Ejemplo de aplicación del stencil test.

Por último, entra en juego el proceso de *blending*, que determina el color final que se va a mostrar utilizando para ello el color especificado en el *fragment shader* del pixel que se está procesando en ese momento y el color que ya había almacenado en el *framebuffer*. Dependiendo del alfa de cada color, se mezclarán ambos para obtener el color final de dicho pixel (ver imagen 4.12).

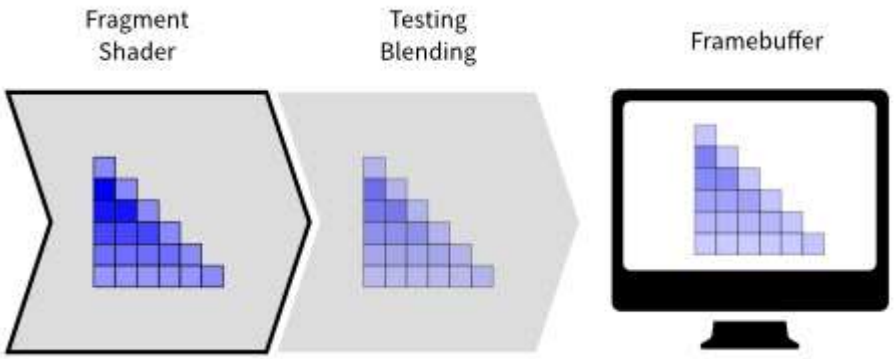


Imagen D.12 Salida del fragment shader, aplicación del blending, almacenamiento en el framebuffer y renderización final en la pantalla.

COMPUTE PIPELINE

- *Compute shader*:

Compute *shader* es el procedimiento que presentan las GPUs actuales para hacer uso de todo su poder computacional al programador. A diferencia de las etapas de la *pipeline* de renderizado, el compute *shader* puede ser considerado en sí mismo como una *pipeline* de una única etapa (ver imagen D.13).

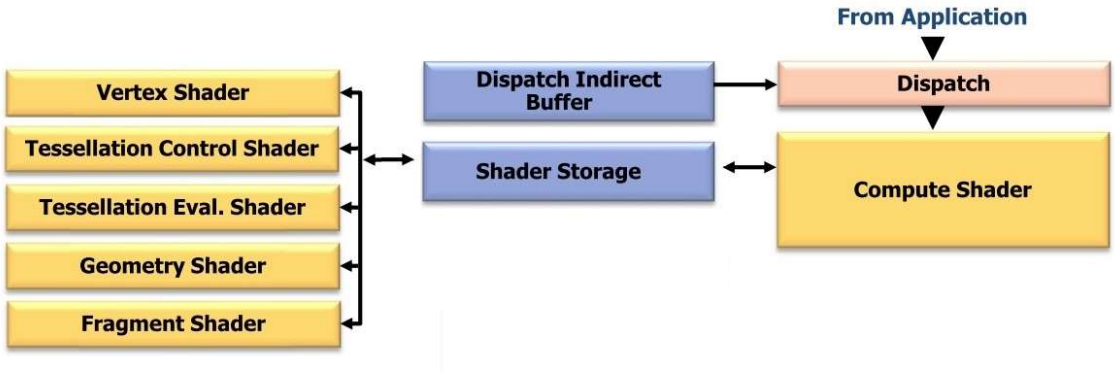


Imagen D.13 Compute shader (derecha), buffer de la GPU (centro). Como se puede ver, el compute shader puede comunicarse con todas las etapas programables de la rendering pipeline.

El *shader* no tiene ninguna entrada ni salida establecida. Toma como entrada los datos que el programador previamente ha introducido en el *shader storage buffer*, y es el programador el que debe explícitamente escribir los resultados en ese buffer antes de finalizar el *shader*. Cabe destacar que la memoria es compartida entre el compute *shader* y toda la *rendering pipeline*, por lo tanto, todos los resultados del *compute shader* pueden ser utilizados como entrada en una etapa programable.

Cada *compute shader* trabaja en una única unidad de trabajo llamada *work ítem*. Éstos están agrupados en grupos denominados *local workgroups* (ver imagen D.14). Cuando se invoca el *shader*, el programador es quien indica la cantidad de estos *workgroups* que va a utilizar. Cada *shader* se ejecuta en paralelo, si a esto unimos el gran poder computacional de la GPU, la *compute pipeline* nos presenta un entorno ideal para realizar cualquier tipo de cálculo pesado como el cálculo físico, la inteligencia artificial o el tratamiento de imágenes.

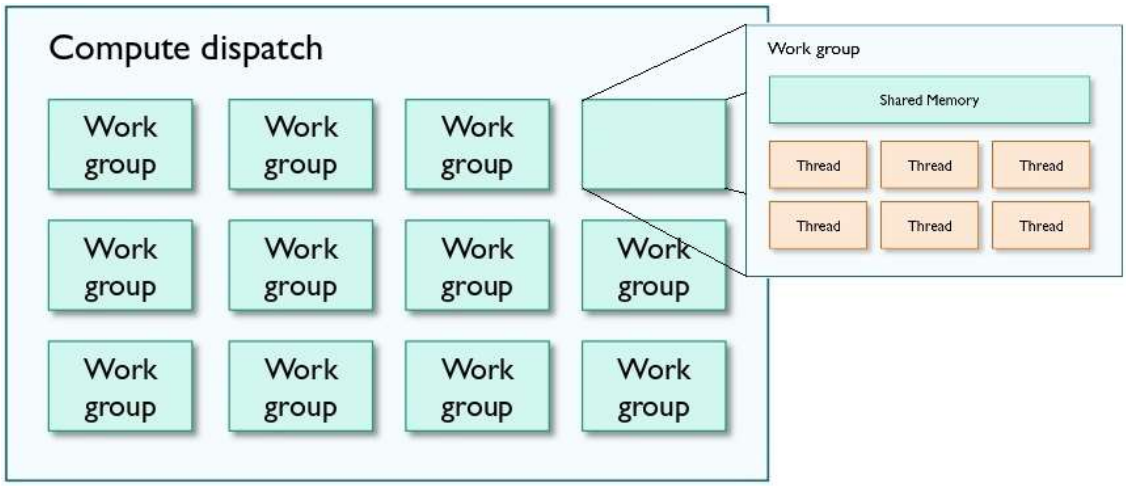


Imagen D.14 Distribución de los work items en el compute shader.