CEIG'09, San Sebastián, Sept. 9-11 (2009)
C. Andújar and J. LLuch (Editors)

Tracking 3D en GPU Basado en el Filtro de Partículas

I. Eskudero[2], J.Sánchez[1], C. Buchart[1], A. García-Alonso[2] y D. Borro[1]

[1]CEIT, Centro de Estudios e Investigaciones Técnicas de Guipúzcoa, San Sebastián
[2]EHU, Universidad del País Vasco, San Sebastián

Abstract

El tracking 3D basado en imágenes se resuelve habitualmente usando restricciones geométricas o mediante algoritmos estocásticos basados en filtros de estados. La primera opción es rápida pero poco robusta. La segunda es robusta pero menos eficiente. En este trabajo se mejora un método de tracking 3D estocástico basado en el filtro de partículas y se adapta sobre la GPU consiguiendo funcionar en tiempo real. Además, se demuestra experimentalmente su validez utilizando secuencias reales de vídeo.

1. Introducción

Un tracker es un rastreador, creado como un conjunto de software, que permitirá seguir al objeto que se seleccione en pantalla. Este tipo de técnicas resultan de interés en campos como la cirugía, procesos industriales, estudios ambientales o entretenimiento. Las aplicaciones van desde añadir objetos virtuales a la escena hasta la asistencia en operaciones de ensamblaje y mantenimiento. En este trabajo se implementa un tracker 3D basado en el filtro de partículas en GPU para así poder conseguir mayor *framerate*. Se ha adecuado el algoritmo para poder aprovechar las capacidades de la GPU. Se saca partido a la gran potencia de cálculo de las GPU para aplicaciones no relacionadas con los gráficos, en lo que recientemente se viene a llamar GPGPU, o GPU de propósito general (*General Purpose GPU*).

La utilización de lenguajes de alto nivel para la GPU ha facilitado enormemente la tarea de programar para ellas, por lo que cada vez son más los proyectos dedicados a sacarle rendimiento, como es el ejemplo del tracker que nos ocupa. Muchos de estos proyectos escapan de los modelos convencionales de programación para GPU, es decir, generar efectos gráficos, y se adentran en el ámbito del GPGPU, que abarca investigaciones científicas o pruebas de simulación.

Sin embargo, el paralelismo inherente a esta herramienta requiere una adecuada reestructuración de los algoritmos. En este trabajo se describen tanto los algoritmos de partida como los aspectos específicos requeridos para mejorar su rendimiento en esta arquitectura de tipo paralelo.

2. Antecedentes

Las técnicas del tracking para realidad aumentada suelen ser de tres formas: la realidad aumentada creada con restricciones geométricas con marcadores (*Artoolkit*, [KB99]), la realidad aumentada creada con restricciones geométricas sin marcadores [CPVVG00] y la creada mediante la utilización de métodos probabilísticos [LP06].

La técnica del filtro de partículas tiene sus comienzos con N. Gordon, D. Salmond y A. Smith [GSS93]. Se basaron en el filtro *bootstrap*, método de remuestreo propuesto por Bradley Efron [Efr79] para implementar filtros bayesianos recursivos [RAG04]. Estos filtros siguen un esquema que representa una cadena de Markov, donde el estado presente depende únicamente del estado anterior y de un factor aleatorio. Su función de transición se representa de este modo:

$$S_k = f(S_{k-1}, n_k)$$

donde S_k y S_{k-1} son los estados presentes y anteriores respectivamente, n_k es el parámetro de perturbación y f es una función de transición [BLN04]. Tras un proceso de inicialización, el filtro de partículas itera tres etapas:

1. Propagación *(Resample)*
2. Estimación (*Compute*)
3. Predicción (*Prediction*)

En el caso de la realidad aumentada cada frame estará representado por un conjunto de partículas, es decir, cada partícula será una hipótesis sobre la posición de la cámara en el siguiente frame, que estará definida por una orientación (q)

DOI: 10.2312/LocalChapterEvents/CEIG/CEIG09/047-055

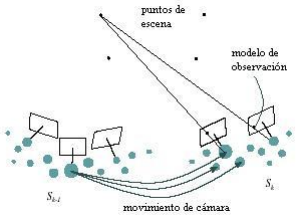

Figura 1: *Partículas inicializadas en el estado x_{k-1} y en el próximo estado x_k. El peso es representado por el tamaño de cada partícula, [LP06]*

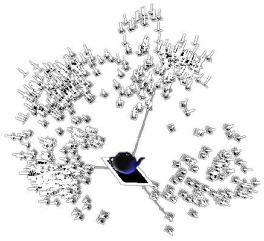

Figura 2: *Representación de distintas partículas generando posiciones diferentes en un espacio 3D, [LP06]*

y una posición ($\vec{t}$) ($x_k = [q,\vec{t}]$) (Figura 1). En la fase de propagación, se generan nuevas partículas, que a continuación se evalúan en la fase de estimación. Por último, en la fase de predicción se selecciona una cantidad fija de partículas que serán las que se propaguen en la siguiente iteración.

Una alternativa al filtro de partículas ha sido el filtro de Kalman [WB01] [Kal60], sin embargo, tiene la desventaja de que solo funciona en sistemas lineales (con funciones de transición lineal).

3. Descripción del algoritmo

3.1. Inicialización

Durante la inicialización, se crean partículas según el propósito propuesto, en este caso seguir el movimiento de una cámara. Se pueden crear aleatoriamente dándoles cualquier posición o rotación o se pueden crear de una manera más ordenada, creando zonas específicas desde donde iniciar las partículas.

En un primer momento, todas las partículas están en la misma posición: en la posición inicial de la cámara. En nuestro caso, se determina esa posición inicial mediante un tracker 2D [SB07] aplicando el algoritmo DLT (*Direct Linear transformation*, [HZ00]) a un objeto virtual de geometría conocida. El estado inicial de las partículas se logrará haciendo un matching 2D-3D entre los puntos detectados en la imágen y los puntos del objeto virtual.

3.2. Propagación

En la propagación se generan nuevas partículas, descendientes de las anteriores. Cada nueva partícula procede de una de las seleccionadas en la generación anterior a la que se introduce una ligera perturbación aleatoria para que el filtro pueda avanzar hacia el mejor resultado. La generación de estas nuevas posiciones es importante pues se trata de conseguir que el movimiento de la cámara coincida lo más posible con algunas de las nuevas partículas. Por tanto, también es importante poder generar el mayor número posible de partículas. El número de partículas generadas debe permitir que las iteraciones del tracking mantengan la frecuencia de visualización en tiempos interactivos.

Existen numerosos métodos para generar partículas, como por ejemplo suponer una velocidad constante o una aceleración constante, sin embargo, tienen algunos inconvenientes. En el caso de los métodos de velocidad constante o aceleración constante se presupone que el movimiento de la cámara será lineal, es decir, que no habrá cambios bruscos en la velocidad o aceleración de la cámara, suposición errónea sobre todo en cámaras manuales, donde siempre habrá pequeñas vibraciones o alteraciones. Utilizando un filtro de partículas, el método de la posición constante es el que mejor resultados obtiene. Este método asume que no existe movimiento entre fotogramas, dándole robustez frente a cambios bruscos en la dirección del movimiento. Se ha optado por generar las partículas utilizando números aleatorios, representados mediante ángulos de Euler [VMK88], y un vector de traslación. Para generar el mayor número posible de partículas se ha decidido utilizar la GPU en vez de la CPU. La potencia de cálculo matricial es muchísimo mayor, ya que además de tener una enorme cantidad de procesadores actuando paralelamente, estos están específicamente diseñados para cálculos con imágenes (matrices al fin y al cabo). Aunque en un caso real las hipótesis no estarían tan dispersas, se puede ver una representación gráfica de cámaras generadas por el filtro de partículas en la Figura 2.

Como se ha dicho en el anterior apartado, se creará una cadena de Markov cuyos estados estarán representados por partículas. La función de transición que se aplica para generar nuevas partículas será la suma de rotaciones y traslaciones aleatorias dadas por el parámetro de perturbación:

$$f^{CP}: R^7 \times R^6 \rightarrow R^7, f^{CP}([q,\vec{t}],[\vec{\omega},\vec{\tau}]) = [\Delta q(\vec{\omega}) \otimes q, \vec{t} + \vec{\tau}]$$

siendo $\vec{\omega}$ una rotación aleatoria descrita mediante los ángulos de Euler, $\Delta q(\vec{\omega})$ su cuaternión equivalente y $\vec{\tau}$ un vector de traslación, por lo que en este caso el vector de ruido será un vector de 6D que ayudará a alterar la rotación y la traslación de la partícula [LP06]. El cuaternión equivalente a la rotación descrita mediante los ángulos de Euler ($\vec{\omega}$) se consigue mediante la siguiente función:

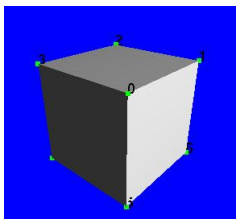

Figura 3: *Representación de un objeto 3D*

$$\Delta q(\vec{\omega}) \approx (\sqrt{1-\varepsilon}, \frac{\omega_x}{2}, \frac{\omega_y}{2}, \frac{\omega_z}{2}) \,/\, \varepsilon = \frac{\|\vec{\omega}\|}{4}$$

3.3. Estimación

Todas las nuevas partículas se analizan durante la fase de estimación, fase donde se calcula el peso de cada partícula, para así poder elegir la hipótesis más probable. Para calcular el peso de cada partícula se hace lo siguiente: partiendo de las coordenadas 3D de un objeto virtual presente en la escena, se calculan sus proyecciones usando la partícula a evaluar. Estas proyecciones se comparan con puntos característicos en la imagen, correspondientes al modelo real detectados por un tracker 2D. Para calcular el punto proyectado ($P^{'}$) por la partícula, basta con aplicar la siguiente función:

$$P^{'} = KMP \,/\, M = [R|\vec{t}], P \in \Re^3$$

donde P será el punto 3D del modelo virtual (ver Figura 3), K será la matriz de calibración (obtenida en preproceso) y M la matriz de modelado calculado a partir de la rotación y la traslación descritas por la partícula.

El peso se asigna en función de la distancia entre los puntos proyectados y los puntos detectados. La función que se ha escogido, es una simplificación de la probabilidad de la observación Gaussiana [LP06] utilizada para filtros Kalman, ya que ésta resulta demasiado pesada:

$$P(y_k|x_k{}^{(n)}, Z) \propto exp(-\sum_{i=1}^{M} d(u(z_i, x_k{}^{(n)}), y_i))$$

donde $x_k{}^{(n)}$ es la partícula a evaluar, y_k la localización de los puntos del modelo real dados por el tracker 2D, z_i el punto i-ésimo del modelo virtual, y_i la localización del i-ésimo punto y u la función proyectora. La función $d(u(z_i, x_k{}^{(n)}), y_i)$ indica si el punto z_i es un *inlier* o un *outlier*, es decir, si la distancia entre el punto proyectado por la partícula y el punto obtenido por el track-er 2D supera un threshold ε_d:

$$d(u(z_i, x_k^{(n)}), y_i) = \begin{cases} 1 & if \parallel u(z_i, x_k^{(n)}) - y_i) \parallel > \varepsilon_d \\ 0 & if \parallel u(z_i, x_k^{(n)}) - y_i) \parallel \leq \varepsilon_d \end{cases}$$

3.4. Predicción

Una vez calculados los pesos de cada partícula, se intenta predecir cuáles son las n mejores a partir de las cuales se producirá la siguiente iteración. La mejor partícula servirá para renderizar la imagen y así seguir con el vídeo. La razón para tomar las n mejores en vez de una sola para continuar con la iteración, es que la que más peso tenga no siempre será la mejor partícula, ya que aunque tenga un margen de error inferior a las demas, puede que en el siguiente frame la dirección tomada por la cámara sea distinta (ver Figura 1).

De este modo, hasta que el vídeo a analizar termine, las tres etapas del filtro de partículas se irán ejecutando iterativamente: propagando nuevas partículas en base a las n partículas escogidas en la iteración anterior (propagación), calculando el peso de cada partícula (estimación) y escogiendo las n mejores en base al peso de cada una de ellas para la siguiente iteración (predicción). Así se podrá sincronizar el movimiento de la cámara real y el de la virtual generando imágenes virtuales que se superpongan a las reales.

4. Implementación en GPU del método propuesto

Tomando como referencia la teoría expuesta en el capítulo anterior, se ha implementado un filtro de partículas capaz de calcular los 6 grados de libertad de la cámara que captura un vídeo. Dado que la programación en GPU tiene ciertas restricciones, se ha adaptado el algoritmo y creado las estructuras de datos necesarias.

4.1. Estructuras de datos

Es importante escoger bien cada estructura que se vaya a utilizar, limitada principalmente por una razón: la GPU pone ciertas restricciones a la hora de pasar o recoger datos. Tanto los datos de entrada como de salida se almacenarán en texturas. Además, es importante minimizar el flujo de datos entre la CPU y la GPU ya que estas transferencias son muy costosas en tiempo.

Cada partícula representará una rotación y una traslación de la cámara. Las traslaciones son vectores de 3 elementos y pueden guardarse en texturas rectangulares de tipo RGB en coma flotante, sin embargo, las rotaciones son matrices de 3×3, por lo que se ha optado por utilizar cuaterniones para representarlas. De esta forma se pueden almacenar en texturas rectangulares de tipo RGBA en coma flotante. Así, el conjunto de partículas quedará almacenado en dos texturas (ver Figura 4) correspondiendo cada téxel a una partícula.

Las perturbaciones necesarias para la fase de propagación también se almacenan en texturas, ya que no pueden ser generadas en la GPU. Dado que cada parámetro de perturbación se compone de una rotación en ángulos de Euler y una traslación, se usarán dos texturas rectangulares RGBA de forma análoga a las usadas para las partículas.

4.2. Adaptación del algoritmo

A la hora de implementar un filtro en la GPU hay que tener en cuenta si el algoritmo se puede paralelizar. Dado que

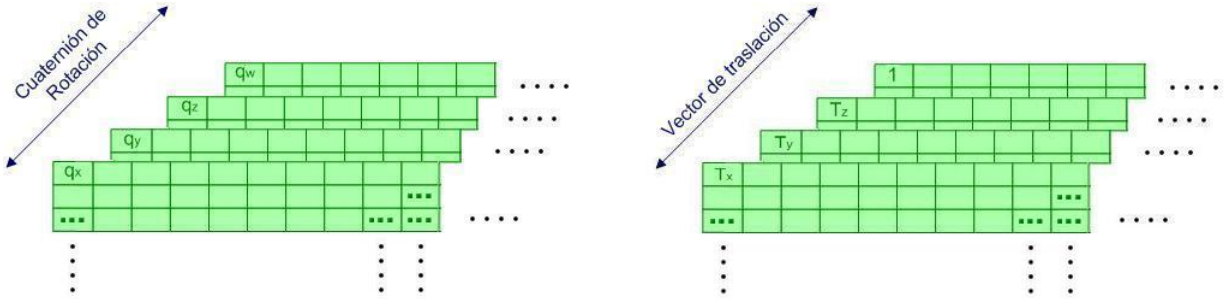

Figura 4: *Representación de las rotaciones y traslaciones en una textura en formato RGBA*

cada partícula representa una hipótesis independiente de las demás, las fases de propagación y estimación no presentan dependencias de datos. Así, estas etapas han sido implementadas utilizando *fragment shaders*, donde cada partícula es procesada por un fragmento diferente. Sin embargo, la fase de predicción no se presta tanto porque hay que comparar todas las partículas entre ellas para encontrar las n mejores. Aunque existen algoritmos de ordenación en GPU [SA08], se han desestimado por no ser adecuados para este trabajo, manteniendo así esta etapa en CPU.

4.2.1. Propagación

La fase de propagación se ha implementado en un *fragment shader* independiente. Sus datos de entrada serán las n mejores partículas escogidas en la iteración anterior y las perturbaciones. Como salida devuelve las nuevas partículas generadas. Cada partícula de las n mejores escogidas, generará un conjunto de partículas nuevas. La estrategia utilizada es dividir el *viewport* en n particiones. En cada una de las particiones se genera el conjunto de nuevas partículas correspondientes a cada una de las n escogidas (ver Figura 5). El número total de partículas generadas dependerá del tamaño total del *viewport*, ya que el fragment shader se ejecutará una vez por cada píxel de éste.

Para cada subdivisión i del *viewport*, los parámetros del *fragment shader* serán:

Datos de entrada: Q, T, x_{k-1}^{i}
Datos de salida: x_k

siendo Q y T las texturas con las perturbaciones, x_{k-1}^{i} la partícula a partir de la cual se generan las nuevas (parámetro uniforme) y x_k el conjunto de partículas generadas para el frame k.

En cada iteración del algoritmo hay que generar nuevos números aleatorios. Actualmente ésto no es posible hacerlo en la GPU de manera eficaz, por lo que es necesario hacerlo en la CPU. Por cuestiones de eficiencia, se ha optado por generar una tabla de números en la fase de inicialización y usarla para todas las iteraciones. Este detalle convierte el algoritmo en un método *pseudoestocástico* pero optimiza el ancho de banda entre la memoria principal y la gráfica.

4.2.2. Estimación

La etapa de estimación se ha implementado usando otro *fragment shader* independiente. Sus datos de entrada serán todas las partículas generadas en la fase de propagación, los puntos 3D del modelo virtual y sus correspondencias en el frame actual detectadas por el tracker 2D. En este caso el *shader* se ejecuta en el *viewport* completo, siendo su salida el peso de cada partícula.

Las partículas generadas se le pasan al *shader* como dos texturas (ver Figura 4). Sin embargo, al tratarse de los datos de salida generados por el *shader* de la fase de propagación, no se requiere ninguna transferencia de datos desde la memoria principal del ordenador. Los puntos del objeto virtual sólo hace falta transferirlos en la fase de inicialización, ya que no cambian a lo largo de la secuencia de vídeo. Sin embargo, los puntos del tracker 2D sí que tienen que ser transferidos a la memoria de la gráfica en cada fotograma. De todos modos, no supone demasiada penalización ya que se trata de muy poca información (2 números por punto).

Datos de entrada: x_k, y_k, P
Datos de salida: $\vec{w}_k$

siendo x_k el conjunto de partículas generadas para el frame k, y_k el conjunto de puntos dados por el tracker 2D, P los puntos 3D del modelo virtual y $\vec{w}_k$ el vector de pesos asociado a las partículas.

4.2.3. Predicción

En la fase de predicción se analizan todos los pesos obtenidos en la fase de estimación. Para ello habrá que descargar la textura que los contenga y luego buscar en la CPU los n mejores. Esta fase es la más pesada, ya que si el número de partículas que se estén analizando es muy elevado, la búsqueda de las mejores resulta lenta. Si al tiempo

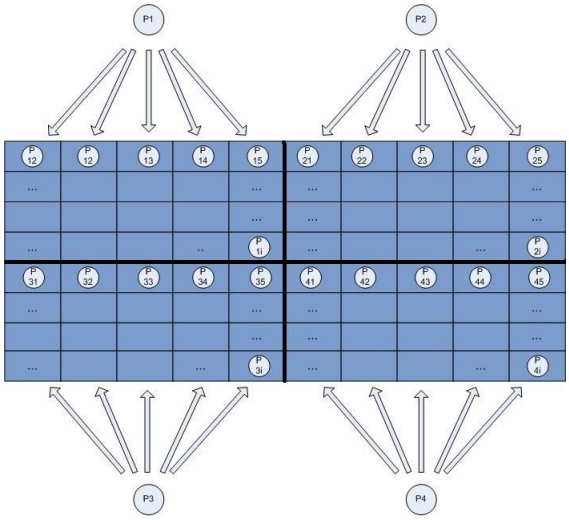

Figura 5: *Representación de la propagación de las partículas en el viewport, partiendo de las n (en este caso n=4) mejores*

de ejecución del algoritmo de comparación (lineal respecto al número total de partículas) se le suma el tiempo necesario para leer la textura de pesos, además del tiempo necesario para obtener también las *n* mejores partículas desde la memoria gráfica, veremos que nos encontramos en el cuello de botella de todo el proceso.

5. Resultados

Se han hecho las pruebas correspondientes con dos configuraciones: una implementada sobre la CPU, usando la librería OpenMP (API para la paralelización de instrucciones) para optimizar las fases más pesadas computacionalmente, y otra implementada en la GPU. Estas son las características del equipo utilizado en las pruebas:

CPU

Intel Pentium(R) 4
3.00GHz de velocidad de reloj del procesador
265KB de memoria caché
1 Núcleo con tecnología *Hyper Threading*

GPU

NVIDIA GeForce 9800 GT
112 procesadores gráficos
512MB de memoria
600MHz de velocidad de reloj del p. gráfico
1500MHz de velocidad de reloj del p. central
1500MHz de velocidad de reloj de la memoria
57.6GB/s de ancho de banda de memoria

El sistema operativo ha sido Microsoft Windows XP Service pack 3, con 1.00GB de RAM y 50GB de disco duro.

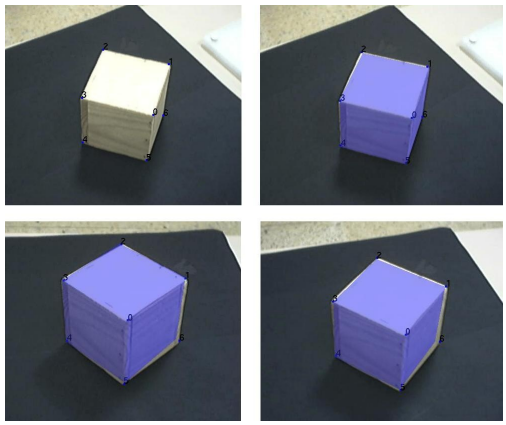

Figura 6: *Evolución del tracking mediante el filtro de partículas en un vídeo*

El escenario de pruebas ha sido un vídeo de 589 frames, en el cual la tarea del filtro de partículas ha sido la de rastrear un objeto geométrico (un cubo rígido), para así poder averiguar la posición de la cámara en cada frame (ver Figura 6). El número máximo de vértices ocluidos durante el vídeo son 4. No hay objetos móviles en la escena. Las pruebas han sido realizadas con dos objetivos: medir la velocidad de cálculo, expresado por la cantidad de partículas que puedan generar los filtros en cada milisegundo, y medir la tasa de error. Para apreciar mejor las diferencias, se han hecho pruebas con 3 configuraciones distintas: la primera creando pocas partículas, entre 500 y 4500, la segunda creando partículas suficientes como para que funcione correctamente el filtro de partículas, entre 4K y 10K, y la tercera, entre 100K y 500K,

Particles	FPS	Generate T.	Compute T.	Resample T.	GetBest T.
500	14.38	1.08	6.77	26.6	0.644
1000	6.637	1.94	13.2	59.4	1.38
1500	11.38	2.98	19.8	20.4	2.16
2000	8.543	4.03	26.5	27.2	2.7
2500	6.812	5.11	33.1	33.9	3.61
3000	5.65	6.34	39.7	40.8	4.39
3500	4.864	7.41	46.3	47.5	4.81
4000	4.249	8.24	53	54.4	5.74
4500	3.782	8.99	59.5	61.1	6.52

Tabla 1: *Tabla de resultados generados mediante la CPU (de 500 a 4500 partículas, T en ms)*

Particles	FPS	Generate T.	Compute T.	Resample T.	GetBest T.
500	273.2	0.497	0.376	1.05	0.209
1000	267.4	0.455	0.282	1.17	0.165
1500	267.4	0.427	0.355	1.21	0.188
2000	249.2	0.438	0.345	1.25	0.212
2500	253.9	0.455	0.322	1.24	0.203
3000	252.2	0.464	0.306	1.23	0.151
3500	247.3	0.395	0.334	1.31	0.172
4000	249.2	0.482	0.269	1.32	0.225
4500	244.2	0.41	0.385	1.33	0.193

Tabla 2: *Tabla de resultados generados mediante la GPU (de 500 a 4500 partículas, T en ms)*

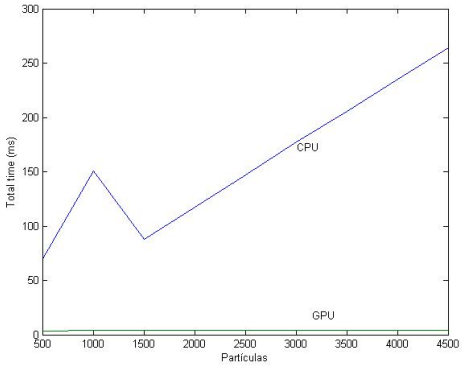

Figura 7: *Representación gráfica del tiempo total de cada frame obtenido con 500-4500 partículas*

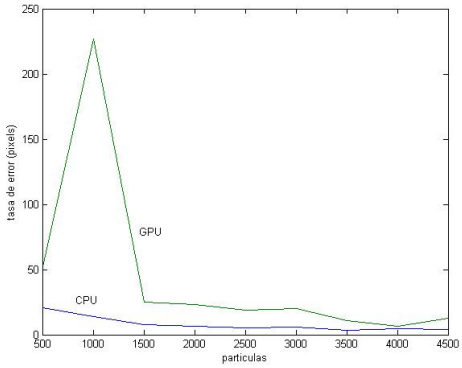

Figura 8: *Representación gráfica de la tasa de error obtenida con 500-4500 partículas*

con la intención de medir la capacidad bruta que pueda tener la implementación de GPU. Se deberá medir la tasa de error para descubrir cúando empieza a funcionar bien el tracker creado mediante el filtro de partículas. No es suficiente generar miles de partículas por segundo, es necesario que se haga correctamente el tracking con esa cantidad de partículas. En todos los casos el número de partículas escogidas en la fase de predicción es 10 y la máxima perturbación es un grado de rotación y un centímetro de traslación en cada eje.

Para las tres configuraciones se han creado tablas con información sobre las tres etapas del filtro: cantidad de partículas analizadas, FPS obtenido, tiempo de generación de partículas (*Generate*), tiempo de computo de pesos (*Compute*), tiempo de preparación de las partículas (*Resample*) y tiempo de selección de partículas (*GetBest*). El *generate* corresponde con la etapa de propagación, el *compute* con la etapa de estimación y la suma de *resample* y *getbest* con la etapa de predicción. No se ha medido el tiempo de respuesta del tracker 2D, sin embargo queda reflejado en el tiempo total. El lector interesado puede consultar [SB07].

5.1. Configuración de pequeña acumulación de partículas

Como se puede ver en las Tablas 1 y 2, mientras que la implementación de la CPU no llega a crear ni 500 partículas en tiempo real (25-30 frames por segundo), la de GPU puede crear 4500, 15 veces más rápido, a 273,2 FPS (ver Figura 7). El máximo y el mínimo entre todas las pruebas de esta configuración generadas en la GPU han sido 302,7 y 216,2 FPS respectivamente.

Se puede apreciar en la Figura 8 que para que el filtro implementado en la CPU empiece a funcionar adecuadamente, con una tasa de error entre 5 y 15 píxels, son necesarias más de 1500 partículas, mientras que la implementación en la GPU aún tiene una tasa de error bastante alta. Esto es debido a que no se generan números aleatorios en cada frame, degradando así el rendimiento (ver apartado 4.2.1) y al hecho de que la aritmética de coma flotante de la tarjeta gráfica es menos precisa que la de la CPU (32 bits Vs 64 bits). Esto aconseja ampliar los experimentos, pues con esta tasa de error el filtro de partículas no puede funcionar como un tracker 3D adecuado.

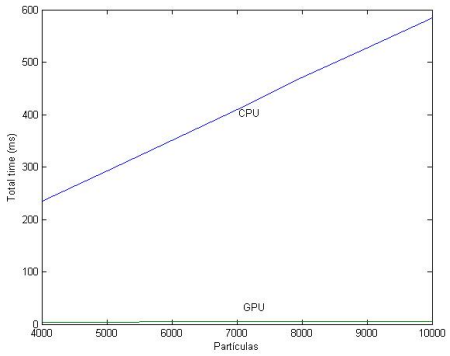

Figura 9: *Representación gráfica del tiempo total de cada frame obtenido con 4K-10K partículas*

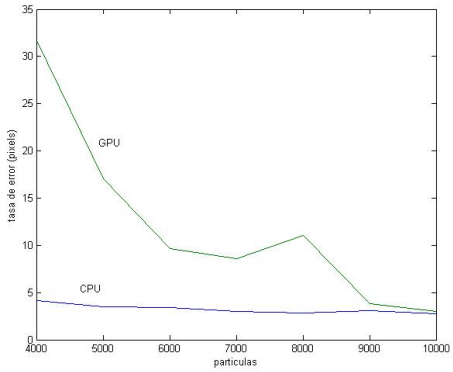

Figura 10: *Representación gráfica de la tasa de error obtenida con 4K-10K partículas*

5.2. Configuración de mediana acumulación de partículas

Particles	FPS	Generate T.	Compute T.	Resample T.	GetBest T.
4000	4.276	8.42	52.3	54	5.86
5000	3.41	10.4	65.9	67.7	6.73
6000	2.853	12.4	78.7	81.1	8.09
7000	2.443	14.6	92.1	94.5	9.25
8000	2.126	16.3	106	109	11.2
9000	1.9	18	118	122	12.2
10000	1.71	20.5	132	135	13.7

Tabla 3: *Tabla de resultados generados mediante la CPU (de 4K a 10K partículas, T en ms)*

Particles	FPS	Generate T.	Compute T.	Resample T.	GetBest T.
4000	242.6	0.485	0.358	1.27	0.257
5000	243.4	0.374	0.329	1.4	0.235
6000	221.9	0.537	0.367	1.45	0.213
7000	221.1	0.408	0.386	1.47	0.241
8000	217.3	0.305	0.399	1.58	0.214
9000	218.9	0.416	0.44	1.53	0.261
10000	200.8	0.497	0.478	1.7	0.214

Tabla 4: *Tabla de resultados generados mediante la GPU (de 4K a 10K partículas, T en ms)*

Como se puede apreciar en la Tabla 3, los FPS en los que se mueve el filtro implementado en la CPU bajan drásticamente, no pudiendo matener el tracking en tiempo real ni con 4K partículas. Sin embargo, dada la actual evolución de las CPU, llegando hoy en día a tener 4 núcleos que trabajan a 2-3GHz, se puede suponer que pueda alcanzar la velocidad de 25-30 FPS con esa cantidad de partículas.

Mientras tanto, la implementación en la GPU sigue superando la barrera de los 25-30 FPS, (entre 8 y 10 veces), incluso creando 10000 partículas (Tabla 4). El máximo y el mínimo en esta configuración ha sido de 264,9 y 195,3 FPS.

Al interpretar la tasa de error en la Figura 10 se puede apreciar que la GPU obtiene tasas de error más aceptables, aunque sigue dando una tasa superior de errores, con igual cantidad de partículas, que la CPU. En torno a 9K-10K partículas la tasa error de la implementación en GPU es similar a la de CPU.

Particles	FPS	Generate T.	Compute T.	Resample T.	GetBest T.
100000	0.2567	140	1050	707	71.3
200000	0.1283	276	2120	1410	142
300000	0.08559	411	3180	2110	212
400000	0.06432	548	4240	2800	282
500000	0.05142	685	5300	3500	353

Tabla 5: *Tabla de resultados generados mediante la CPU (de 100K a 500K partículas, T en ms)*

5.3. Configuración de gran acumulación de partículas

Con una gran acumulación de partículas se verá la potencia de cálculo de la implementación en GPU respecto a la implementación en CPU. Si se analizan las Tablas 5 y 6, se puede ver claramente que ni siquiera con una CPU de cuatro núcleos se puede alcanzar una frecuencia de 25 FPS, mientras que con una tarjeta gráfica un poco más potente se podrá alcanzar esa cifra creando 500*K* partículas por frame. El máximo y el mínimo han sido 78,2 y 19,03 FPS. Respecto a la evolución de cada etapa del filtro, el tiempo tiende a subir linealmente (ver Figura 11).

La tasa de error se ha bajado en la implementación en la GPU hasta situarse en valores parecidos a la implementación en la CPU (ver Figura 12), oscilando entre 3 y 2 píxels.

Particles	FPS	Generate T.	Compute T.	Resample T.	GetBest T.
100000	71.54	0.487	1.54	6.59	0.687
200000	41.08	0.493	2.64	12.9	1.44
300000	30.38	0.622	3.71	17.8	1.77
400000	23.25	0.455	4.94	23.2	2.14
500000	20.27	0.492	6.05	26.7	2.45

Tabla 6: *Tabla de resultados generados mediante la GPU (de 100K a 500K partículas, T en ms)*

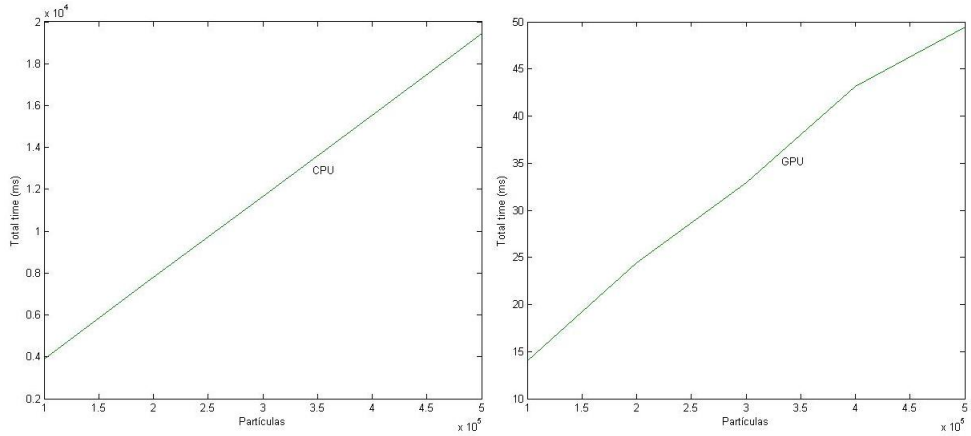

Figura 11: *Representación gráfica del tiempo total de cada frame obtenido con 100K-500K partículas*

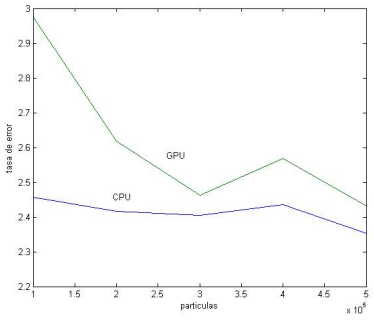

Figura 12: *Representación gráfica de la tasa de error obtenida con 100K-500K partículas*

6. Conclusiones

Esta investigación ha querido comprobar la eficiencia y la precisión del filtro de partículas en la GPU respecto a la CPU. La paralelización del algortimo para su implementación en la GPU ha requerido adaptaciones en las estructuras de datos y en su arquitectura. También se exponen las variantes introducidas para mejorar la eficiencia.

La investigación incluye una serie de experimentos en los que se analizan los factores clave para obtener una implementación útil del algoritmo: número de partículas generadas en cada iteración, FPS obtenido y tasa de error.

Es previsible que pronto CPUs más potentes y con mayor número de procesadores puedan mantener un frame-rate de tiempo real generando del orden de 4-10K partículas y precisiones del orden de los 5 píxels. Actualmente las GPUs permiten superar con creces los requisitos de tiempo real, con precisión del orden de 2-3 píxels. Esto se logra generando gran número de partículas: 1-5M partículas.

Agradecimientos

El Dr. García-Alonso colaboró en esta investigación con ayuda del Ministerio de Educación TIN2006-14968-C02-01, y el contrato de Jairo Sánchez está financiado por el Programa Torres Quevedo del Ministerio de Educación y Ciencia y cofinanciado por el Fondo Social Europeo.

References

[BLN04] BASHARIN G. P., LANGVILLE A. N., NAUMOV V. A.: *The Life and Work of A. A. Markov*. Elsevier Science, New York, NY, ETATS-UNIS, 2004.

[CPVVG00] CORNELIS K., POLLEFEYS M., VERGAUWEN M., VAN GOOL L.: Augmented reality using uncalibrated video sequences. *2nd European workshop on 3D structure from multiple images of large-scale environments 2018* (2000), 144–160.

[Efr79] EFRON B.: *Bootstrap methods: Another look at the jackknife*. Institute of Mathematical Statistics, 1979.

[GSS93] GORDON N., SALMON D., SMITH S.: A novel approach to nonlinear/non gaussian bayesian state estimation. *Radar and Signal Processing, IEE Proceedings F 140* (1993), 107,113.

[HZ00] HARTLEY R., ZISSERMAN A.: *Multiple View Geometry in computer vision*. Cambridge University Press, 2000.

[Kal60] KALMAN R.: A new approach to linear filtering and prediction problems. *Journal of Basic Engineering 82* (1960), 35–45.

[KB99] KATO H., BILLINGHURST M.: Marker tracking and hmd calibration for a video-based augmented reality conferencing system. In *International Workshop on Augmented Reality (IWAR)* (San Francisco, USA, 1999), pp. 85–94. Uso de ARToolkit.

[LP06] LLOYD PUPILLI M.: *Particle filtering for real-time camera localisation*. PhD thesis, University of Bristol, 2006.

[RAG04] RISTIC B., ARUMPALAM S., GORDON N.: *Beyond the Kalman Filter: Particle Filters for Tracking Applications*. Artech House, 2004.

[SA08] SINTORN E., ASSARSSON U.: Fast parallel gpu-sorting using a hybrid algorithm. *Journal of Parallel and Distributed Computing* (2008).

[SB07] SÁNCHEZ J., BORRO D.: Non invasive 3d tracking for augmented video applications. In *IEEE Virtual Reality 2007 Conference, Workshop "Trends and Issues in Tracking for Virtual Environments"(IEEE VR'07)* (Charlotte, North Carolina, USA., 2007), pp. 22–27.

[VMK88] VARSHALOVICH D. A., MOSKALEV A. N., KHERSONSKII V. K.: Description of rotation in terms of euler angles. In *Quantum Theory of Angular Momentum* (Singapore: World Scientific, 1988), pp. 21–23.

[WB01] WELCH G., BISHOP G.: An introduction to the kalman filter. In *SIGGRAPH (Computer Graphics)* (Los Angeles, CA, USA, 2001). Curso sobre el Kalman Filter Disponibles las slides (en otro curso mas generico "Tracking: Beyond 15 Minutes of Thought").