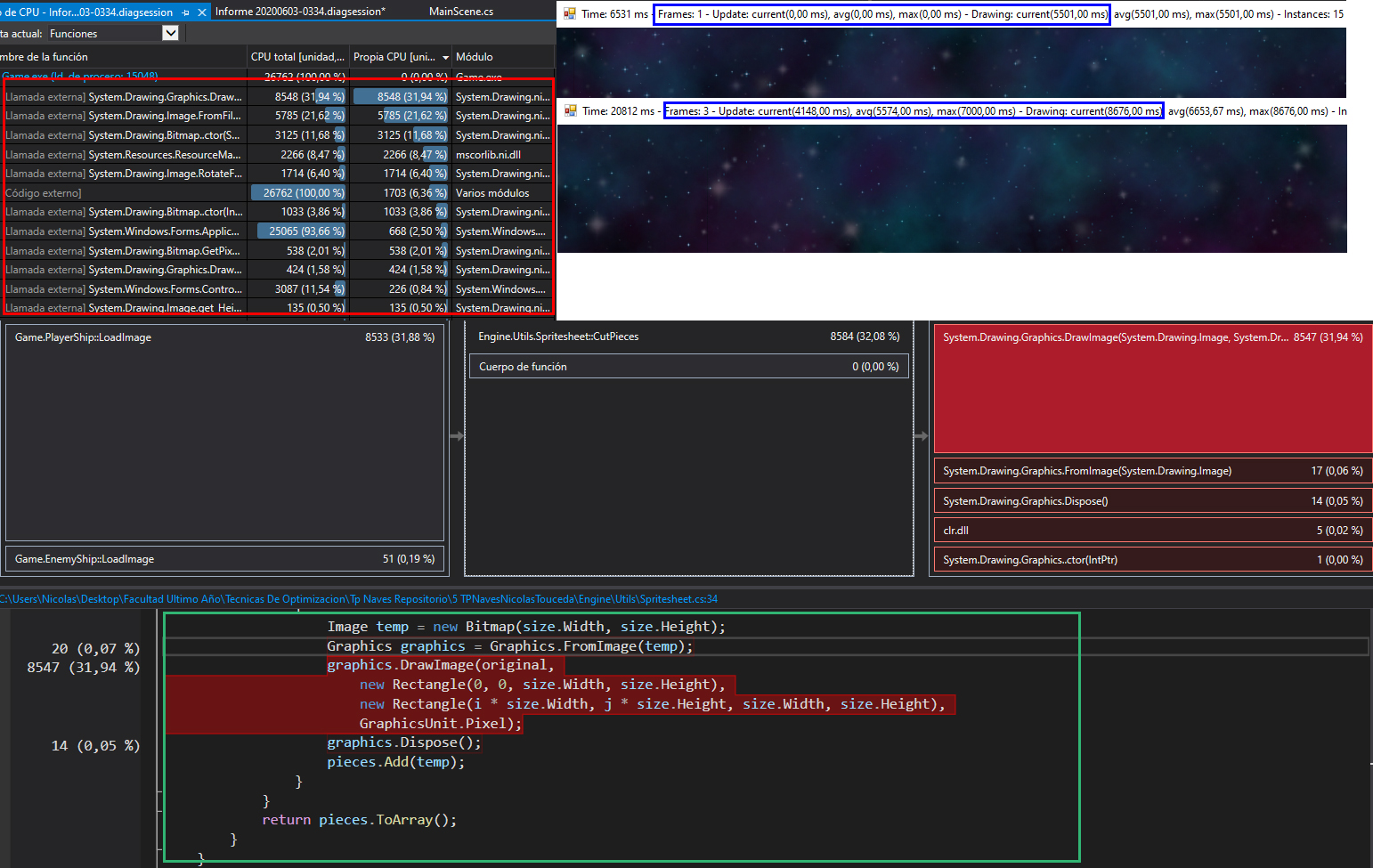
TP NAVES Técnicas De Optimización

Nicolas Touceda

HIPOTESIS 1

Mediciones:

Hipotesis:

Los principales problemas encontrados, gracias a las mediciones son:

En el Cuadro Rojo de la imagen: Encontramos el Primer HotSpot, donde el método de dibujo (System.Drawing.Graphis.Draw…) Utiliza muchos recursos y ocupa el 32% del procesador, esto genera mucho retraso en el programa.

Cuadro Verde de la imagen: Encontramos un bloque de código, graphics.DrawImage() que se encuneta en el método public Image[] CutPieces(Size size), este manda a dibujar la lista de imágenes y ocupa el 32% del tiempo de la CPU.

Cuadros Azules de la imagen: Vemos que en cada Frame que se ejecuta el juego, el Drawing tarda entre 5 a 8 segundos en dibujar. Esto es un problema porque no deja avanzar el programa.

Teniendo estas 3 mediciones en cuenta sabemos que este método draw es un problema.

Revisando el código llegue a la conclusión de que tanto el PlayerShip como EnemyShips, al momento de ser dibujadas, llaman 2 veces al método LoadImage(). Lo que este método hace es desencadenar una serie de pasos, en el que, se recortan y dibujan todas las imágenes y son almacenadas en un array, finalmente este array es retornado, para ser dibujado (Esto ocurre en en CutPieces() dentro de la clase Spritesheet()).

El problema de esto es que, cada vez que quiero cargar una imagen de una nave, el programa manda a recortar y crear todas las imágenes de vuelta.

Solución Propuesta:

Modificar 2 métodos para un mejor funcionamiento del programa, primero el método CutPieces voy a implementar un IF para que cree las imágenes una sola vez y no tenga que crearlas cada vez que se le llama el método LoadImage()

Voy a modificar el Método LoadImage() para acceder a mi imagen directamente por el índice de array y voy a guardar esta imagen en una variable.

Resolución:

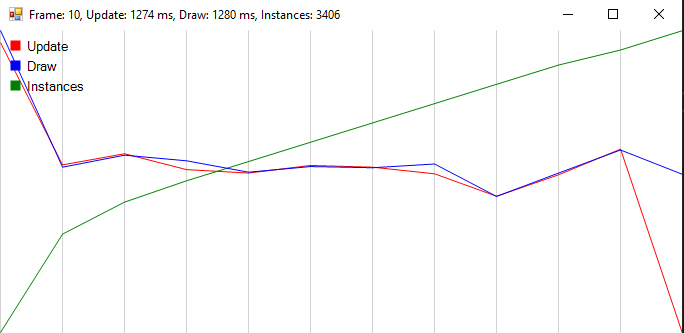


Comparando esta imagen con la de medición, llegue a la conclusión que el programa mejoro mucho, ya que el tiempo de dibujado paso de 5-8 segundos a 1-2 segundos y por ende el programa avanza mucho más rápido, en el mismo tiempo de prueba avanzo 9 frames en vez de 3.

La Implementación y los cambios Fueron Positivos.

Hipotesis 2

Mediciones:





Hipotesis:

Como podemos ver, en la imagen de medición, hay un problema en la cantidad de instancias, estas se multiplican excesivamente frame a frame.

Revisando la lista de objetos, All Children, note que frame a frame se añaden más muchas estrellas que nunca se liberan de la memoria.

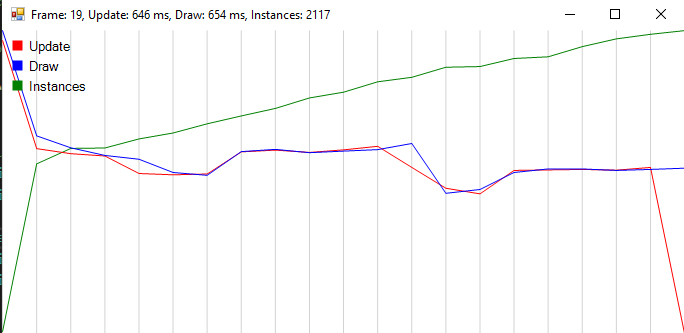
Solución Propuesta:

Lo que propongo es crear una nueva estructura de datos para separar las estrellas de AlChildren.

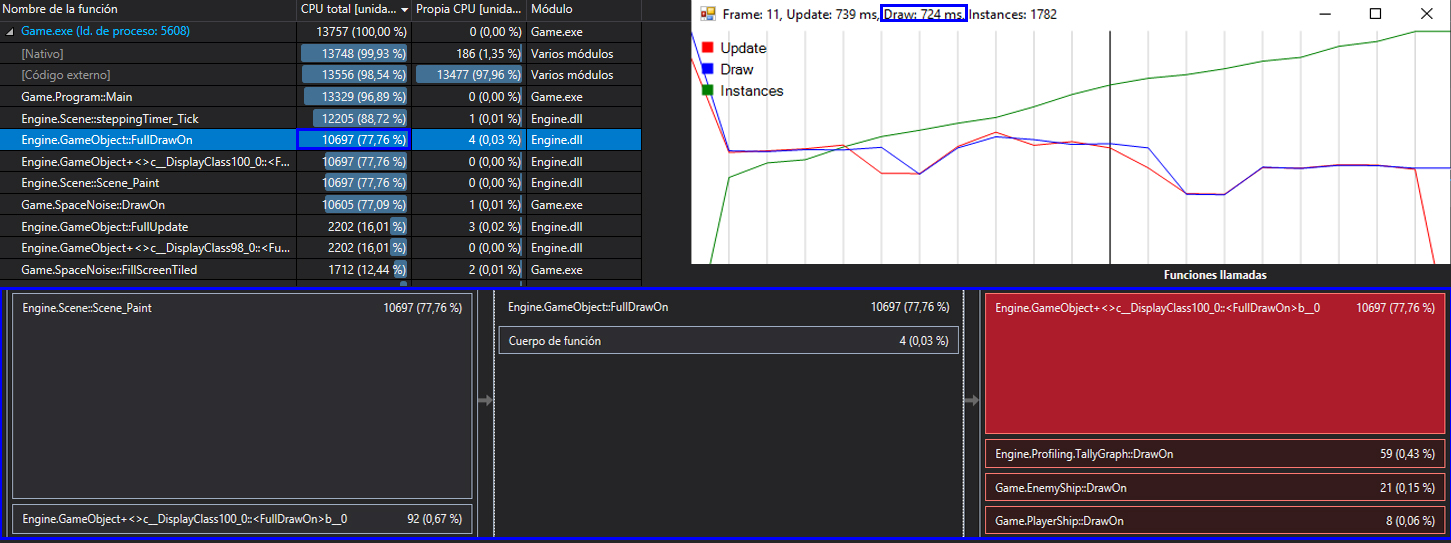
Una vez que tenga esto, voy a implementar algún método que elimine las estrellas que estén fuera de pantalla, de esta manera voy a bajar mucho el número de instancias.

Resolucion:

Bajo mucho el numero de instancias a comparacion con la primer medicion, las estrellas se liberan al salir de la pantalla el cambio fue positivo.



Hipotesis 3

Mdiciones: 

Hipotesis:

Revisando las nuevas imágenes de medición, destaco un problema en el método FullDrawOn(), que ocupa un 77,77% del CPU Total, Este es llamado por la Scene\_Paint.

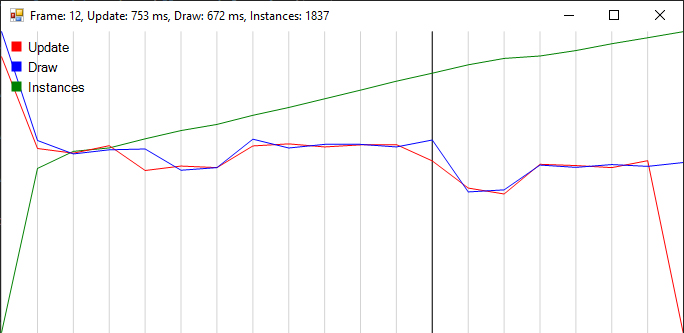
Entre frame y frame se tarda entre 600 a 800 ms en dibujar los objetos en pantalla

Revisando el código pude notar que el método FullDrawOn(), agarra la lista de objetos, y por cada objeto se vuelve a llamar a si misma varias veces, esto provoca que el mismo objeto se llame más de una vez, y cuantos más objetos tenga la lista, más veces se va a recorrer.

Solución Propuesta

Propongo modificar el método FullDrawOn() para que cada objeto se dibuje una sola vez.

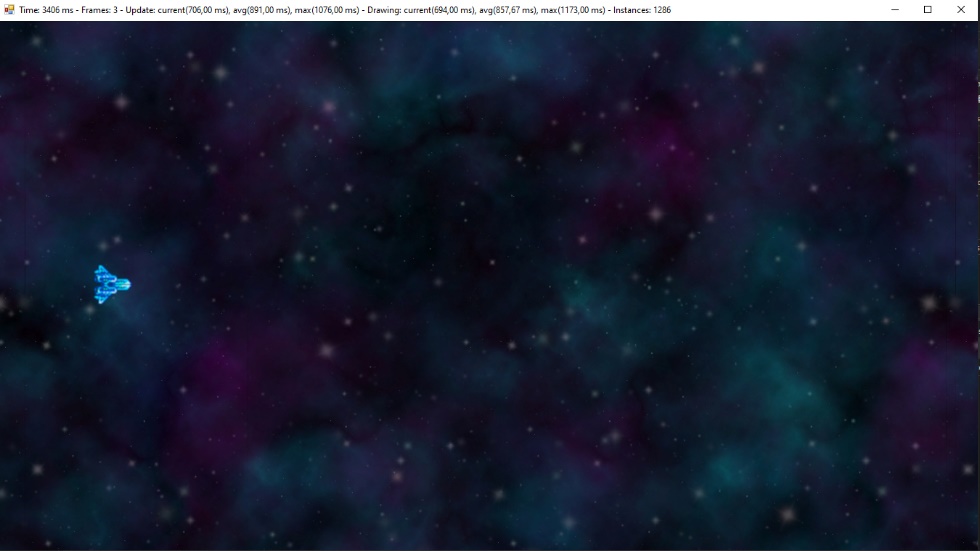
Resolución:



No hubo cambios muy significativos en el tiempo de dibujado, pero ahora recorre 1 sola vez los objetos, el cambio es positivo.

Hipotesis 4

Mediciones:



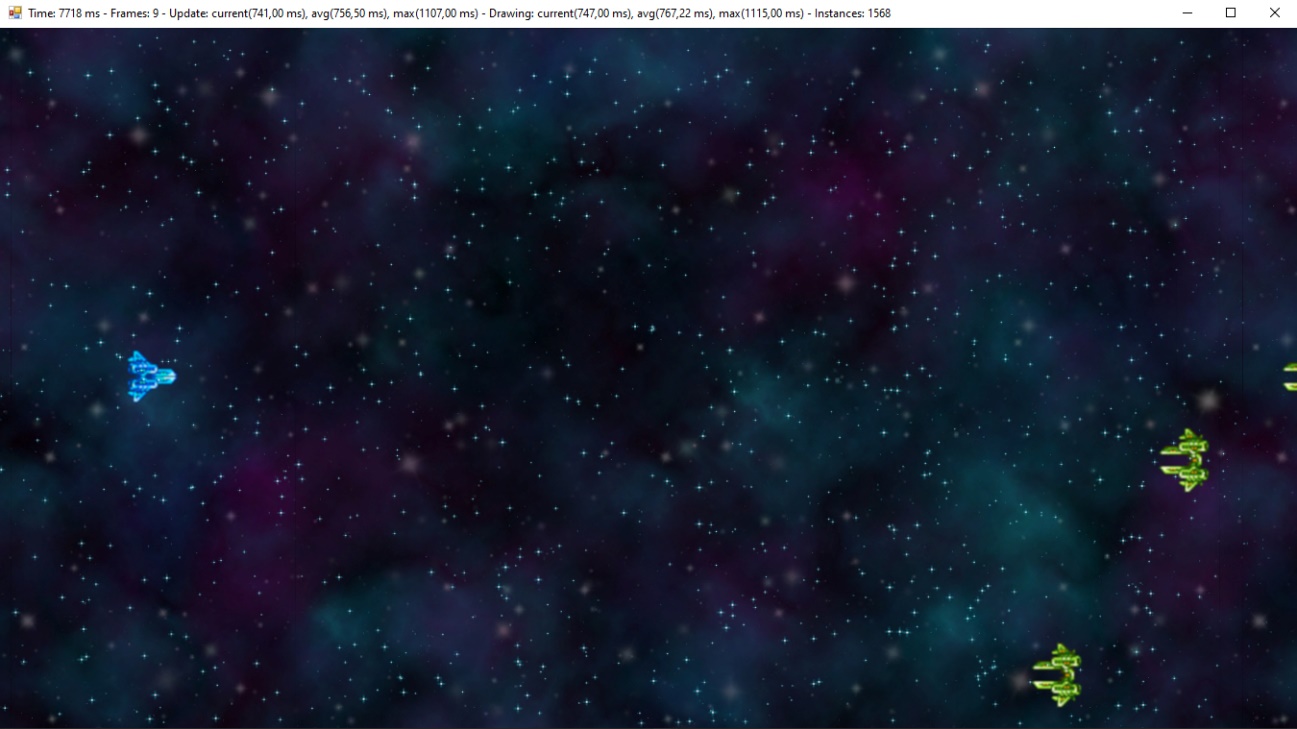
Hipotesis:

Encontré un bug que fue causado en mi Hipotesis 2, al separar las estrellas de la lista children, estas ya no se dibujan en pantalla, esto lo podemos ver en la imagen de la medición.

Solución Propuesta

Que en el FullDrawOn() mande a dibujar la estructura de datos que contiene las estrellas.

Resolución:



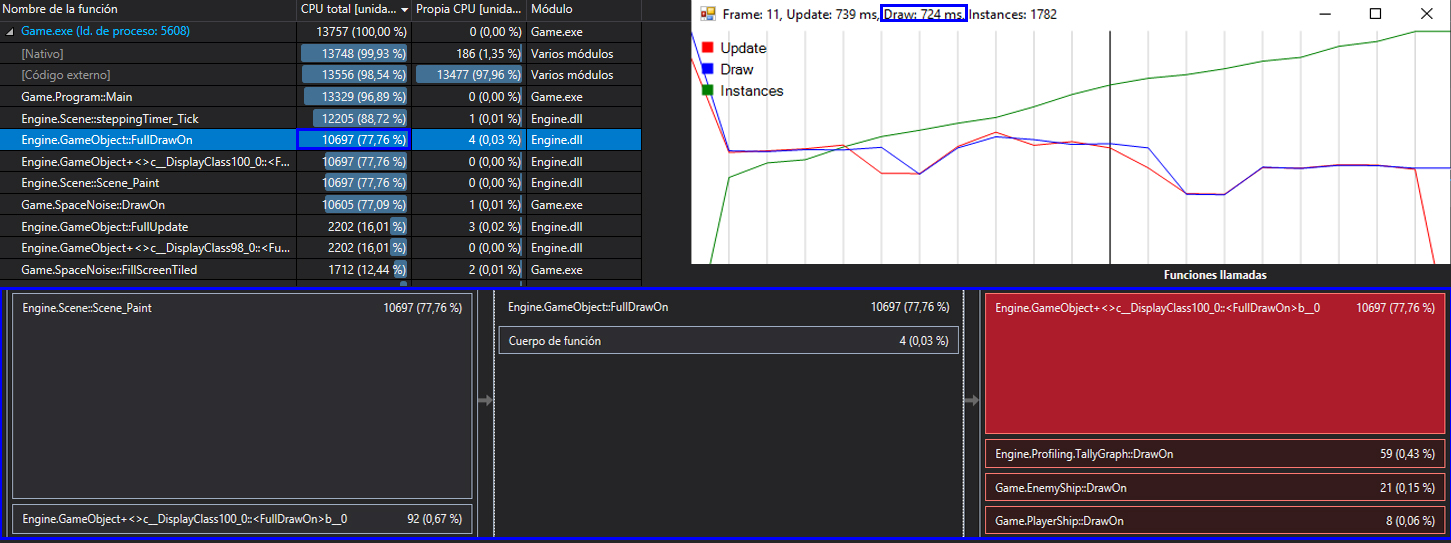
Como podemos ver en la imagen de resolución, si la comparamos con la imagen de medición, las estrellas se dibujan correctamente.

El cambio fue positivo ya que el bug fue arreglado.

Hipotesis 5

Mediciones:

Aunque ya aplique varias optimizaciones al dibujado, sigue habiendo un problema que retrasa todo el juego. Voy a utilizar las mismas mediciones de la Hipotesis 3 ya que el problema del dibujado no fue solucionado.



Hipotesis:

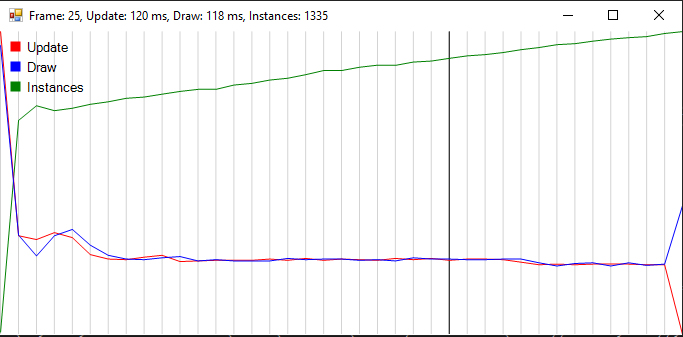
Revisando el DrawOn De cada objeto, encontré que el SpaceNoice realiza muchos cálculos y se dibuja varias veces, realizando varias pruebas, como eliminar estos objetos, la aplicación funcionaba mucho mejor, entonces llegué a la conclusión que los objetos de tipo SpaceNoice ralentizan mucho el programa al momento de ser dibujados, ya que, dentro de este, en todo momento, se están creando nuevas imágenes para ser dibujadas y el dibujo es actualizado muchas veces.

Solución Propuesta

Para arreglar este problema propongo, guardar la imagen base e ir interactuando sobre su Point, en vez de crear todo el tiempo nuevas imágenes como estaba ocurriendo en el programa.

Dentro del código tengo 4 objetos de tipo SpaceNoice, propongo juntar las 4 imágenes en una para tener un solo objeto SpaceNoice.

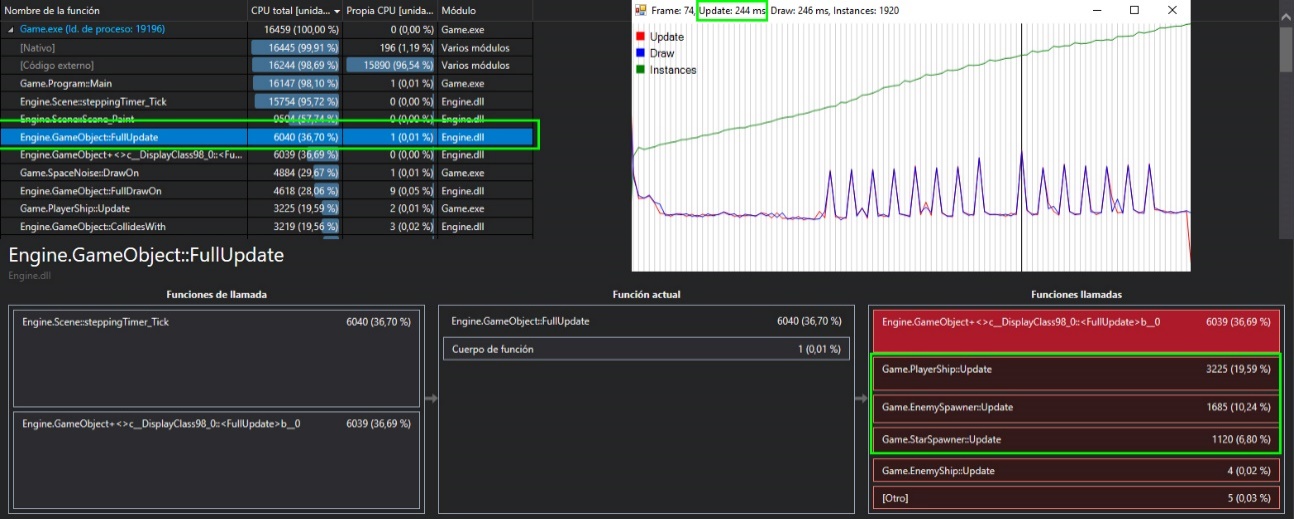
Resolución:



Los cambios fueron muy positivos, mejoro mucho la velocidad del programa, Los frames avanzan mucho mas rápido, el dibujado tarda 120ms, en si el programa avanza a una velocidad más jugable.

Hipotesis 6

Mediciones:



Hipotesis:

Revisando el método FullUpdate(), que ocupa el 36,7% del CPU Total, encontré como principal problema la actualización del PlayerShip, ya que ocupa el 19,59% del CPU Total, esto es demasiado para una sola nave, por otro lado, revisando el grafico encontramos que el tiempo máximo que tarda en actualizar es de 244ms.

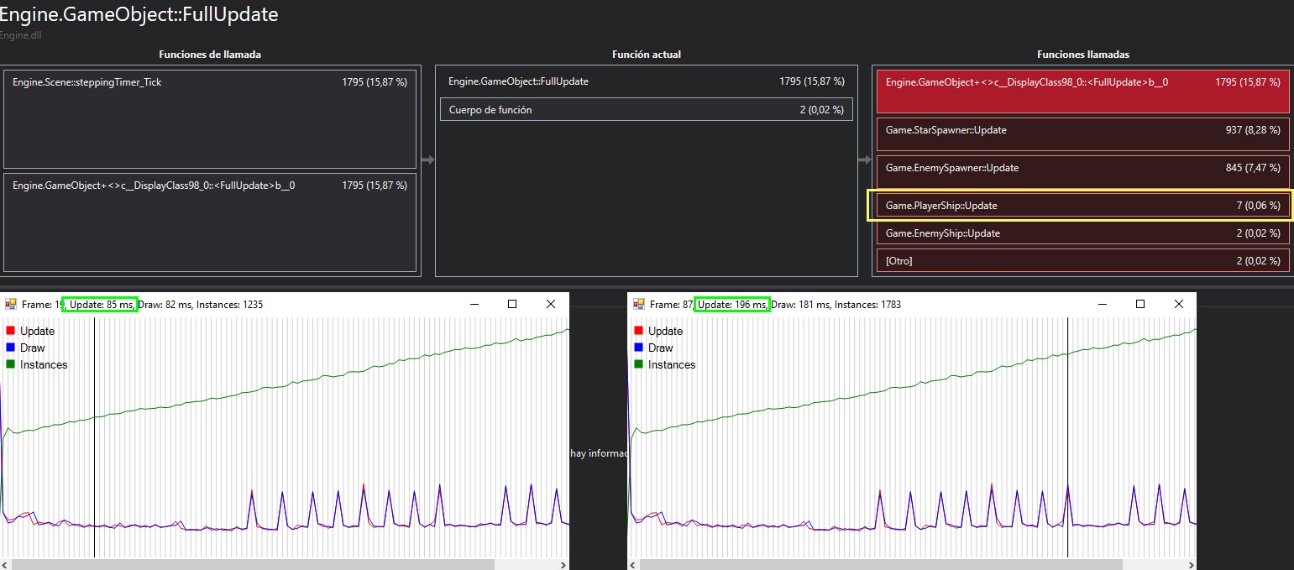
El Método Update(), del PlayerShip, se encuentran dos métodos que retrasan la ejecución, estos son CheckForCollision() y CheckForPowerUps(). Estos métodos agarran la lista de AllObjects y las recorre, buscando que objeto que colisionan con PlayerShip y los guardan en un IEnumerable.

Solución Propuesta:

Propongo cambiar los métodos, en el caso de CheckForPowerUps(), voy a recorrer la lista de AllObjects, buscando y almacenando en un Array, los objetos que son de tipo PoweUp, y más adelante recorro el Array, para detectar si hay algún PowerUp que colisiona con PlayerShip

En el caso de CheckForCollision() aplique una optimización muy parecida, Voy a recorrer la lista de AllObjects almacenando en un Array los objetos que son de tipo EnemyShip e ignorando los objetos que sean Null, luego accedo a los elementos de este nuevo Array y detecto si algún EnemyShip colisiona con PlayerShip

Resolución:



Como podemos ver en la imagen, (en el cuadro amarillo), si la comparamos con las mediciones iniciales, el Update del PlayerShip paso de ocupar un 19,59% del CPU a solo el 0.06% del mismo. Esto se puede comprobar en la segunda medición (en los cuadros verdes), donde vemos que el tiempo de actualización tiene una media de 85ms, con picos de 196ms, cuando antes corría a una media de 110ms con picos de 244ms.

Hipotesis7

Mediciones:



Hipotesis:

Como se puede ver en la Medición, se encontró un bug en el cual todas las EnemyShips tienen la misma textura.

Revisando el código encontré el problema, Los EnemyShips llamaban a Loadimage() y obtenían la textura de nave verde y después se cargaba el shipIndex de la imagen.

Solución Propuesta:

Propongo un cambio muy simple, primero que se cargue el shipIndex, y luego se llame a LoadImage(), simplemente invirtiendo estas 2 líneas, busco lograr que la imagen de EnemyShip se cargue correctamente.

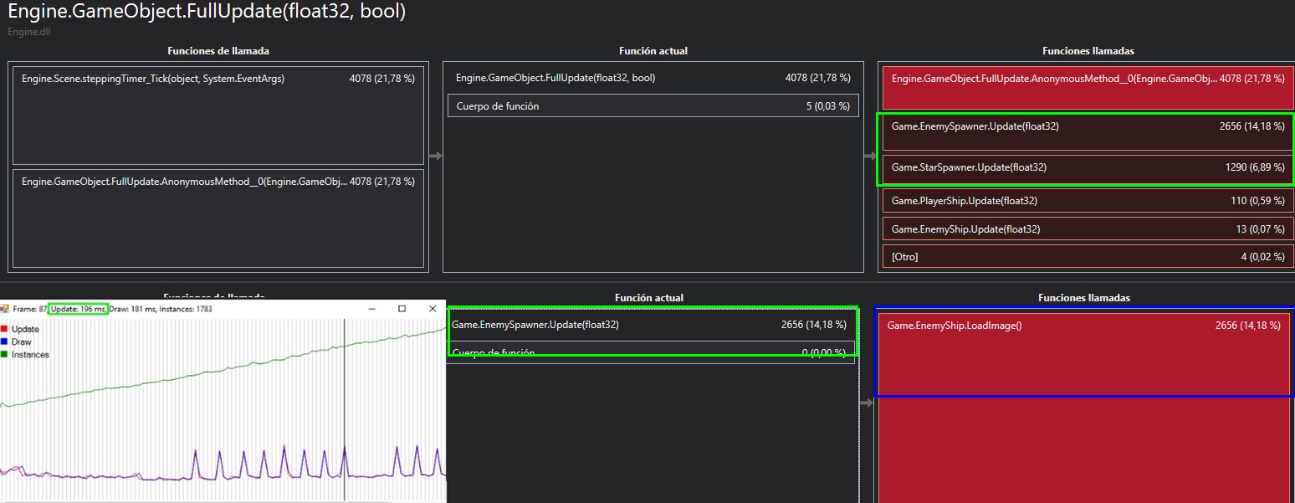
Resolución:

La implementacion fue positiva, las naves se dibujan correctamente.



Hipotesis8

Mediciones:



Hipotesis:

Revisando los cuadros verdes de las mediciones, encontramos el método FullUpdate() ocupa el 21,78% del CPU Total, dentro de este, encontré como principal problema la actualización del EnemySpawner, ya que ocupa el 14,18% del CPU Total, por otro lado, revisando el grafico encontramos picos de 196ms en el Update.

También podemos ver que el método EnemySpawner.Update() llama a LoadImage(), y en este, es donde se consume un 14,18% de los recursos.

Revisando el código encontré que, cada ves que un EnemySpawner genera un nuevo enemigo, este llama a LoadImage(), en este método es donde se recortan todas las Imágenes de las naves y son guardadas en un array. Esto es lo que genera picos de latencia en el Update, ya que el programa está creando de forma constantemente los nuevos enemigos.

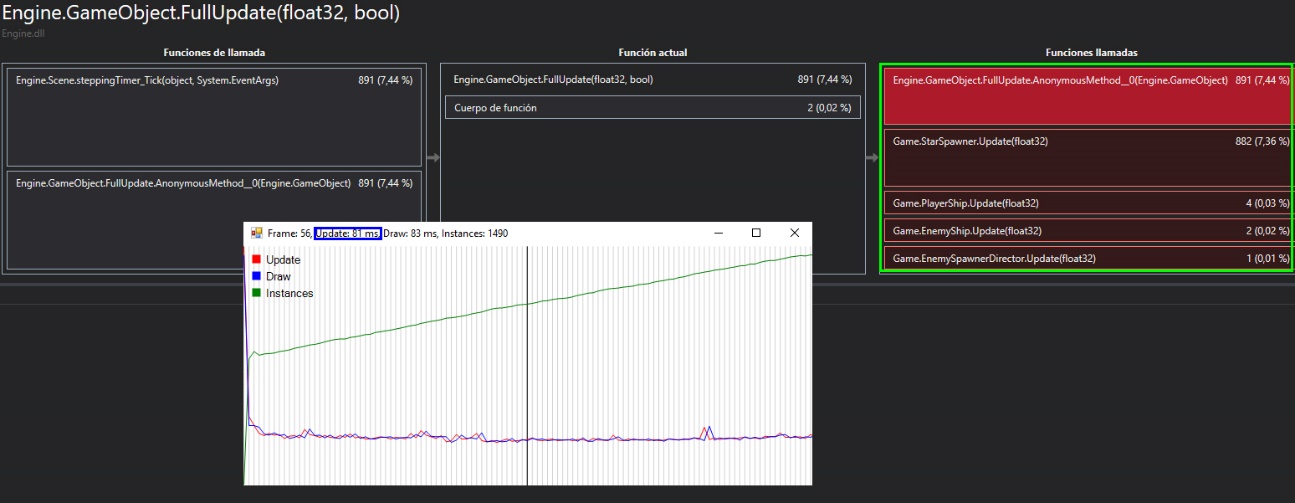
Solución Propuesta:

Mi propuesta es crear un Array global dentro de la clase Spritesheet, que contenga todas las imágenes de las naves ya recortadas y creadas.

De esta manera los EnemyShips pueden acceder a este Array y cargar sus imágenes de forma directa.

Por último voy a mencionar de que manera se carga este Array de Imágenes: Cuando creo mi primer nave PlayerShip, y recorto las imágenes por primera vez, se guardan en el Array, para que los EnemiShips puedan acceder.

Resolución:



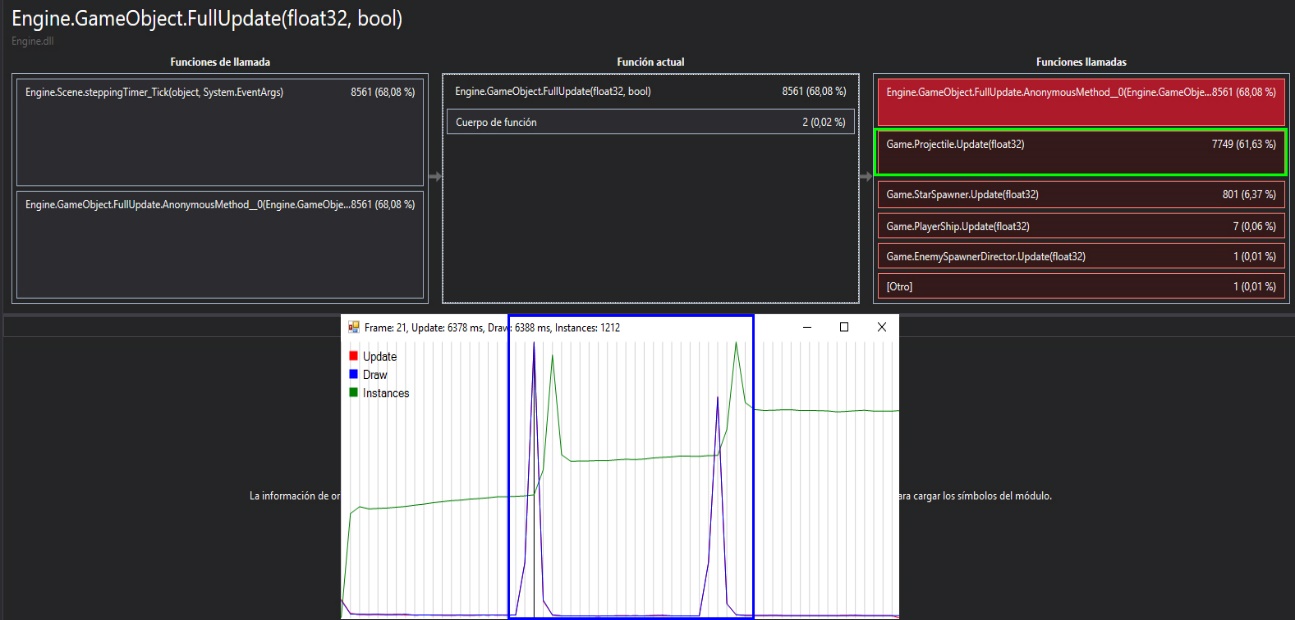
Mirando el cuadro verde de la resolución, y lo comparamos con las mediciones iniciales, podemos ver que la Función Game.EnemySpawner.Update() ya no aparece, paso de ocupar el 14,18% al 0,01% del CPU Total.

En el cuadro azul, podemos ver que el juego corre con una media de 80-90ms y ya no tiene los picos de ms en el update, como ocurría en las imágenes de las mediciones.

Los cambios fueron positivos.

Hipotesis9

Mediciones:



Hipotesis:

Al momento de disparar un par de Projectile el programa queda pensando y tarda mucho en actualizar.

Como se ve en el cuadro verde, en el Game.Proyectile.Update(), encontramos que se consume el 61,63% del tiempo del CPU.

En el cuadro azul del gráfico, podemos ver que hay picos de 6378ms en el Update y el Draw.

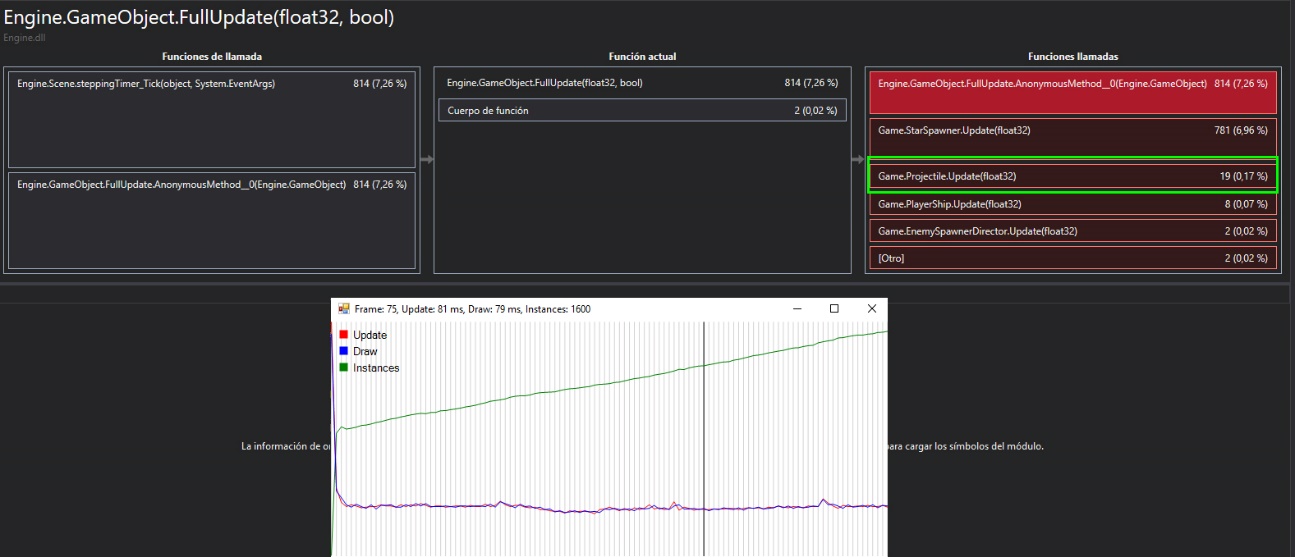
Revisando el código, pude notar que el problema de los proyectiles era el mismo de la Hipotesis6. Projectile tiene el método CheckForCollision(), este agarran la lista de AllObjects y la recorre, calculando posibles colisiones buscando EnemyShips, por otra parte, los Projectile que no alcanzan a ningún enemigo, quedan dispersos en el mundo y no se liberar nunca de la memoria.

Solución Propuesta:

La solución que propongo es cambiar el método CheckForCollision() del proyectil para que recorra la lista de AllObjects almacene en un Array los objetos que son de tipo EnemyShip e ignorar el resto de objetos, luego accedo a todos los enemigos y detecto si algún Proyectil colisiona con EnemyShip.

Por otro lado voy a agregar un IF en el Update() de los Projectile, para borrarlos de memoria si salen del límite del mundo.

Resolución:

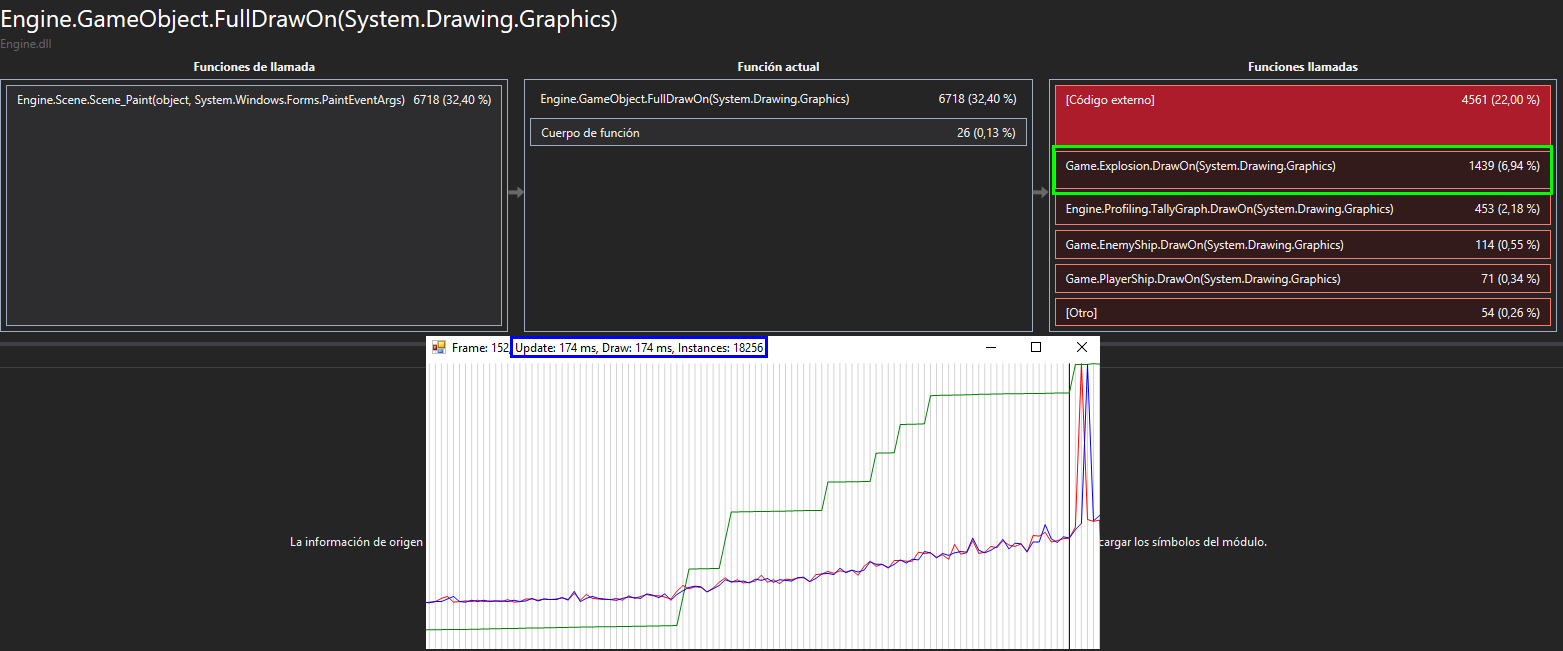


Tras disparar varios Projectile y comparando el nuevo gráfico, vemos que el problema se solucionó.

Si miramos el cuadro verde podemos notar que Game.Proyectile.Update(),paso de consumir el 61,63% a 0,17% del tiempo del CPU.

Hipotesis10

Mediciones:



Hipotesis:

Al momento de disparar proyectiles y eliminar EnemyShips, se generan explosiones con muchas partículas en pantalla y causa ralentización en el programa.

Revisando las mediciones vemos que en el cuadro verde el Game.Explosion.DrawOn() Consume un 6,94% de los recursos, por otro lado en el cuadro azul vemos que la cantidad de instancias aumentan exponencialmente cada vez que se genera una nueva explosión.

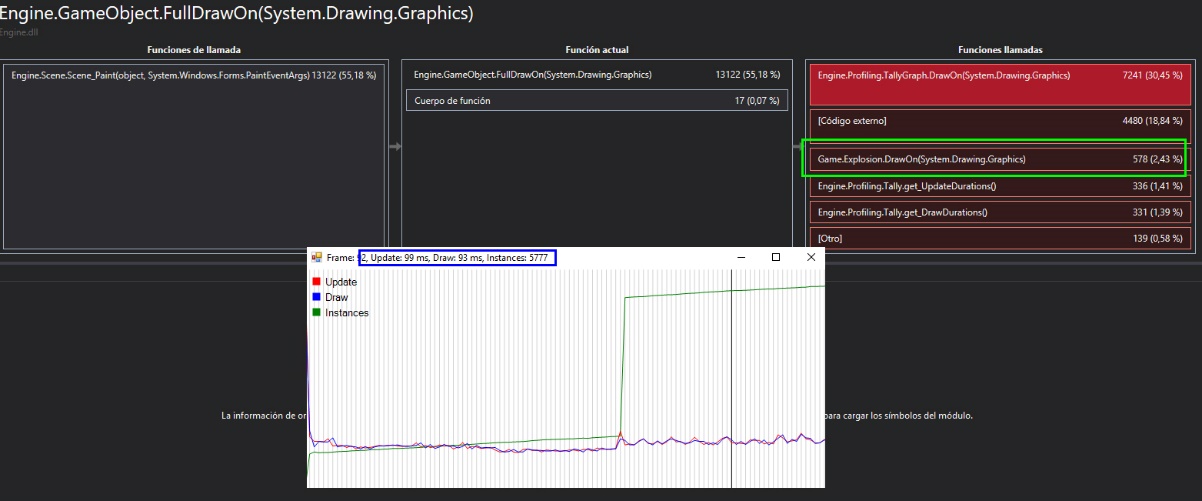
Solución Propuesta:

Propongo crear 2 Listas que almacenen las partículas de las explosiones, cuando una lista se llena, se libera la primera para añadir la nueva explosión, de esta manera le da tiempo a ejecutar las 2 explosiones y controlo el límite de instancias y objetos en el mundo.

Por otro lado, voy a crear una Matriz que guarde los Brushes utilizados para crear las partículas, porque en cada explosión se generan muchos pens y brushes nuevos de forma aleatoria

Para controlar las instancias también voy a aplicar un Delete() a los EnemyShip cuando sales de pantalla

Resolución:



Como se ve en el cuadro verde de las nuevas mediciones, el dibujado paso de consumir 6,94% de CPU a solo 2,43%.

Revisando el cuadro azul, vemos que las instancias se mantienen alrededor de 5000-6000 y ya no crece constantemente como ocurría antes.

Los cambios fueron positivos.