Universidad Complutense de Madrid
Facultad de Informática

Proyecto de Sistemas Informáticos 2009/2010

Sistema de Visión Estereoscópica para Navegación Autónoma de vehículos no tripulados

por

Daniel Martín Carabias
Raúl Requero García
José Andrés Rodríguez Salor

Profesor director: Gonzalo Pajares Martinsanz

Madrid, 2010

Se autoriza a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado.
Firmado:

Resumen

La visión estereoscópica artificial es un campo muy amplio que forma parte de lo que se conoce como visión por computador. Técnicamente consiste en el procesamiento de dos imágenes obtenidas mediante sendas cámaras, a partir de una escena tridimensional 3D. Este procesamiento está orientado a reconstruir la escena en 3D a partir de las dos imágenes, izquierda y derecha. Un aspecto que cabe destacar es que las cámaras están desplazadas una cierta distancia, tal y como ocurre con nuestros ojos. El trabajo del computador consiste en identificar en ambas imágenes aquellos píxeles en las dos imágenes que se corresponden con la misma entidad física en la escena 3D, usando para ello algoritmos especializados. La distancia que separa estos píxeles se conoce como disparidad. La medida de la disparidad sirve para obtener la distancia a la que se sitúa físicamente ese objeto en la escena con respecto a las dos cámaras.

La visión estereoscópica es un campo que tiene numerosas aplicaciones y en el que a día de hoy se están invirtiendo numerosos recursos en investigación. Concretamente, una de esas aplicaciones es la detección de obstáculos por parte de robots. Nuestro proyecto está orientado hacia esa aplicación práctica, si bien se centra exclusivamente en los aspectos relacionados con la correspondencia de los elementos homólogos en las imágenes. Para ello hemos implementado diversas técnicas y algoritmos de visión estereoscópica usando el lenguaje de programación C#.

Palabras clave: visión estéreo, estereopsis, visión por computador, cámaras, escena 3D, disparidad, C#, tratamiento de imágenes, anaglifo, robótica

Abstract

Stereo vision is a broad field that is part of computer vision. Technically, it consists of the processing of two images adquired by two cameras, from a given scenario. This processing is aimed to reconstruct the 3D scene from both images, namely left and right images. One thing that is worth mentioning is that the two cameras are shifted a certain distance, as it happens with our eyes. The computer basically identifies in both images those pixels that match, using specialized algorithms. The distance that separates those pixels is known as disparity. Disparity is next used in the calculation of the distance between the object in the scene and the cameras.

Stereo vision (also known as stereopsis) is a field with multiple applications and in which it is invested many resources in research. One of those applications is the detection of obstacles by robots. Our project is oriented towards this practical application; although this work is focused only on the computation of the disparities, i.e. the correspondence between pixels in the images. We have implemented several stereo vision techniques and algorithms using the C# programming language.

Keywords: stereo vision, stereopsis, computer vision, cameras, 3D image, robotics, disparity, C#, image processing, anaglyph

Índice general

1 Introducción 3
1.1 Motivación del proyecto . . . 3
1.2 Objetivo del proyecto . . . 3
1.3 Organización de la memoria . . . 3

2 Conceptos teóricos 5
2.1 Introducción . . . 5
2.2 Geometría del sistema estereoscópico . . . 5
2.3 Algoritmo estándar de correlación . . . 6
2.4 Algoritmo estéreo de Lankton . . . 7

3 Diseño técnico del proyecto 9
3.1 Introducción . . . 9
3.2 Organización del diseño . . . 10
3.2.1 Diagramas de clase . . . 10
3.2.2 Principales diagramas de secuencia del algoritmo de correlación . . . 12
3.2.3 Principales diagramas de secuencia del algoritmo de Lankton . . . 14

4 Servicios y tecnologías utilizados 19
4.1 .NET Framework . . . 19
4.2 AForge.NET . . . 20
4.3 Windows Presentation Foundation (WPF) . . . 20
4.3.1 Windows Forms . . . 20
4.3.2 ¿Por qué Windows Presentation Foundation? . . . 20
4.3.3 Construir interfaces gráficas en WPF mediante Expression Blend . . . 21

5 Manual de usuario 23

6 Proceso de desarrollo 31
6.1 Análisis previo en MATLAB . . . 31
6.2 Desarrollo en C# del algoritmo estándar . . . 31
6.3 Desarrollo de un nuevo algoritmo: Lankton . . . 32
6.4 Búsqueda y desarrollo de una nueva versión del algoritmo de Lankton . . . 32
6.5 Problemas de rendimiento en la implementación en C# . . . 32
6.6 Pruebas con imágenes reales . . . 34
6.7 Detección de objetos en la imagen . . . 34
6.8 Integración con el simulador de robot . . . 35

7 Resultados y conclusiones **37**
7.1 Imágenes utilizadas 37
7.1.1 Imágenes de Middlebury 37
7.1.2 Imágenes de simulador 38
7.1.3 Imágenes reales 38
7.1.4 Imágenes reales, para detección de objetos 39
7.2 Resultados 40
7.2.1 Resultados con las imágenes de Middlebury 40
7.2.2 Resultados con las imágenes de simulador 41
7.2.3 Resultados con las imágenes reales 45
7.2.4 Resultados obtenidos para la detección de objetos 47

A Diseño técnico completo. Diagramas de secuencia **49**
A.1 Diagramas del algoritmo de correlación 49
A.2 Diagramas del algoritmo de Lankton y de su versión mejorada 60
A.3 Diagramas de la clase BitMapFast 92
A.4 Diagramas de la clase Filtros 96
A.5 Diagramas de la clase Gradiente 116

B Interfaz de programación de la aplicación **123**
B.1 Espacio de nombres ProyectoSI 123
B.1.1 Clase ProyectoSI 123

Capítulo 1

Introducción

1.1 Motivación del proyecto

La visión estereoscópica artificial es un campo que tiene especial importancia en la robótica. Los robots que son enviados a otros planetas por las agencias espaciales tienen que dirigirse por un terreno que no está libre de obstáculos. Por lo tanto, estos robots, además de estar equipados por dos cámaras, deben incorporar un determinado software que permita reconstruir la escena en 3D a partir de cada par de imágenes que capturen.

Ese software no solamente debe reconstruir la escena 3D con la mayor precisión posible, es importante también que lo haga en el menor tiempo posible, con el fin de que las decisiones se tomen de manera precisa y ajustada en el tiempo.

Esta aplicación se plantea también con el objetivo de integrarla en otro proyecto dentro de la asignatura Sistemas Informáticos enfocado a la navegación de un robot. Nuestro proyecto proporcionaría la información suficiente para que el robot determine la presencia de obstáculos en una determinada posición y a una determinada distancia, y por tanto pueda modificar su trayectoria para eludirlo.

1.2 Objetivo del proyecto

En visión estereoscópica artificial existe una gran cantidad de algoritmos, siendo algunos más eficientes que otros en términos de precisión y tiempo de respuesta. El objetivo principal de este proyecto es desarrollar una aplicación que obtenga la representación 3D de una escena a partir de dos imágenes tomadas mediante cámaras. Para ello hemos analizado los algoritmos de visión en estéreo más prometedores y hemos elegido e implementado aquellos que nos han parecido más apropiados. A partir de la información que proporcionen estos algoritmos se podría ayudar a planificar la ruta de un robot autónomo.

1.3 Organización de la memoria

Esta memoria está organizada en los siguientes capítulos. En el capítulo segundo se analizan los conceptos teóricos en los que está basado el prototipo que hemos construido. En el capítulo tercero se detalla el diseño y la arquitectura general del prototipo construido. En el capítulo cuarto recogemos los servicios y las tecnologías que hemos usado para construir nuestro proyecto, así como una explicación de por qué se han elegido. El capítulo quinto recoge un breve manual de usuario para conocer mejor cómo se usa la aplicación. El capítulo sexto explica qué proceso hemos seguido para construir nuestro

proyecto así como los problemas con los que nos hemos encontrado. El capítulo séptimo detalla los resultados que hemos obtenido probando nuestra aplicación con diferentes imágenes, así como las conclusiones que hemos extraído. Para finalizar incluimos un par de apéndices con más información técnica sobre el diseño del proyecto (diagramas de comportamiento y resumen de la interfaz de programación accesible por el usuario).

Capítulo 2

Conceptos teóricos

2.1 Introducción

Esta sección introductoria pretende abordar los conceptos básicos de visión estereoscópica en los que está basado el prototipo construido. Nuestra referencia principal sobre las generalidades del proceso de visión en estéreo ha sido el texto [PDIC07, cap. 11].

La visión estereoscópica es uno de los procedimientos que existen para obtener información acerca de la estructura tridimensional de una escena y por tanto de las distancias a los objetos que estén situados en ella. Esto es útil en muchos y diversos campos de la tecnología, como la robótica, en la que un robot móvil debe disponer de información precisa sobre el entorno que le rodea para poder operar sin riesgos.

La obtención por parte de un computador de las mencionadas distancias no es en absoluto un procedimiento directo. Requiere como paso previo la obtención de la separación existente entre las dos cámaras, la distancia focal de las lentes y la separación relativa de las proyecciones en los objetos en las imágenes. En este punto se toma como referencia el proceso de visión estereoscópica biológico: los dos ojos de un ser humano están separados una cierta distancia, por lo que las imágenes de un mismo objeto tomadas por cada uno ellos se proyectan sobre las retinas en posiciones diferentes. Basándose en este hecho podemos reconstruir la escena 3D y tener cierta noción de profundidad aplicando un procedimiento de *triangulación geométrica* entre ambas imágenes, izquierda y derecha, que constituyen el par estereoscópico.

2.2 Geometría del sistema estereoscópico

Partiremos de un sistema formado por dos cámaras cuyos ejes ópticos son mutuamente paralelos, encontrándose separadas una distancia que se denomina *línea base*. Un concepto que surge del hecho de tener más de una cámara es el de línea epipolar, tratado a continuación:

Definición: Tal y como se observa en la imagen 2.1, sean dos imágenes de un mismo objeto, I_1 e I_2, tomadas por dos cámaras diferentes. Dado un punto, P_1 de I_1, su correspondencia en I_2, P_2, pertenece necesariamente a una línea recta completamente determinada por los puntos P_1 y P_2 que se denomina *línea epipolar*.

En este sistema, los ejes ópticos de las dos cámaras están separados únicamente en una dimensión, la horizontal, por lo que un punto de la escena captado por las dos cámaras va a diferir exclusivamente en su componente horizontal.

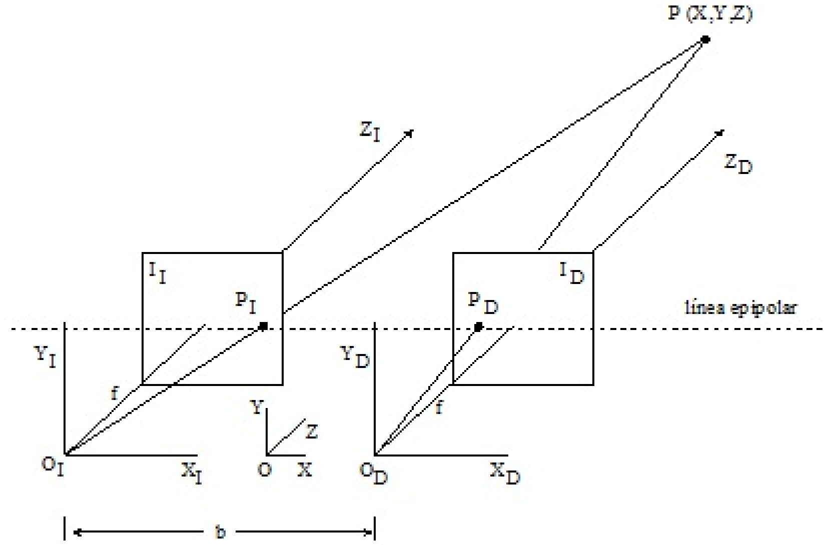

Figura 2.1: Geometría de un par de cámaras con los ejes ópticos paralelos

El origen del sistema de coordenadas es O, siendo la longitud focal efectiva f, y la línea base, b. En cada una de las dos cámaras se establece un sistema de coordenadas 3D (X, Y, Z), por lo que $P_1(X_I, Y_I)$ y $P_2(X_D, Y_D)$ son las proyecciones en las imágenes izquierda y derecha, respectivamente, de un punto P(X, Y, Z) de la escena.

Los rayos de proyección PO_I y PO_D definen un plano de proyección del punto en la escena 3D que recibe el nombre de *plano epipolar*. Como consecuencia de la geometría del sistema se obtiene la denominada *restricción epipolar*, que limita la búsqueda de correspondencia de manera que en el sistema estándar de ejes paralelos todos los planos epipolares generan líneas horizontales al cortarse con los planos de las imágenes.

En un sistema como el anterior definimos la *disparidad* de un par de puntos emparejados, esto es, $P_I(X_I, Y_I)$ y $P_D(X_D, Y_D)$ como X_I - X_D.

Utilizando el sistema anterior, podemos obtener, a partir de la geometría del mismo, las siguientes ecuaciones:

$$\left.\begin{array}{l} O_I : \dfrac{\frac{b}{2}+X}{Z} = \dfrac{X_I}{f} \\ O_D : \dfrac{\frac{b}{2}-X}{Z} = \dfrac{X_D}{f} \end{array}\right\} \Rightarrow \left.\begin{array}{c} X_I = \dfrac{f}{Z}(X+\dfrac{b}{2}) \\ X_D = \dfrac{f}{Z}(X-\dfrac{b}{2}) \end{array}\right\} \Rightarrow \left. d = X_I - X_D \quad = \dfrac{f*b}{Z}\right\} \Rightarrow Z = \dfrac{f*b}{d} \tag{2.1}$$

2.3 Algoritmo estándar de correlación

El algoritmo de correlación se basa en el siguiente hecho: Para establecer la correspondencia de un píxel de la imagen izquierda en la imagen derecha se recorre la línea epipolar que pasa por él y se escoge el píxel que más se le parezca, a partir de sus valores de intensidad. Para aumentar la robustez de este proceso, se utiliza un entorno de vecindad alrededor del píxel (ventana).

2.4 Algoritmo estéreo de Lankton

Ahora nos planteamos como objetivo el estudio para su posterior diseño de uno de los algoritmos estéreo que según el análisis de Middlebury[1] ofrece mejores resultados. Este algoritmo, descrito en [Lan07] es una simplificación del propuesto en [KSK06]. Dicho algoritmo aporta como novedad que en lugar de calcular disparidades para píxeles individuales, se calculan planos de disparidad para cada uno de los segmentos de color que se pueden extraer de una de las dos imágenes. La obtención del plano de disparidad óptimo se basa en un algoritmo de propagación de confianza, basada en la obtención de los planos de color.

Un algoritmo de propagación de confianza (en inglés, *belief propagation*) consigue reducir la complejidad computacional de un problema aprovechando la estructura gráfica del mismo. Es por tanto un método bayesiano que se usa especialmente en inteligencia artificial y en teoría de la información.

El primer paso del algoritmo propuesto por Klaus *y col.* consiste en obtener un plano de segmentos de color a partir de la escena. Una reflexión minuciosa del concepto de disparidad nos lleva a la conclusión de que la disparidad no varía demasiado dentro de cada una de las regiones. Los cambios abruptos de disparidad (y por tanto, de distancia), se producen en los cambios de una región a otra.

La correspondencia entre píxeles es uno de los pasos centrales de cualquier algoritmo de visión en estéreo, tal y como se ha comentado anteriormente. Klaus *y col.* utilizan como medida de correspondencia entre píxeles una suma ponderada de las diferencias en valor absoluto de intensidad (en adelante haremos referencia a este concepto como CSAD) y una medida basada en el gradiente de la imagen (en adelante, CGRAD). A continuación definimos el concepto de gradiente, que resulta familiar en el contexto de la Física, pero que puede resultar extraño a aquellos lectores que no tengan una base en tratamiento digital de imágenes.

Definición: Se define el *gradiente* de una imagen expresada como una función de dos variables, f(x, y), en un punto dado (x, y), como un vector de dos dimensiones, perpendicular al punto donde se calcula, generalmente de borde, y dado por la ecuación

$$\vec{G}[f(x,y)] = \begin{pmatrix} G_x \\ G_y \end{pmatrix} = \begin{pmatrix} \frac{\partial f(x,y)}{\partial x} \\ \frac{\partial f(x,y)}{\partial y} \end{pmatrix} \tag{2.2}$$

El vector G posee una magnitud y dirección dadas por

$$|G| = \sqrt{G_x^2 + G_y^2} \tag{2.3}$$

$$\phi(x,y) = \arctan \frac{G_y}{G_x} \tag{2.4}$$

Una forma de calcular el gradiente expresado en la ecuación (2.2) es aplicar de manera directa la definición matemática de derivada:

$$G_x = \frac{f(x+\Delta x) - f(x-\Delta x)}{2 * \Delta x} \tag{2.5}$$

$$G_y = \frac{f(y+\Delta y) - f(y-\Delta y)}{2 * \Delta y} \tag{2.6}$$

Los parámetros CSAD y CGRAD se definen de la siguiente manera:

$$CSAD(x,y,d) = \sum_{i,j \in N(x,y)} I_1(i,j) - I_2(i+d,j) \tag{2.7}$$

[1] http://vision.middlebury.edu/stereo/

$$CGRAD(x,y,d) = \sum_{i,j \in N_x(x,y)} |\Delta x I_1(i,j) - \Delta x I_2(i+d,j)| + \sum_{i,j \in N_y(x,y)} |\Delta y I_1(i,j) - \Delta y I_2(i+d,j)| \quad (2.8)$$

N(x, y) representa una ventana de tamaño 3x3 centrada en el punto de posición (x, y). En caso de que las imágenes sean en color, se realiza la suma sobre cada una de sus tres componentes (rojo, verde y azul).

La medida de disparidad pondera los valores de CSAD y CGRAD multiplicando a uno de ellos por un peso w y al otro por su complemento en uno, de esta forma se obtiene:

$$C(x,y,d) = (1-w) * CSAD(x,y,d) + w * CGRAD(x,y,d) \quad (2.9)$$

Por último, la calidad del resultado se optimiza filtrando aquellos píxeles que no proporcionan una confianza alta. Para ello se realiza el cálculo de disparidades desplazando la imagen izquierda sobre la derecha y viceversa. También se incluye un proceso final de tipo *winner-take-all*, que viene a sintetizarse como elegir de todos los candidatos posibles el mejor de todos los desplazamientos realizados.

Capítulo 3

Diseño técnico del proyecto

3.1 Introducción

Una vez establecido el objetivo y elegido el método de correspondencia que ha de aplicarse, se hace necesario definir el diseño bajo el cual se va a proceder a la implementación del mismo.

El análisis y diseño de aplicaciones informáticas debe abordarse con técnicas y metodologías adecuadas, acompañadas por una precisa gestión de proyectos y una eficaz gestión de la calidad. También es importante poder contar con el soporte de entornos y herramientas adecuadas, que faciliten la tarea del profesional informático y de los usuarios a la hora de desarrollar sistemas de información.

Además, una buena gestión de información es fundamental en las empresas. Es por ello por lo que el desarrollo de sistemas de información se ve sometido actualmente a grandes exigencias en cuanto a productividad y calidad, y se hace necesaria la aplicación de un nuevo enfoque en la producción del software, más cercano a una disciplina de ingeniería que a los hábitos y modos artesanales que por desgracia se han venido aplicando en más de una ocasión con sus correspondientes lamentables consecuencias.

Para el diseño, se ha elegido la metodología UML, que permite modelar (analizar y diseñar) sistemas orientados a objetos como el elegido en nuestro caso. Nos ahorraremos los detalles históricos acerca de este lenguaje y los conceptos de ingeniería del software que se suelen emplear, dado que pueden encontrarse en numerosos libros [PI05].

De la gran variedad de herramientas que nos ofrece el lenguaje UML, se han elegido dos tipos de diagramas: diagramas de clase y diagramas de secuencia. Los diagramas de clase son un tipo de diagramas estáticos que describen la estructura de un sistema mostrando sus clases, atributos y las relaciones entre éstos. En ellos, aparecen las propiedades, también llamadas atributos o características, que son valores que corresponden a un objeto. Además aparecen las operaciones, comúnmente conocidas como métodos, y son aquellas actividades que se pueden realizar con o para dicho objeto.

Los diagramas de secuencia muestran la interacción de un conjunto de objetos en una aplicación a través del tiempo y se modelan para cada método de la clase. Los pasos se estructuran cronológicamente desde la parte superior del diagrama a la parte inferior; la distribución horizontal de los objetos es arbitraria.

De manera ideal, el diseño técnico de cualquier aplicación debe hacerse previamente a la implementación, pero, como sucede en cualquier tipo de proyecto, hemos ido desarrollando ambas partes en paralelo haciendo los ajustes pertinentes en el diseño técnico para que la correspondencia de ambas fuese lo más fiable posible.

3.2 Organización del diseño

La aplicación está organizada en diferentes clases, cada una de las cuales contiene las distintas funciones que componen el proyecto en su conjunto. Dichas clases han sido diseñadas en forma de diagramas de secuencia y de clases, los cuales se han ido modificando a medida que ha surgido la necesidad.

A continuación, vamos a hacer una descripción que facilitará la lectura de los diagramas en cuestión, así como las imágenes de dichos diagramas. Para evitar una lectura monótona, se presentarán en este capítulo los principales diagramas de la aplicación. En el apéndice primero se muestra el diseño detallado completo, incluyendo los correspondientes diagramas de clase y de secuencia.

Dada la extensión de algunos diagramas, se dividirán en varias páginas para poder visualizarlos a un tamaño aceptable, siendo el orden de presentación desde la parte superior del diagrama a la inferior. Cuando los diagramas desbordan el ancho de la página también se dividen siguiendo el orden de izquierda a derecha. Por ejemplo, si tenemos un diagrama de grandes dimensiones que hemos de dividir en cuatro partes, en las siguientes páginas se mostrará de esta forma: primera página: parte superior izquierda del diagrama; segunda página: parte superior derecha del diagrama; tercera página: parte inferior izquierda del diagrama; cuarta página: parte inferior derecha del diagrama.

3.2.1 Diagramas de clase

En primer lugar, los diagramas de clase que se muestran a continuación (figura 3.1 en la que podemos ver las clases *AlgoritmoEstereo, Anaglifo, Lankton y Lankton2* con sus respectivas funciones asociadas) no contienen las relaciones entre ellos por simplicidad. A pesar de esto, si se desean ver las relaciones, es más recomendable hacerlo sobre los diagramas de secuencia ya que en ellos, en cada método mostrado, existen referencias a otros diagramas de las distintas clases, y es ahí donde se contemplan qué dependencias existen con exactitud.

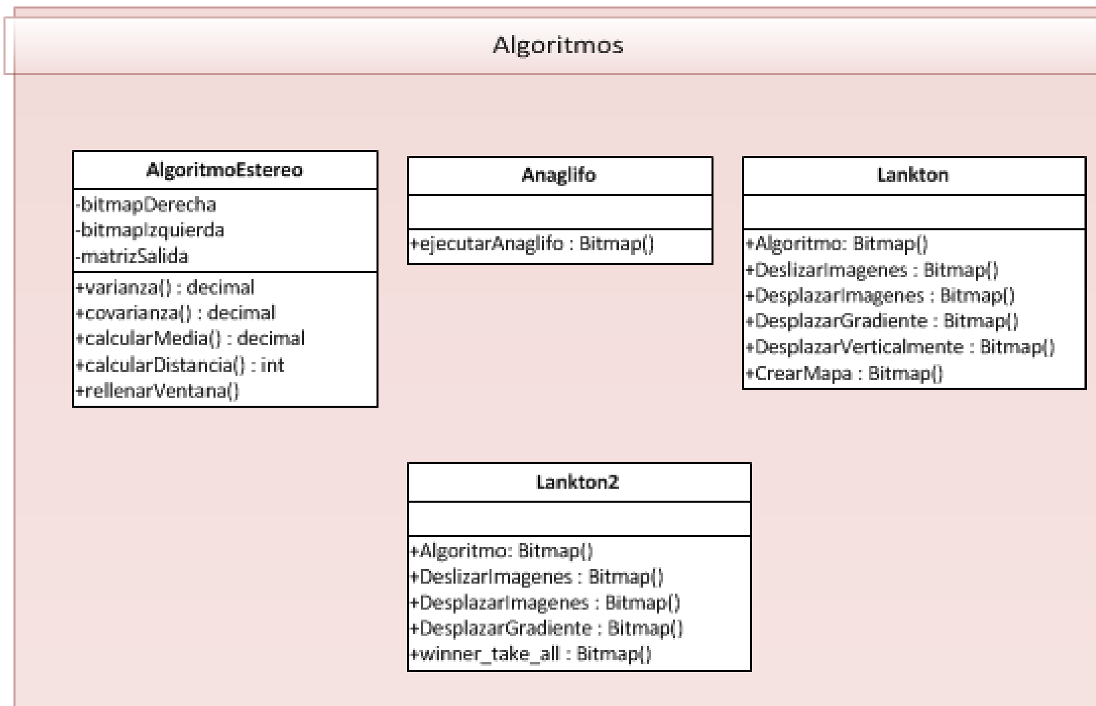

Figura 3.1: Diagrama de las principales clases de la aplicación.

Además de las clases mostradas en el diagrama anterior, tenemos otras clases más específicas (que podemos ver en la figura 3.2, en la que se muestran las clases *Filtros* y *Gradiente* con sus respectivas funciones asociadas) que intervienen en la implementación de los algoritmos, así como estructuras de datos adaptadas a nuestras necesidades (figura 3.3 en la que podemos ver las clases *ContenedorBitmapDisparidad*, *BitmapFast*, *DatosXML*, *ParametrosAlgoritmoEstandar*, *Parametros-DeteccionObjetos* y *ParametrosAlgoritmoLankton* con sus respectivas funciones asociadas).

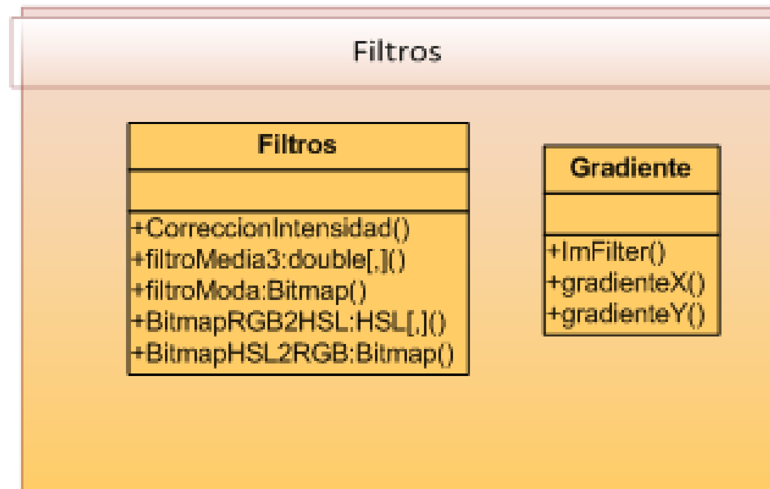

Figura 3.2: Diagrama de las clases referidas a los filtros utilizados.

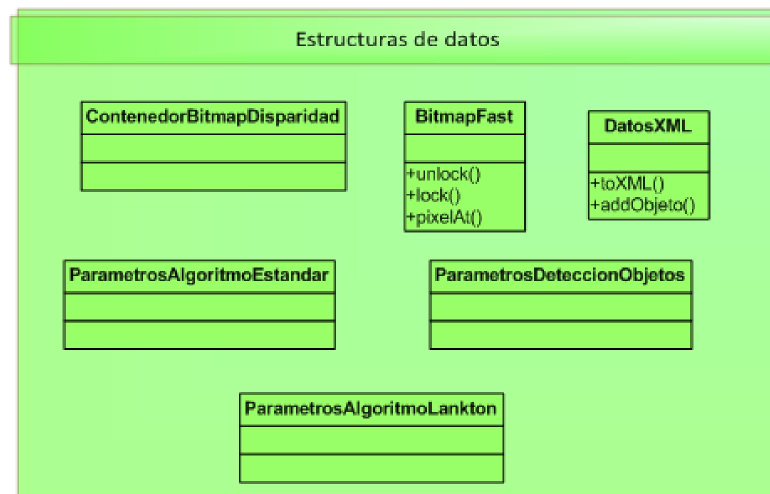

Figura 3.3: Diagrama de las clases referidas a las estructuras de datos utilizadas.

3.2.2 Principales diagramas de secuencia del algoritmo de correlación

La introducción teórica al algoritmo estándar de correlación se encuentra en el capítulo 2 (2.3).

Dicho algoritmo está implementado en una clase que hemos denominado *AlgoritmoEstereo*. Esta clase contiene una serie de funciones que permiten ejecutarle.

La función principal se llama *Algoritmo* A.1, y es ésta la que utiliza el resto de funciones.

Explicación del diagrama principal del algoritmo de correlación

Para comenzar, el algoritmo toma las dos imágenes de entrada que se van a procesar en forma de *BitmapFast*. Se eligió esta clase *BitmapFast*(implementada por nosotros con accesos directos a memoria) en lugar de *Bitmap* porque el procesamiento de las imágenes mejoraba considerablemente

en tiempo.

A continuación, recorreremos la imagen izquierda de izquierda a derecha y de arriba hacia abajo e iremos rellenando las sucesivas ventanas escogidas con los valores de intensidad de los píxeles correspondientes a la posición de la imagen en la que nos encontramos. El tamaño de la ventana dependerá de la elección del usuario.

Una vez rellenada la ventana, calculamos las distancias entre píxeles teniendo en cuenta dicha ventana y la distancia con respecto a la imagen derecha. Dada la complejidad de este cálculo, se explicará más adelante en otra subsección.

Una vez obtenidas las distancias de los píxeles de una imagen respecto a la otra, se almacenan en una matriz de salida (que será nuestra matriz de disparidad).

Tanto al principio como al final de la ejecución, se muestra la fecha y la hora con el fin de comprobar el tiempo de ejecución del algoritmo.

Se devuelve como resultado la matriz de disparidad.

Una vez definida la función *Algoritmo* A.1, podemos pasar a describir las funciones implicadas con mayor detalle, de las cuales, las más complejas son *calcularDistancia* A.5 y *rellenarVentana* A.3. En primer lugar, veamos *rellenarVentana* A.3.

Explicación del diagrama rellenarVentana

Tras la declaración de la matriz que constituirá nuestra ventana, comprobamos los límites de la misma que vamos a rellenar. Esta comprobación tiene como objetivo verificar que la imagen no se va a desbordar por exceder los límites de la misma mediante el paso de la ventana.

Si nos salimos de la ventana, devolvemos un valor nulo, en caso contrario nos guardaremos, según los valores que tengamos en la imagen, los píxeles con cada una de sus componentes espectrales (RGB) en nuestra ventana. Finalmente devolvemos la ventana con los valores de intensidad.

Por último, la función *calcularDistancia* (cuyo diagrama podemos ver en A.5), funciona de la siguiente manera:

Explicación del diagrama calcularDistancia

Tras la declaración de las variables iniciales y asignaciones de valores a las variables no inicializadas, se procede a realizar los cálculos que se detallan a continuación. En primer lugar, se calcula la media de los valores de la ventana izquierda (esto consiste simplemente en coger sus valores, sumarlos y dividir el resultado obtenido entre el número de valores de dicha ventana). Esta media se realiza sobre la componente roja.

A continuación, se calcula la varianza izquierda, cuyo cálculo consiste en, para cada valor de la ventana (también sobre la componente roja), restar a dicho valor la media y elevar esto al cuadrado. Se acumula la suma de estos resultados obtenidos y se divide entre el tamaño de la ventana. La siguiente fórmula resume esta operación:

$$\sigma^2 = \sum_{i=1}^{M} \sum_{j=1}^{N} \frac{(I_k(i,j) - \mu_k)^2}{M * N} \tag{3.1}$$

Debido a la geometría del sistema estereoscópico, un mismo objeto en la escena tridimensional se proyecta más a la izquierda en la imagen derecha que en la izquierda. Debido a este fenómeno, recorremos hacia la izquierda la imagen derecha y vamos rellenando ventanas sucesivamente. En caso de que las ventanas no sean nulas, se calcula su media, su covarianza y su varianza.

Para el cálculo de la covarianza vamos tomando la componente roja de la ventana izquierda y le restamos la media izquierda. Por otra parte tomamos la componente roja de la ventana derecha y le restamos la media derecha. Multiplicamos ambos valores y añadimos el valor a una variable entera que

acumule el resultado de sumar todos esos productos que vamos construyendo mientras recorremos las ventanas. El resultado que devuelve la covarianza es el obtenido tras dividir el valor acumulado anteriormente por el cuadrado del tamaño de la ventana.

Finalmente, obtenemos el valor de correlación dividiendo la covarianza entre el producto de las varianzas izquierda y derecha.

De los valores de correlación obtenidos, nos quedamos con el máximo, y el píxel que haya generado ese valor será el elegido como el correspondiente píxel de la imagen derecha que aparece en la imagen izquierda.

La distancia será la diferencia entre las posiciones de los píxeles de ambas imágenes.

3.2.3 Principales diagramas de secuencia del algoritmo de Lankton

El algoritmo de Lankton descrito en la sección 2.4 es un algoritmo de obtención de matrices de disparidad más completo que el de correlación. Este algoritmo ha sido implementado en la clase Lankton.

La idea fundamental de dicho algoritmo es que, tomando dos imágenes, se superpone una sobre la otra y se desliza horizontalmente. Para cada movimiento del deslizamiento, se obtiene una matriz de disparidad, y se escoge el mejor deslizamiento, entendiendo por tal el que proporcione como resultado una matriz de disparidad con el menor número de píxeles nulos, ya que estos píxeles proceden de decisiones en los que el algoritmo no ha encontrado correspondencia.

Como es posible que las cámaras no se encuentren bien calibradas, esta operación debe realizarse desplazando también verticalmente algunos píxeles las imágenes (los que el usuario determine). Para cada desplazamiento vertical hay que hacer el deslizamiento horizontal del que hemos hablado anteriormente. Dada la complejidad del algoritmo, inicialmente se obtuvo un diseño esquemático de los principales pasos que seguía el algoritmo de la figura 3.4. En la figura se puede ver un diseño simple cuyo objetivo era sencillamente que aplicase correctamente todos los pasos sobre las imágenes. Si bien tras su implementación y comprobada su ineficiencia, nos obligó a realizar un nuevo diseño de dicho algoritmo con el fin de obtener mejores resultados en lo que a tiempo de ejecución se refiere (figura 3.5).

A continuación explicamos el diagrama del algoritmo A.11, que es el principal diagrama del algoritmo de Lankton:

Explicación del diagrama principal del algoritmo de Lankton

Se toma como referencia temporal la fecha y hora actual con el fin de calcular el tiempo transcurrido en la ejecución.

A continuación, aplicamos un filtro corrector de intensidad, ya que es altamente probable en base a nuestra experiencia que las dos imágenes del par estereoscópico tengan distintos brillos y niveles de intensidad. Con este filtro se consiguen dos imágenes normalizadas a las que podemos aplicar nuestro algoritmo. La corrección se realiza siguiendo los siguientes pasos:

1. Las imágenes se convierten al formato de color HSL.
2. Se halla la media de la luminancia de cada una de las dos imágenes.
3. Se halla la desviación típica de la luminancia de cada una de las imágenes.
4. En la imagen que tenga la menor media aplicamos a la luminancia de cada píxel la siguiente fórmula:

$$píxel.L = píxel.L + offset (3.2)$$

$$offset = mediaMayor - g * mediaMenor \quad (3.3)$$

$$g = desviacionTipicaMenor/desviacionTipicaMayor \quad (3.4)$$

5. Se convierten la imágenes de nuevo a formato de color RGB.

En el bucle que podemos ver en el siguiente paso del diagrama, observamos que vamos haciendo una ejecución del algoritmo para un cierto número de desplazamientos verticales de las imágenes, ya que como hemos indicado previamente, es posible que las cámaras no estén perfectamente calibradas.

Tras desplazar verticalmente ambas imágenes lo especificado por el usuario, utilizamos hilos para acelerar la ejecución, ya que nuestro siguiente paso será deslizar la imagen derecha sobre la izquierda, y la izquierda sobre la derecha. Nos creamos copias de las imágenes para que ambas operaciones se puedan realizar simultáneamente, ya que son procesos independientes.

A continuación, obtenemos las mejores disparidades en las matrices de disparidad, y creamos el mapa de disparidad de cada desplazamiento vertical que guardaremos en un vector de soluciones.

Una vez que tenemos todas las matrices de disparidad, elegimos el mejor resultado de los que se encuentren en el vector de soluciones siendo éste el resultado del algoritmo.

A continuación se explica con mayor nivel de detalle las operaciones de las principales funciones del algoritmo de Lankton, como son las funciones *DesplazarVerticalmente* (A.14), *DeslizarImagenes* (A.15) y *elegirMejorResultado* (A.25).

1. *DesplazarVerticalmente* (A.14):

   Esta función simplemente consiste en mover todas las filas de la imagen a la fila anterior o posterior (según se quiera desplazar hacia arriba o hacia abajo) y añadir una fila de ceros en la última o primera fila (donde se haya quedado una fila vacía). Se realiza esto tantas veces como filas quieran desplazarse.

2. *DeslizarImagenes* (A.15):

   Los parámetros de entrada que recibe esta función son: los dos Bitmaps (el de la imagen izquierda y el de la imagen derecha), los enteros con decimales *mins* (disparidad mínima), *maxs* (disparidad máxima), *peso* (que es el parámetro *w* en la ecuación 2.9, el tamaño de la ventana y dos matrices de salida identificadas como *mindiff* (matriz de mínimas diferencias) y la matriz de disparidades (*matrizDisparidad*).

   Tras inicializar todas las variables que necesitaremos, rellenamos toda la matriz *h* (que sirve para aplicar el filtro de la media) con el valor de la inversa del tamaño de la ventana al cuadrado. A todos los elementos de la matriz *mindiff* se les asigna el valor infinito para poder calcular valores mínimos posteriormente, estableciendo comparaciones.

   A continuación creamos dos máscaras, hx y hy (que se corresponden con implementaciones específicas de los gradientes Gx y Gy definidos en las ecuaciones 2.5 y 2.6. Después elevamos al cuadrado cada uno de los cuatro resultados obtenidos (para cada imagen con cada máscara) y bloqueamos los cuatro bitmaps resultantes para realizar operaciones sobre ellos.

   Rellenamos cuatro matrices (g1X, g1Y, g2X, g2Y) de forma que, el elemento en la posición (i,j) de cada matriz se le asigna con la suma de los valores de las componentes RGB de cada uno de los bitmaps anteriores respectivamente. Estas cuatro matrices representan por lo tanto gradientes.

Figura 3.4: Diseño inicial del algoritmo de Lankton.

Una vez realizada esta operación, desbloqueamos los Bitmaps y vamos probando a desplazar horizontalmente la imagen que entra como derecha por parámetro, y desplazamos los gradientes g2X y g2Y.

Restamos la imagen original menos la desplazada elemento a elemento y, si alguna de las componentes RGB es menor que cero en algún píxel, la fijamos a cero. Posteriormente se procede a filtrar mediante la media los resultados previos. Este filtro tiene por objetivo eliminar píxeles espúreos con poca representatividad de una disparidad, obteniendo así una matriz de disparidad más homogénea.

El resultado será la mejor matriz de disparidades obtenida tras los distintos desplazamientos horizontales.

3. *elegirMejorResultado* (A.25):

   Esta función recibe como entrada el vector de bitmaps soluciones y un parámetro cantidad, que indica la cantidad de soluciones que tenemos.

   Recorremos el vector de soluciones hasta agotar las mismas, y para cada solución, contabilizamos el número de ceros que contiene su matriz de disparidad. El objetivo consiste en obtener el menor número de ceros posible, siendo este el mejor resultado que elijamos.

Finalmente, en la figura 3.5, en la parte izquierda podemos observar las mejoras introducidas con el filtro de corrección de intensidad introducido en la nueva versión del algoritmo de Lankton. En la derecha hemos mejorado la eficiencia de los deslizamientos de las imágenes mediante la ejecución con uso de hilos.

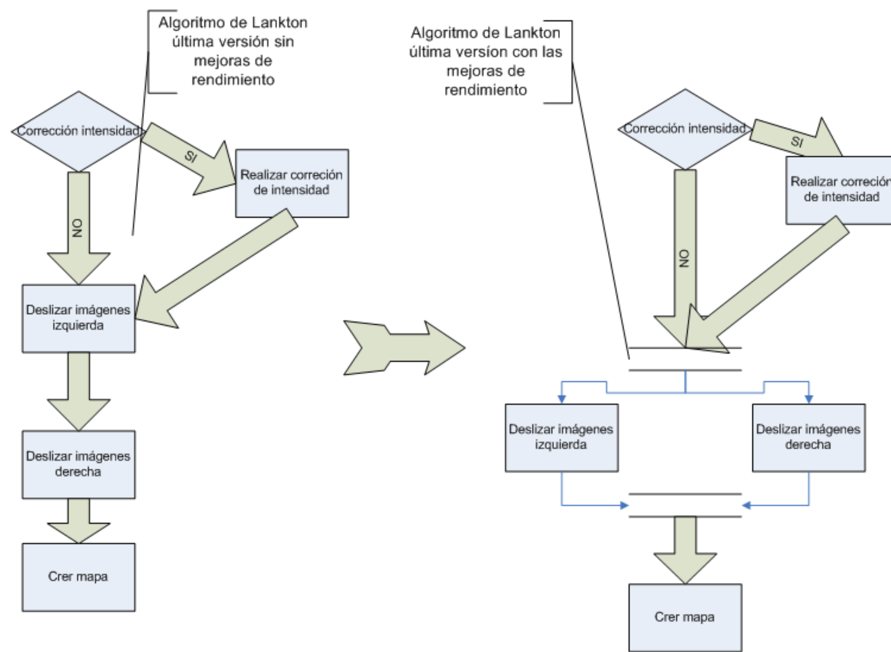

Figura 3.5: Diseño del algoritmo de Lankton sin optimizar (izquierda) y optimizado (derecha)

Capítulo 4

Servicios y tecnologías utilizados

En el capítulo anterior se detalló la arquitectura del prototipo que hemos construido. En este capítulo nos centraremos en los programas y tecnologías que nos han permitido integrar tanto nuestro código como el código de terceras personas que hemos usado bajo licencia.

4.1 .NET Framework

Para desarrollar nuestro proyecto hemos usado el lenguaje de programación C#, que es uno de los lenguajes capaces de utilizar el marco de trabajo .NET de Microsoft. C# es un lenguaje de tipo gestionado, como Java, lo que quiere decir que el compilador no genera instrucciones directamente ejecutables por el procesador. En su lugar genera un código en lenguaje intermedio denominado CIL (*Common Intermediate Language*) que es convertido a código nativo en tiempo de ejecución por un componente denominado compilador JIT (*Just In Time*).

El entorno de programación (IDE) que hemos usado para escribir código C# ha sido la edición Professional de Visual Studio 2008, disponible a través de la suscripción MSDN Academic Alliance a la que tenemos acceso como estudiantes de la Universidad Complutense. Visual Studio es un entorno similar al resto que se utilizan en otros lenguajes de programación (Eclipse, NetBeans, etc.). Ofrece el concepto de soluciones y de proyectos con el objetivo de organizar el código fuente y abstraer el proceso de compilación, que se realiza mediante el compilador C# de Microsoft, Csc.exe. También admite multitud de complementos o *plug-ins*, que hemos usado entre otras cosas para poder integrar un repositorio de código fuente en el desarrollo de nuestro proyecto con el fin de que cada miembro del grupo pudiera contribuir de manera eficiente.

Sintácticamente, el lenguaje C# resulta familiar a aquellas personas con un cierto manejo del lenguaje Java, el cual constituye una materia esencial en la práctica totalidad de las escuelas de Informática. Para el lector interesado, las especificaciones oficiales de este lenguaje en su última versión, la 4.0 (si bien en nuestro prototipo hemos usado una versión anterior, la 3.5) están disponibles en Internet en la siguiente dirección:

http://download.microsoft.com/download/0/B/D/0BDA894F-2CCD-4C2C-B5A7-4EB1171962E5 /CSharp%20Language%20Specification.htm

.NET Framework está compuesto por un conjunto de ensamblados (DLL) que engloban las diferentes funcionalidades más comunes requeridas por los proyectos de programación: Entrada/Salida, interfaces gráficas, estructuras de datos comunes, sincronismo y paso de mensajes, entre otras. Concretamente, para nuestro proyecto nos hemos centrado en las capacidades de tratamiento gráfico de .NET. El ensamblado System.Drawing proporciona diversas clases que abstraen algunos conceptos de tratamiento y representación de imágenes digitales. La clase Bitmap es la que representa una imagen

de mapa de bits en .NET Framework. La constructora de esta clase recibe como parámetro la imagen que queramos representar en memoria y ésta queda almacenada básicamente como una matriz de bits que representan una terna de colores rojo (R), verde (G) y azul (B). Dicha clase dispone de propiedades accesoras y mutadoras para acceder o modificar cada uno de sus píxeles.

4.2 AForge.NET

Durante el desarrollo del proyecto nos dimos cuenta de que necesitábamos realizar algunas operaciones sobre las imágenes repetidas veces, para las que .NET no ofrece una solución inmediata. Un ejemplo de esto es la conversión entre los sistemas de colores RGB y HSV. Por tanto, buscamos en Internet librerías gráficas para .NET cuya licencia nos permitiera integrarlas en nuestro proyecto. Finalmente la librería elegida fue AForge.NET.

AForge.NET es un proyecto dirigido a investigadores en las áreas de visión por computador, robótica e inteligencia artificial. Contiene una interfaz para el tratamiento de imágenes dentro del ensamblado AForge.Imaging.Filters. Nosotros hemos hecho uso de este ensamblado en nuestro proyecto.

El proyecto Aforge.NET está en constante evolución gracias a los esfuerzos de la comunidad de desarrolladores. Existe un repositorio público en la siguiente dirección de Internet:

http://code.google.com/p/aforge/

4.3 Windows Presentation Foundation (WPF)

Pensamos que en la informática actual, todo proyecto software además de cumplir con una serie de requisitos en cuanto a su calidad, debe ser amigable y en cierta medida vistoso para el usuario. Es por ello que hemos hecho uso de una tecnología de Microsoft para crear interfaces gráficas de última generación, Windows Presentation Foundation (WPF).

Partiendo de código C# actualmente hay disponibles dos paradigmas para crear interfaces gráficas: Windows Forms y Windows Presentation Foundation (WPF). A continuación describimos brevemente en qué consiste Windows Forms y explicaremos por qué nos hemos decidido finalmente a utilizar Windows Presentation Foundation (WPF).

4.3.1 Windows Forms

Windows Forms es un API de programación incluida en .NET Framework que le permite al programador crear interfaces gráficas. Windows Forms abstrae al programador de la manera en que Windows crea ventanas y componentes, de una manera similar a como lo hace Swing de Java. En la figura 4.1 vemos un ejemplo de interfaz gráfica creada usando Windows Forms.

4.3.2 ¿Por qué Windows Presentation Foundation?

El hecho de que exista una API de programación gráfica para .NET, sencilla de utilizar para cualquier programador que haya usado Swing de Java alguna vez, hace pensar por qué hemos usado una tecnología completamente nueva, incipiente, y algo más compleja de utilizar.

La respuesta es que Windows Presentation Foundation (WPF) no supone un reemplazo de lo que se está usando ahora mismo de manera predominante, está indicado para construir interfaces gráficas para cierto tipo de proyectos con unos requisitos muy específicos. Estos proyectos necesitan combinar diferentes tipos de elementos multimedia (vídeos, imágenes, etc.), o bien proporcionar una interfaz similar a la navegación por una web. Creemos que nuestro proyecto puede aprovecharse de

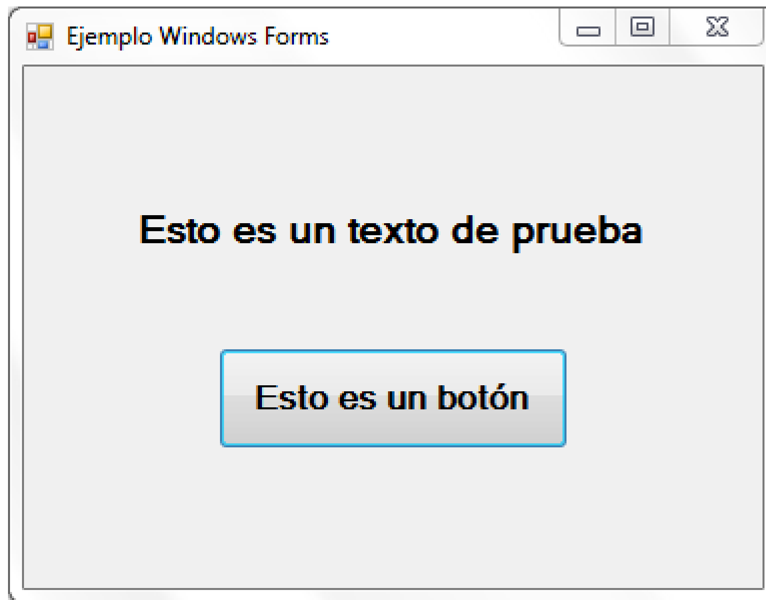

Figura 4.1: Ejemplo de aplicación Windows Forms

estas ventajas que ofrece WPF. Adicionalmente, WPF proporciona formas para crear animaciones y efectos visuales que pensamos son atractivos de cara a presentar la aplicación a un usuario final.

4.3.3 Construir interfaces gráficas en WPF mediante Expression Blend

Si bien el programa Visual Studio que hemos usado para diseñar y programar la lógica de nuestro proyecto permite realizar interfaces gráficas en WPF, se trata de una herramienta más bien orientada a desarrolladores y no a diseñadores, por lo que en su lugar hemos usado la herramienta que proporciona Microsoft a los diseñadores que hagan uso de la tecnología WPF, Expression Blend. Este programa permite realizar interfaces gráficas en WPF de manera más o menos sencilla o intuitiva, arrastrando componentes gráficos a un lienzo y creando animaciones de manera sencilla modificando en el tiempo ciertos atributos de esos componentes. Se puede obtener más información sobre Expression Web en esta página: http://www.microsoft.com/expression/products/blend_overview.aspx

Capítulo 5

Manual de usuario

La interfaz de usuario ha sido diseñada de la manera más intuitiva posible para los usuarios de la aplicación (figura 5.1).

Está constituida por una serie de botones que nos permiten desplazarnos por las distintas posibilidades de la aplicación. Además, en la parte superior izquierda de la interfaz tenemos un par de botones de retroceso y avance para poder navegar de forma más cómoda, y que en caso de cometer un error, podamos retroceder fácilmente para cambiar lo que necesitemos sin necesidad de empezar de nuevo desde el principio.

En la ventana principal podemos contemplar las distintas posibilidades del algoritmo.

La primera opción aplica, sobre un par de imágenes elegidas por el usuario, el algoritmo estéreo estándar, es decir, el basado en correlación, sección 2.2.

El segundo botón realiza el anaglifo de las imágenes dadas. Los anaglifos son imágenes de dos dimensiones capaces de provocar un efecto tridimensional, cuando se visualizan con lentes especiales (lentes de color diferente para cada ojo).

Se basan en el fenómeno de síntesis de la visión binocular. Las imágenes de anaglifo se componen de dos capas de color, superpuestas pero desplazadas ligeramente una respecto a la otra para producir el efecto de profundidad que proporciona precisamente la disparidad existente.

Los pasos a seguir para construir el anaglifo son:

- Para la imagen izquierda, se eliminan los colores azul y verde.
- Para la imagen izquierda, se elimina el color rojo.
- Finalmente, la imagen izquierda se arrastra hacia la derecha de forma que ambas queden superpuestas.

De esta manera, la imagen contiene dos imágenes filtradas por color, una para cada ojo. Cuando se ve a través de las gafas especiales, se revelará una imagen tridimensional gracias a la percepción de la corteza visual de nuestro cerebro.

Finalmente, el tercer botón sirve para ejecutar el algoritmo estéreo de Lankton sobre las imágenes que el usuario elija. El algoritmo de Lankton ya fue explicado en el capítulo de los conceptos teóricos (2) y en el de diseño (3).

Supongamos que elegimos la opción *Algoritmo estéreo estándar*. A continuación, llegamos a la pantalla de *Parámetros del algoritmo*. Aquí debemos elegir los parámetros más adecuados según el tamaño de las dos imágenes y las propiedades que posean, como puede ser el hecho de que estén o no equilibradas, o si tienen distintas intensidades.

En la parte izquierda de la figura 5.2 podemos observar cuatro parámetros configurables:

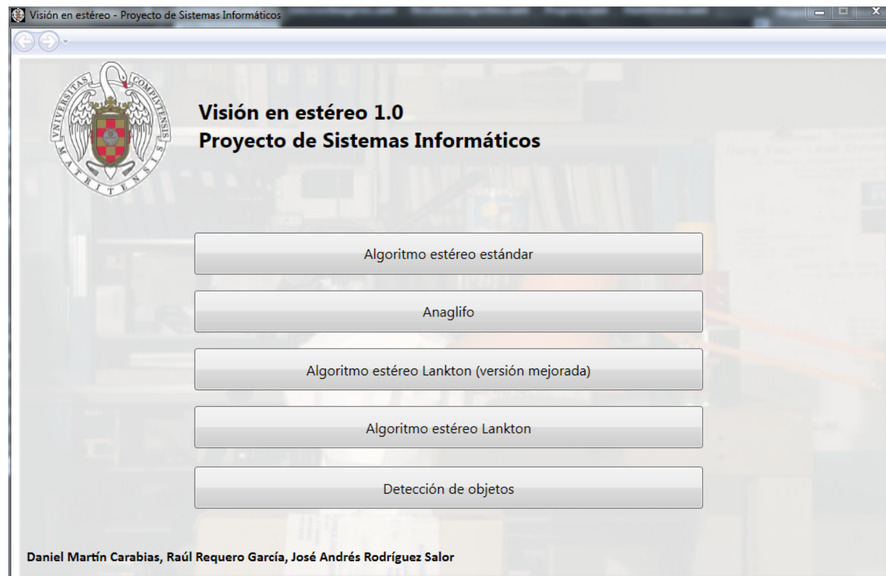

Figura 5.1: Pantalla principal de la aplicación

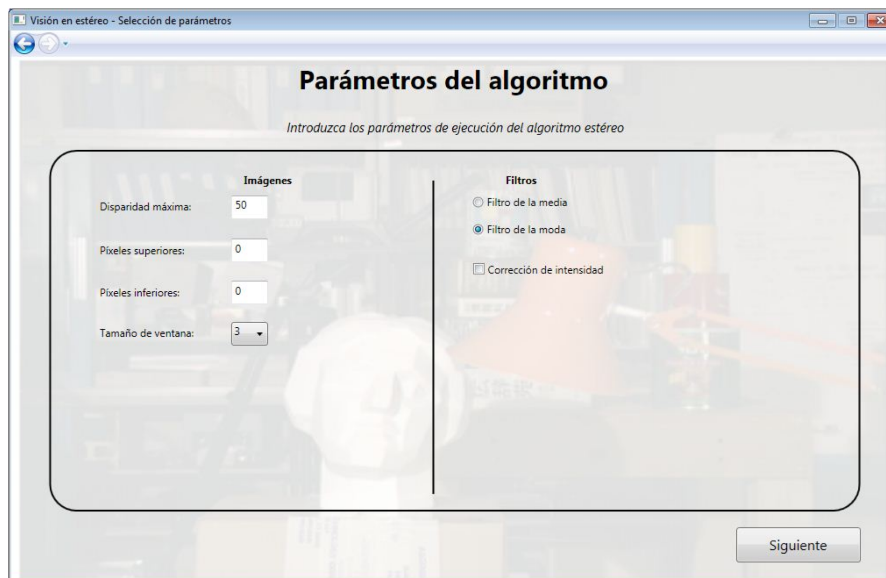

Figura 5.2: Parámetros del algoritmo estándar fijados

1. *Disparidad máxima*: es la distancia en número de píxeles entre un mismo punto en la imagen izquierda y la derecha. Realmente, lo que queremos calcular es la disparidad en cada punto, por lo que, en función del tamaño de la imagen y de las cámaras, tenemos que acotar la búsqueda que realizará el algoritmo indicando el valor máximo de disparidad que se detectará para ahorrar tiempo de ejecución.

2. *Píxeles superiores*: se puede dar el caso de que no tengamos dos cámaras perfectamente calibra-

das y las imágenes, por tanto, no estén en la misma horizontal. Para compensar este desajuste, utilizamos este parámetro de forma que, si la imagen izquierda está desplazada verticalmente hacia abajo con respecto a la imagen derecha, se ejecute el algoritmo desplazando hacia arriba la imagen izquierda como máximo tantos píxeles como indique este parámetro.

3. *Píxeles inferiores*: es análogo al parámetro anterior, pero en el caso de que la imagen izquierda estuviera en una vertical superior a la de la imagen derecha. Se debe tener especial cuidado al fijar estos parámetros, ya que al incrementar su valor originará que se dispare el tiempo de ejecución del algoritmo. Por lo tanto, aunque incluimos esta posibilidad en nuestra aplicación, conviene tomar imágenes calibradas previamente.

4. *Tamaño de la ventana*: a la hora de aplicar los distintos filtros debemos fijar el tamaño de la ventana necesaria para realizar los cálculos previstos, esto es el número de píxeles que se toman alrededor del píxel examinado en un determinado momento. A mayor tamaño de ventana se suele tener mayor precisión (salvo que la imagen sea demasiado compleja) pero también mayor tiempo de ejecución. Este valor indica el número de filas y columnas que tendrá el cuadrado en el que el centro del mismo será el píxel sobre el que estemos operando. Si por ejemplo elegimos un tamaño de ventana de 3, tendremos un cuadrado de 9 casillas en el que el píxel central será el de interés.

En la parte derecha de la figura 5.2 aparecen dos parámetros más:

1. *Filtros*: se permite al usuario elegir entre dos posibles filtros. Ambos filtros sirven para eliminar la excesiva variedad de píxeles que se producen al calcular las disparidades ya que esto proporciona un excesivo número de píxeles con disparidades muy cambiantes, haciendo además que su visualización sea muy difusa.

   Estos filtros se aplican sobre la ventana del tamaño elegido. En el caso del filtro de la media, se trata de tomar todos los valores de la ventana, calcular su media (se suman todos sus valores y se divide entre el número de casillas) y colocar en el píxel central del cuadrado (nuestro píxel a tratar) el valor obtenido.

   En el caso del filtro de la moda calcula, como su propio nombre indica, la moda de los valores de la ventana, es decir, el valor de intensidad que más se repite en la misma. Al píxel central de la ventana se le asigna el valor de la moda, así obtenido.

   Por lo general, el filtro de la moda produce mejores resultados que el de la media.

2. *Correccion de intensidad*: Esta opción sirve para aplicar el filtro de normalización de intensidad. Es muy útil para imágenes en las que la luminosidad de ambas es distinta debida a reflejos de luz producidos por el Sol o por luces artificiales.

Después de elegir los parámetros según las necesidades del usuario, presionamos el botón siguiente. Recordamos que siempre que se desee se puede retroceder con los botones de navegación situados en la parte superior izquierda de la interfaz.

A continuación podemos observar la pantalla de *Selección de imágenes*. Aquí se nos permite elegir las imágenes que deseemos para posteriormente ejecutar el algoritmo. Es fundamental situar en la parte izquierda la imagen izquierda y en la parte derecha la imagen derecha, ya que de lo contrario el algoritmo no obtendrá los resultados previstos al esperar esta configuración. Las imágenes se podrán cargar usando los botones de *Cargar imagen...*, que permiten buscar en nuestras carpetas las imágenes deseadas.

En la figura 5.3 se muestra la selección de dos imágenes estereoscópicas, izquierda y derecha respectivamente.

Figura 5.3: Pantalla de selección de imágenes con dos imágenes ya cargadas

Una vez elegidas las imagenes que deseemos, podemos proceder a ejecutar el algoritmo con las mismas y los parámetros fijados. Pulsamos el botón *Ejecutar*, y tras una espera de tiempo variable obtendremos los resultados esperados.

En la figura 5.4 podemos observar el resultado de la ejecución del algoritmo de correlación. Si nos situamos en distintos puntos de la imagen vemos las coordenadas de la misma, así como la disparidad en esa posición.

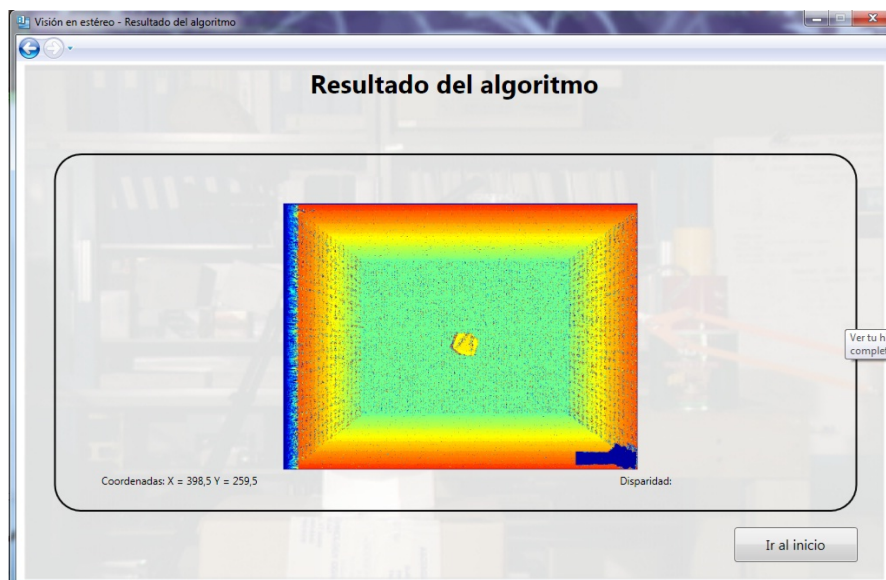

Figura 5.4: Resultados tras la ejecución del algoritmo estándar

Si volvemos a la pantalla principal del algoritmo pulsando el botón *Ir al inicio* y elegimos la opción *Anaglifo*, se nos mostrará de nuevo la pantalla de carga de imágenes.

Figura 5.5: Imágenes seleccionadas para realizar el anaglifo

Esta vez hemos elegido otras dos imágenes distintas (figura 5.5), y el resultado obtenido es el anaglifo cuya generación se ha explicado al principio de este capítulo, figura 5.6.

Figura 5.6: Resultados al realizar el anaglifo

Por último, regresamos a la pantalla principal pulsando de nuevo el botón *Ir al inicio* y elegimos la siguiente de las opciones que nos queda por probar: *Algoritmo estéreo Lankton (versión mejorada)*. Tras pulsar dicho botón, nos aparecerá una nueva ventana de parámetros ligeramente distintos a los del algoritmo estándar, basado en correlación. Entre los parámetros nuevos que encontramos en la figura 5.7, los desplazamientos verticales de las imágenes funcionan de la siguiente manera: cuanto se introduce un valor positivo, la imagen que hayamos desplazado se moverá hacia abajo, y los píxeles sobrantes en la parte superior se rellenarán de ceros (equivalente a poner píxeles negros).

Por otro lado el parámetro *Verticalidad* se utiliza para imágenes que no están exactamente calibradas, para que de esta forma, se examinen píxeles superiores e inferiores en ambas imágenes de cara a encontrar la mejor correspondencia. Normalmente valdrá 0 (imágenes calibradas) ya que incrementar este parámetro tendrá como consecuencia un aumento considerable del tiempo de ejecución.

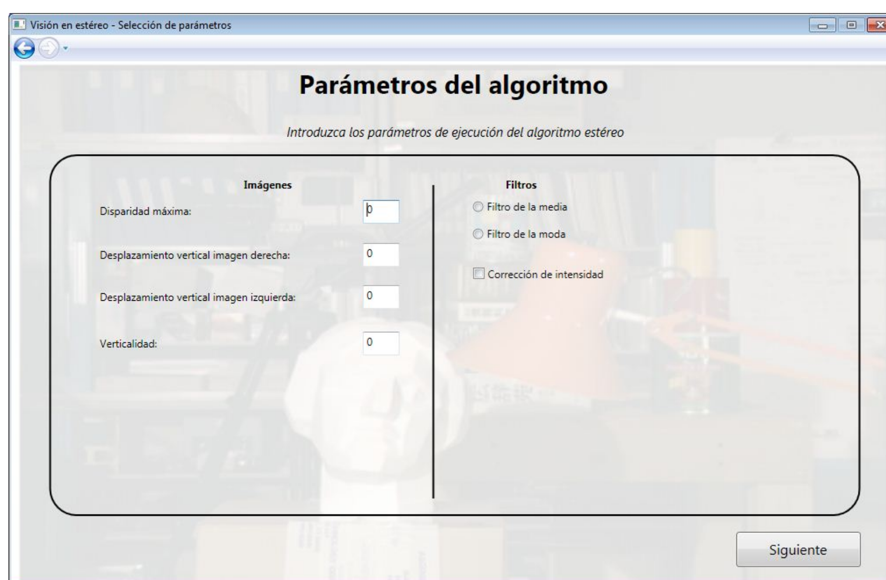

Figura 5.7: Pantalla de parámetros del algoritmo Lankton (versión mejorada)

Tras elegir los parámetros deseados y pulsar el botón *Siguiente*, llegaremos a la pantalla de carga de imágenes. De nuevo elegiremos las dos imágenes a procesar y procederemos a ejecutar el algoritmo.

El resultado se presentará de forma similar a como hemos visto en el algoritmo estándar, es decir, con una imagen que muestra las disparidades coloreadas en función de la distancia de los distintos elementos que componen la imagen. En la figura 5.8 podemos ver el resultado del cálculo de disparidades de dos imágenes tomadas a partir de un balón de reglamento.

Esta versión mejorada del algoritmo de Lankton incluye el buen funcionamiento de dicho algoritmo para imágenes tomadas de la realidad. La diferencia con el siguiente botón que vamos a probar (Algoritmo estéreo Lankton) es que este último funciona particularmente bien para imágenes simuladas (más exáctamente para las del estudio de Middlebury), mientras que para las reales es mejor utilizar la versión mejorada.

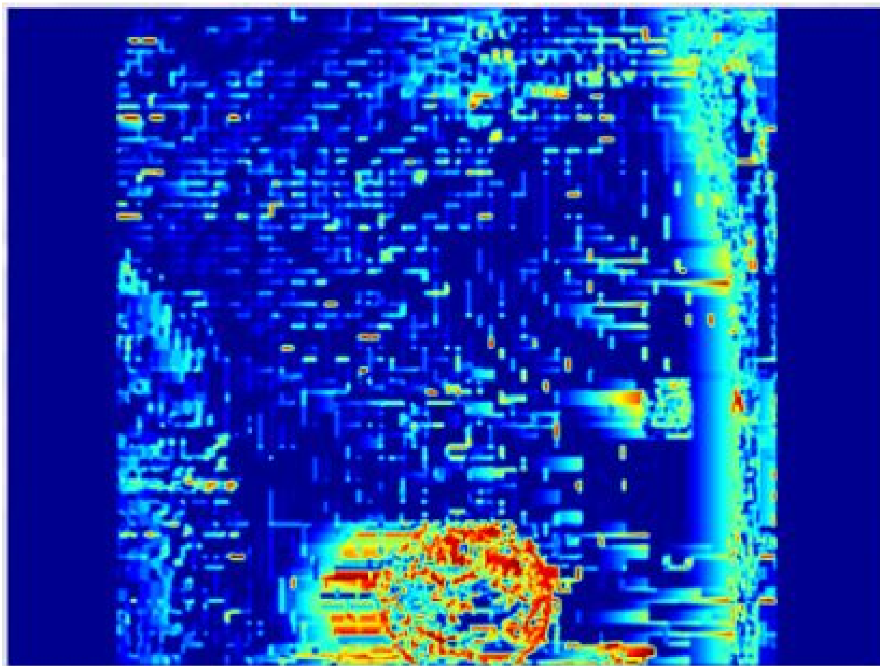

Figura 5.8: Pantalla de resultados del algoritmo Lankton (versión mejorada).

En el capítulo de resultados y conclusiones (7) se pueden ver los distintos resultados obtenidos tras la aplicación de dichos algoritmos en distintas imágenes y parámetros.

Finalmente, el último botón se refiere a la *Detección de objetos*. Al presionar dicho botón, y como en casos anteriores, se muestra una ventana de parámetros (como se puede observar en la figura 5.9):

Los parámetros que indican la disparidad máxima, los desplazamientos verticales de la imagen derecha e izquierda y la verticalidad ya están explicados anteriormente.

Respecto a los nuevos parámetros, el *umbral de color* es el margen a partir del cual se detectan objetos de un color determinado con respecto al píxel que se toma en cada momento.

El tipo de píxel es el píxel de ejemplo a partir del cuál se va a realizar la detección de objetos. En nuestro caso tenemos predefinido un píxel de tipo *Bote*, referido al color de los píxeles de un bote amarillo. También se puede definir el píxel a mano con la opción *Personalizar* y eligiendo los valores RGB deseados por el usuario.

Al presionar el botón *Siguiente*, podremos cargar de nuevo dos imágenes, y al ejecutar el algoritmo obtendremos como resultado la matriz de disparidades y la imagen original con su centroide bien determinado con un pequeño cuadrado de color verde.

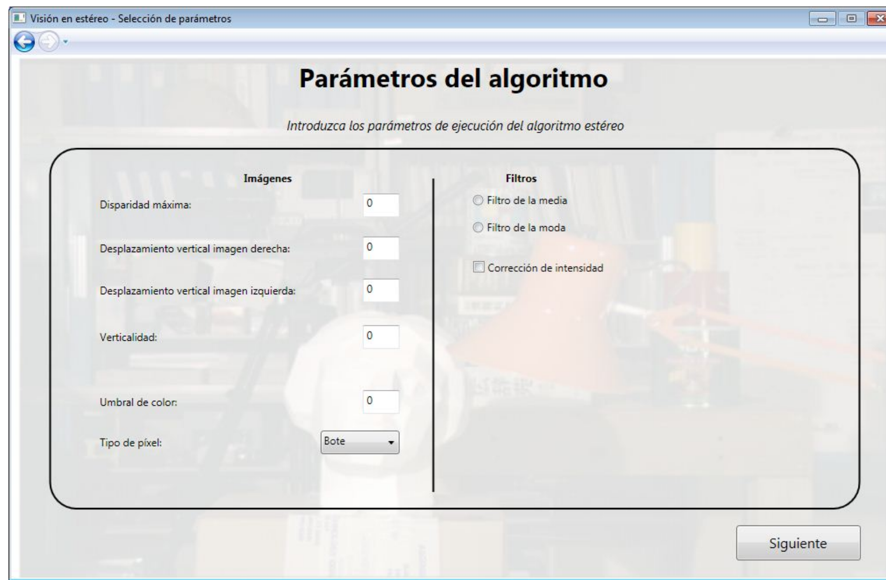

Figura 5.9: Pantalla de parámetros de la opción de detección de objetos

Capítulo 6

Proceso de desarrollo

Todo desarrollo de proyecto de tamaño medio o grande requiere una cierta planificación para que se finalice en un plazo determinado y con unos requisitos prefijados con anterioridad. Podríamos hablar, por lo tanto, de una secuencia de etapas necesarias, que son las que se han seguido en este diseño con el fin de acabar con un proyecto que cumpla con los objetivos planteados.

Cada una de estas etapas no está exenta de fallos y regresiones que obligan a repasar una etapa anterior, por este motivo es recomendable tener siempre una visión global del proyecto y no quedarse únicamente en la etapa que se esté desarrollando en ese momento. Por otra parte, los fallos, regresiones o simplemente descubrimientos que se hagan en una etapa pueden influir potencialmente en las etapas posteriores. Por ejemplo, si un determinado algoritmo no ofrece resultados idóneos, se puede plantear otra etapa intermedia que incluya modificaciones sobre el mismo o el análisis de otros existentes que pudieran proporcionar mejores resultados en ese escenario en particular.

A continuación describimos las etapas que hemos seguido para desarrollar nuestro proyecto:

6.1 Análisis previo en MATLAB

Como paso previo al desarrollo del algoritmo, utilizamos MATLAB con el fin de visualizar unos resultados iniciales y poder basarnos en ellos cuando realizásemos las primeras pruebas a nuestro código.

En un principio probamos con un algoritmo estándar de visión estereoscópica, cuyos resultados se comentan en el capitulo 7. Las imágenes que utilizamos eran sencillas, en el sentido de que las distancias o disparidades a los objetos que aparecen en ellas están claramente diferenciadas. En el momento en que los resultados con estas imágenes sencillas fueron satisfactorios, nos pusimos a implementar el algoritmo estándar de visión estéreo en C#.

6.2 Desarrollo en C# del algoritmo estándar

El desarrollo del algoritmo estándar de visión estéreo en C# no nos resultó demasiado complicado teniendo el código de MATLAB como apoyo. Las pruebas con las imágenes iniciales ofrecieron unos resultados satisfactorios y exactamente iguales a los de MATLAB. Tras esto planteamos como etapa posterior el análisis de imágenes del proyecto Middelbury, algo más complicadas que las imágenes inicialmente utilizadas.

Una primera ejecución de nuestro algoritmo con esas imágenes nos ofreció resultados que no fueron satisfactorios. Para cerciorarnos de que no se trataba de un fallo de nuestra implementación,

probamos con las mismas imágenes en MATLAB y el resultado fue igualmente bastante mejorable.

6.3 Desarrollo de un nuevo algoritmo: Lankton

Tras una búsqueda en Internet de posibles algoritmos de visión estéreo que pudieran ofrecer mejores resultados con imágenes de Middlebury, dimos con la web de Shawn Lankton, autor de diversos artículos sobre visión artificial. En su web encontramos una implementación en MATLAB del algoritmo de visión descrito en [KSK06]. Así pues, nos dispusimos a analizar los resultados de este nuevo algoritmo con el nuevo conjunto de imágenes. Las pruebas iniciales en MATLAB con el nuevo conjunto de imágenes ofrecieron unos resultados mejores que los que se obtuvieron con el algoritmo estándar, por lo que este nuevo algoritmo se convirtió en el candidato para nuestro proyecto. La implementación del nuevo algoritmo no nos resultó demasiado complicada.

Después de la implementación nos dispusimos a realizar pruebas con las imágenes de Middlebury y éstas fueron igual de satisfactorias que en MATLAB. Cuando comprobamos que funcionaba bien con dichas imágenes, el siguiente paso fue probar con imágenes tomadas de un simulador basado en TrueVision. Los resultados con ellas fueron satisfactorios, y por tanto, como comprobación, las probamos de nuevo en el algoritmo implementado en MATLAB. El resultado fue el mismo que con el algoritmo implementado en C#. A la vista de estos resultados nos pusimos a buscar otros algoritmos que funcionasen para las imágenes tomadas con el simulador mencionado.

6.4 Búsqueda y desarrollo de una nueva versión del algoritmo de Lankton

Buscando por la web de Shawn Lankton, encontramos una versión previa al algoritmo implementado [KSK06]. Tras las pertinentes pruebas con él, los resultados que ofrecía para las imágenes tomadas con el simulador eran muy buenos. Tras detectar las diferencias entre los dos algoritmos implementamos esta versión en C#. Como cabía esperar, los resultados de las pruebas realizadas con las imágenes tomadas con el simulador en la implementación de C# fueron satisfactorias. Con los resultados de los dos algoritmos nos decantamos por esta segunda versión para pasar a las pruebas con imágenes reales.

En este punto se nos ofrecía otro reto: disminuir el tiempo de ejecución del algoritmo en la implementación de C#, ya que el tiempo que requería la ejecución era demasiado elevado. A partir de aquí teníamos dos caminos que seguir:

1. Probar cómo se comportaba el algoritmo con imágenes reales
2. Mejorar el tiempo de ejecución del algoritmo

A continuación detallamos los pormenores de las dos opciones.

6.5 Problemas de rendimiento en la implementación en C#

Una vez implementado este nuevo algoritmo en C# observamos que los resultados eran similares a los que se obtenían en MATLAB, pero el rendimiento era mucho peor. De hecho, si el algoritmo implementado en MATLAB tardaba 1 minuto, la implementación en C# requería siete u ocho veces más tiempo. Nuestro siguiente paso fue analizar por qué ocurría esta diferencia tan considerable de rendimiento y tratar de reducirla al mínimo posible.

Uno de los principales cuellos de botella que existía en nuestra aplicación es la forma que teníamos de acceder a los píxeles de la imagen. Para representar las imágenes en C# utilizamos la clase Bitmap que nos proporciona .NET Framework. Esta clase permite representar en memoria una imagen de mapa de bits, pero el acceso a ésta es demasiado lento debido a que para un píxel dado de la imagen realiza un recorrido por todos los píxeles hasta que se llega al elegido. Para mejorar el tiempo de acceso a cada uno de los píxeles de la imagen, implementamos una clase que utiliza punteros y accede directamente al píxel elegido. A esta clase la hemos llamado *BitmapFast*.

Cuando solucionamos el problema de acceso a los píxeles de la imagen, el tiempo descendió considerablemente, pero no todo lo deseable.

Para seguir mejorando el rendimiento nos dedicamos a estudiar cada parte del código. Después de realizar un análisis detallado del rendimiento del código, nos dimos cuenta de que había una parte del mismo que se podía ejecutar en paralelo, centrándonos a continuación en esta posibilidad.

A partir de este punto, nuestra aplicación manejaba hilos y esto hizo que el tiempo se redujera casi a la mitad con respecto a la versión anterior.

Con estas dos modificaciones en el código y algún que otro cambio en los bucles, el rendimiento mejoró considerablemente respecto de la versión de MATLAB.

Los pasos del algoritmo de Lankton en su última versión son los que aparecen en la figura 6.1.

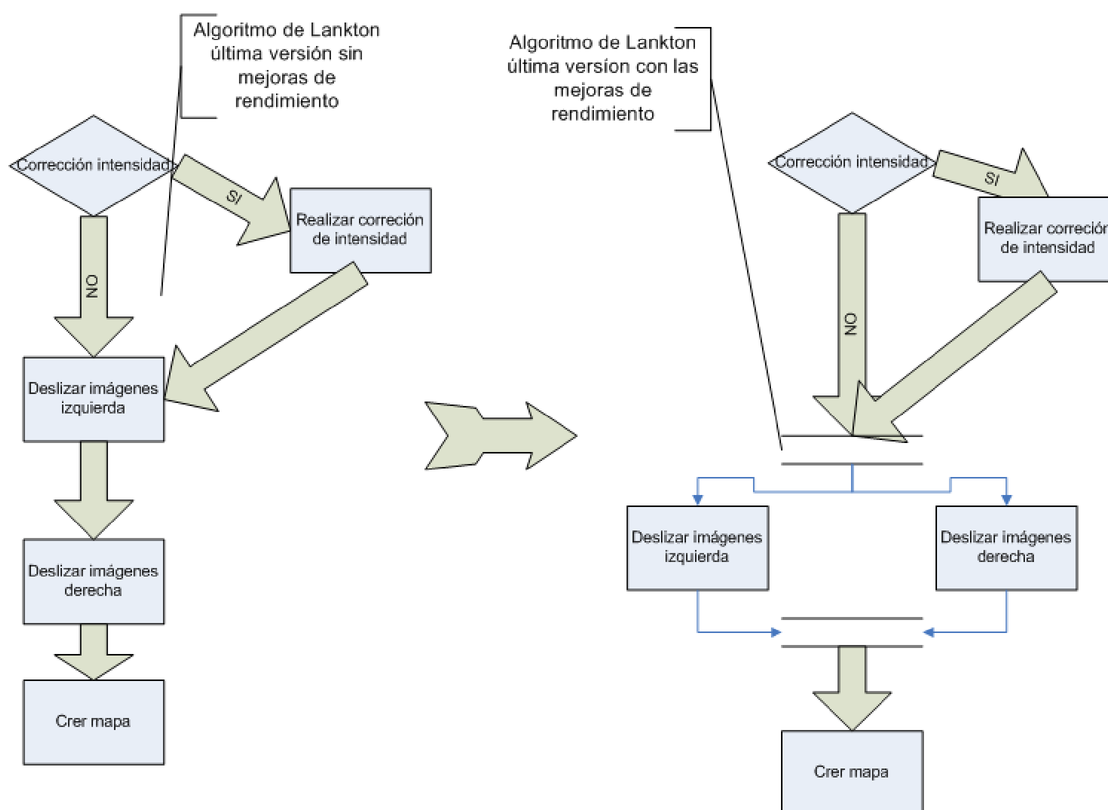

Figura 6.1: Diagrama que representa los pasos del algoritmo de lankton en su última versión con y sin mejoras de rendimiento

6.6 Pruebas con imágenes reales

Realizamos pruebas con imágenes tomadas por el robot Surveyor. El resultado no fue todo lo satisfactorio que hubiese sido deseable ni en MATLAB ni en el algoritmo de C#. El resultado de la ejecución nos proporcionaba un mapa de disparidad donde muchos de los píxeles se les asignaba disparidad cero.

Nos pusimos a estudiar el código más en profundidad para detectar por qué ocurría esto. Después de estudiarlo a fondo, nos dimos cuenta de que una parte del código donde el algoritmo unía el resultado del desplazamiento de las imágenes y elegía uno u otro (*winner-take-all*) era demasiado restrictivo y dejando muchos píxeles con valor de disparidad cero. Encontrado el problema nos dedicamos a solucionarlo. En este punto, surgió un nuevo método *winner-take-all* que no era tan restrictivo como el anterior proporcionando mejores resultados. Otro de los problemas que encontramos fue que las dos imágenes tomadas por las cámaras del robot no tenían la misma intensidad. Para solucionarlo implementamos un filtro de corrección de intensidad basado en la obtención de las medias de intensidad de cada imagen a partir de la imagen transformada previamente al modelo de color HSL. La imagen con media menor se le aumentaba la luminiscencia a cada píxel en un valor dado a partir de la formula 3.2. Este algoritmo se ha explicado en la sección 3.

6.7 Detección de objetos en la imagen

Debido a las necesidades del grupo con el que teníamos que cooperar, necesitábamos algún método que nos permitiera detectar obstáculos dentro de las imágenes para poder suministrar esta información al algoritmo de navegación desarrollado por el mencionado grupo. En una primera investigación encontramos un método basado en el filtrado por disparidad. Esto consiste en filtrar la imagen a partir de una disparidad predeterminada de forma que con los píxeles que nos quedaran obtener una posición (x, y) hallada con la media, representando el punto medio del obstáculo. Este método terminamos descartándolo por su falta de operatividad respecto de las necesidades de navegación.

Nos dispusimos a buscar otros métodos en la web, libros, y otros materiales de consulta. Encontramos la librería OpenCV [1] para C# denominada Emgu CV [2] que permite detectar objetos previamente fijados a partir de un entrenamiento previo. OpenCV es una librería desarrollada en C++ que es bastante conocida en el ámbito de la visión por computador. Concretamente, esta librería incluye un clasificador que por defecto está entrenado para detectar caras, pero que puede utilizarse para entrenar cualquier otro tipo de objetos, tal y como se describe en [VJ04].

El clasificador que incorpora OpenCV es en realidad un conjunto de clasificadores. La decisión final de clasificación se calcula ponderando cada una de las decisiones particulares de los clasificadores que integran este conjunto. Durante el entrenamiento aprende cada clasificador del grupo. Cada uno de los clasificadores de cada grupo se considera un clasificador "débil", en el sentido de que están compuestos por árboles de decisión de una única variable. Durante el entrenamiento cada árbol de decisión aprende sus decisiones de clasificación a partir de los datos disponibles, a la vez que aprende un peso para cada decisión, según su precisión sobre los datos. Este proceso continua hasta que la tasa total de error sobre el conjunto de datos (la cual proviene de los votos ponderados de los clasificadores) cae por debajo de un determinado umbral. Como se comenta en [FS95], este algoritmo da buenos resultados, siempre y cuando el conjunto de datos del entrenamiento sea considerablemente grande (del orden de miles de imágenes).

Para poder hacer funcionar la librería OpenCV en C#, es necesario crear un envoltorio (*wrapper*) en .NET, o utilizar alguno ya existente, como la librería Emgu CV. Uno de los motivos por los que

[1] http://sourceforge.net/projects/opencvlibrary/
[2] http://www.emgu.com

usamos Emgu CV y no alguna de las otras alternativas existentes es que Emgu CV está escrita completamente en C#, por lo que puede ser compilado sin problemas en la plataforma Mono, lo que quiere decir que el código generado es multiplataforma, puede ser usado tanto en Windows, como en Linux o en Mac OS X.

Paralelamente consideramos otro método de detección de objetos basado en color. Este método se fundamenta en obtener los píxeles de la imagen más similares a un color dado. Dicho método requería que los objetos también estuvieran prefijados. La forma elemental de proceder es la siguiente. Dado un color en modelo RGB, se buscan los píxeles que más se asemejan según el criterio de mínima distancia dado por la siguiente expresión:

$$d = \sqrt{(R1 - R2)^2 + (G1 - G2)^2 + (B1 - B2)^2} < offset \quad (6.1)$$

donde R1, G1, B1 y R2, G2, B2 son las componentes de los respectivos píxeles en el modelo RGB. El valor del offset se fija en cada ejecución del algoritmo y permite filtrar los píxeles. Debido a los problemas y el escaso tiempo disponible para probar a fondo la librería de OpenCV Emgu para la detección de objetos, nos decantamos por el método de detección por color para llevar a cabo la detección de objetos, en la escena 3D. El estudio a fondo de la funcionalidad de OpenCV puede plantearse como una opción de estudio para futuros proyectos de esta naturaleza.

6.8 Integración con el simulador de robot

Como hemos comentado en la introducción, uno de los objetivos de nuestro proyecto es la integración con un grupo dedicado a la conducción de un robot. Con este motivo hemos incorporado un método público que aplica el algoritmo de detección de objetos a un par de imágenes pasadas como parámetro y devuelve como resultado un documento XML.

Este documento XML proporciona la siguiente información sobre el objeto detectado: Coordenadas X e Y y la disparidad máxima de dicho objeto.

El proceso de integración con el grupo dedicado a la conducción del robot no estuvo exento de problemas. Como el grupo objeto de la coordinación estaba utilizando su propia versión de la librería AForge.NET, tuvo problemas a la hora de usar nuestra librería, pues las versiones de AForge.NET diferían.

Una vez ejecutaron nuestra librería, nos comentaron que les vendría bien que nuestro algoritmo además de devolver un fichero XML con toda la información sobre el objeto podría devolver un formulario gráfico con los resultados obtenidos. Lo que hicimos fue crear un formulario gráfico que mostrara la imagen izquierda y sobre ella se marcara un punto de color verde en la posición central del objeto detectado. Un parámetro booleano permite decidir si se muestra esta ventana o no.

Capítulo 7

Resultados y conclusiones

En este capitulo presentamos los resultados de nuestros algoritmos para los cuatro tipos de imágenes que hemos tratado a lo largo del desarrollo del proyecto, a saber:

- Imágenes de Middlebury
- Imágenes de simulador
- Imágenes reales, tomadas con el robot.
- Imágenes reales, para detección de objetos.

7.1 Imágenes utilizadas

Como se ha mencionado previamente se distinguen tres tipos de imágenes, que se describen a continuación.

7.1.1 Imágenes de Middlebury

Las imágenes estereoscópicas típicas de Middlebury son las que aparecen en las figuras 7.1.

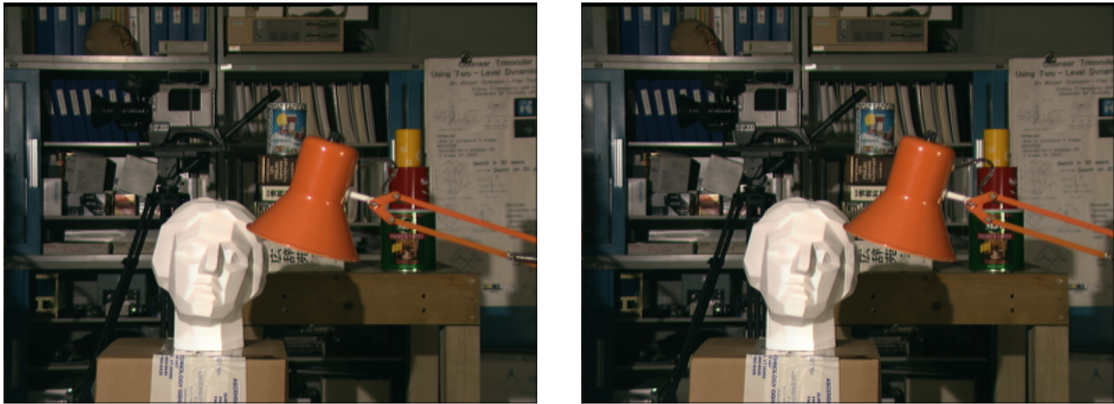

Figura 7.1: Imagen derecha e izquierda de Middlebury

7.1.2 Imágenes de simulador

Todos los resultados que se han obtenido con las imágenes de simulador se han basado en las imágenes mostradas en la figura 7.2. Se han elegido estas imágenes para hacer pruebas porque tienen un fondo bien diferenciado del resto de la imagen. Sólo existe un objeto en la escena (el cubo) que está aislado y es fácilmente reconocible. Por tanto esta imagen es una buena candidata para comprobar la efectividad de los algoritmos ya que no existen objetos que puedan generar errores en los resultados.

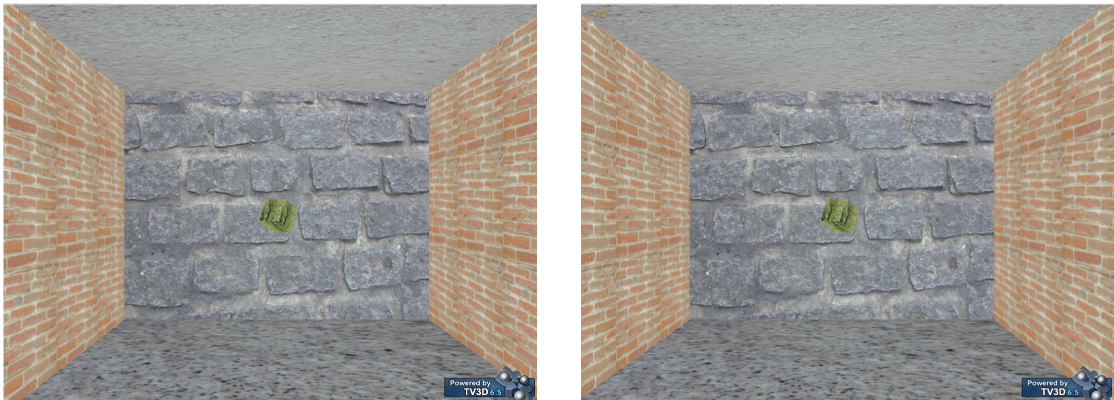

Figura 7.2: Imagen derecha e izquierda de simulador

7.1.3 Imágenes reales

Las imágenes reales mostradas en la figura 7.3 se han intentado obtener manteniendo las propiedades de las imágenes del simulador 7.2. Esto es, tratando de mantener un fondo homogéneo y un único objeto en la escena. Si bien, esto no ha sido del todo posible. La dificultad que presentan estas imágenes es que no presentan los mismos niveles de intensidad las dos imágenes y además el objeto (el balón) tiene brillos, cosa que dificulta el obtener buenos resultados en los algoritmos.

Figura 7.3: Imagen derecha e izquierda reales

7.1.4 Imágenes reales, para detección de objetos

Las imágenes 7.4 y 7.5 han sido las utilizadas para probar el algoritmo de detección de objetos en una imagen 3D. En dichas imágenes se puede observar un bote de color amarillo que se diferencia bien dentro de las escena, de este modo no resulta demasiado complicada su detección. La problemática de estas imágenes radica en los reflejos del bote que se producen en el suelo que van a producir que el algoritmo no obtenga el centro del bote exactamente.

Figura 7.4: Imagen derecha e izquierda para detección de objetos, pertenecientes al primer par

Figura 7.5: Imagen derecha e izquierda para detección de objetos, pertenecientes al segundo par

7.2 Resultados

7.2.1 Resultados con las imágenes de Middlebury

A continuación se muestran los resultados obtenidos para las imágenes de Middlebury mostradas en la figura 7.1.

Resultados con la primera versión del algoritmo de Lankton

- Sin realizar un filtro de moda posterior: como se puede observar en la imagen de la figura 7.6 se ha obtenido un mapa de disparidad satisfactorio. Con colores cálidos que indican disparidades altas se puede ver la lámpara, eso nos indica que ésta está más cerca del objetivo de las cámaras mientras que el fondo con color azul que indica valores bajos de disparidad nos dice que está alejado del objetivo de la cámara. Este resultado no ha sido tratado con ningún filtro posterior.

- Realizando un filtro de moda posterior: en la imagen 7.7 podemos ver el mismo resultado que en la imagen 7.6 pero con un tratamiento posterior. Se ha realizado un filtrado por moda de los valores de disparidad de la imagen. Si analizamos la imagen, comprobamos que se han resaltado los bordes de los objetos apareciendo más zonas homogéneas. Por tanto, hemos mejorado la calidad del resultado ya que ahora la imagen tiende a homogeneizarse.

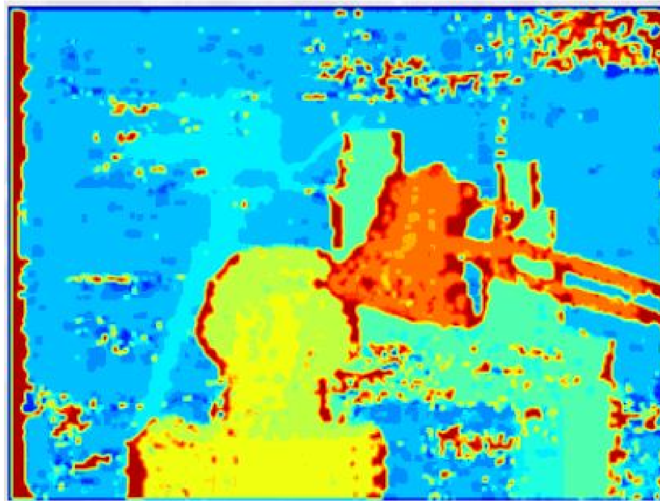

Figura 7.6: Mapa de disparidad con el algoritmo estéreo de Lankton (primera versión y sin filtro de moda)

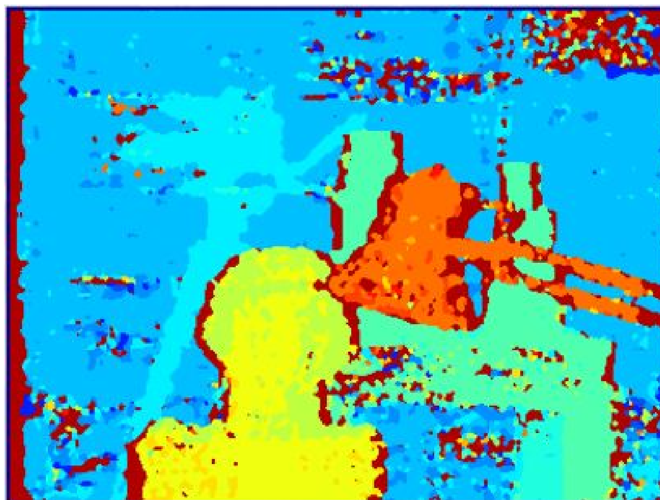

Figura 7.7: Mapa de disparidad con el algoritmo estéreo de Lankton (primera versión y con filtro de moda)

7.2.2 Resultados con las imágenes de simulador

A continuación se muestran los resultados obtenidos para las imágenes de simulador mostradas en la figura 7.2.

Resultados con el algoritmo estéreo estándar

El resultado de aplicar el algoritmo estándar a las imágenes de simulador es el que se muestra en la figura 7.8. El resultado obtenido resulta ser satisfactorio. Se puede observar el cubo en el centro de la imagen con un color amarillo que indica una mayor proximidad a las cámaras y por tanto con valores mayores de disparidad, mientras que con un color azul verdoso aparece la pared del fondo de la imagen (valores menores de disparidad). Otra de las cosas que se puede observar en la imagen es el degradado del rojo al azul verdoso que nos indica que cuanto más nos alejamos de las cámara menor es la disparidad. Este algoritmo nos ofrece buenos resultados para este tipo de imágenes, si bien el problema es que el tiempo de ejecución para obtener los resultados es grande.

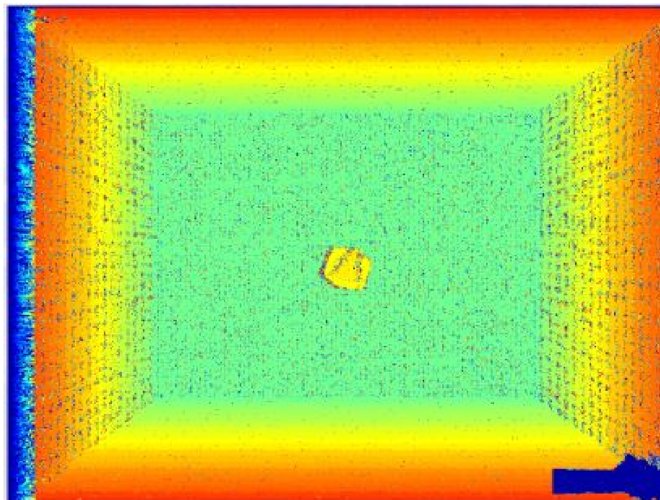

Figura 7.8: Mapa de disparidad obtenido con el algoritmo estéreo estándar para las imágenes del simulador

Resultados con la primera versión del algoritmo de Lankton

En la imagen de la figura 7.9 se puede ver la razón por la cual hemos implementado otra versión del algoritmo de Lankton, ya que el mapa de disparidad obtenido no resultas ser el esperado.

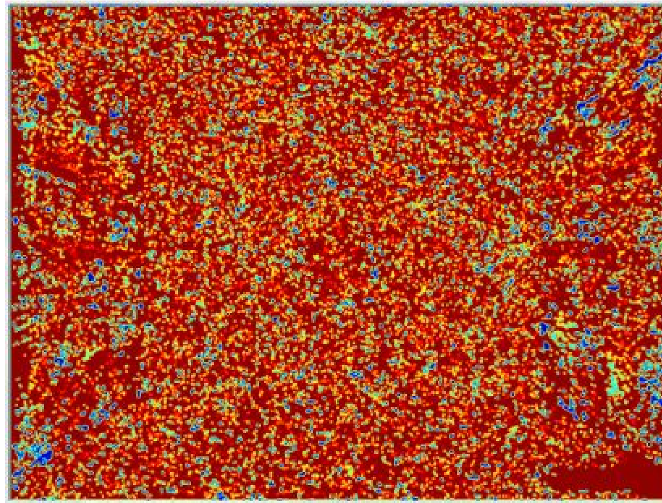

Figura 7.9: Mapa de disparidad obtenido con la primera versión del algoritmo de Lankton para imágenes del simulador

Resultados con la versión definitiva del algoritmo de Lankton

- Sin ningún tipo de tratamiento posterior del mapa de disparidad

  Como se puede observar en la imagen 7.10 el resultado obtenido es muy parecido al obtenido con el algoritmo estándar 7.8. La única diferencia existente entre los mapas de disparidad es que en el obtenido con el algoritmo estándar podemos ver que no es tan homogéneo como el mapa obtenido con la versión definitiva del algoritmo de Lankton. Esto se podría haber solucionado con un filtrado por moda posterior pero se habría incrementado el tiempo de computo, sin embargo con la versión definitiva del algoritmo de Lankton no nos hace falta realizar un tratamiento posterior y además el tiempo de cómputo es considerablemente más bajo.

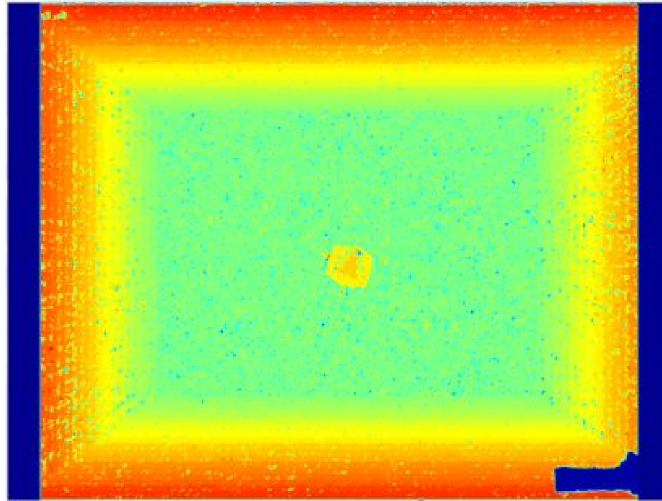

Figura 7.10: Mapa de disparidad obtenido con la versión definitiva del algoritmo de Lankton para imágenes del simulador sin ningún tratamiento posterior

- Realizando un filtrado por moda del mapa de disparidad

  El mapa de disparidad de la figura 7.11 prácticamente es el mismo que el de la imagen de la figura 7.10 si bien con un tratamiento posterior con el filtro de la moda. Con esto hemos conseguido homogeneizar más la imagen y eliminar valores erróneos.

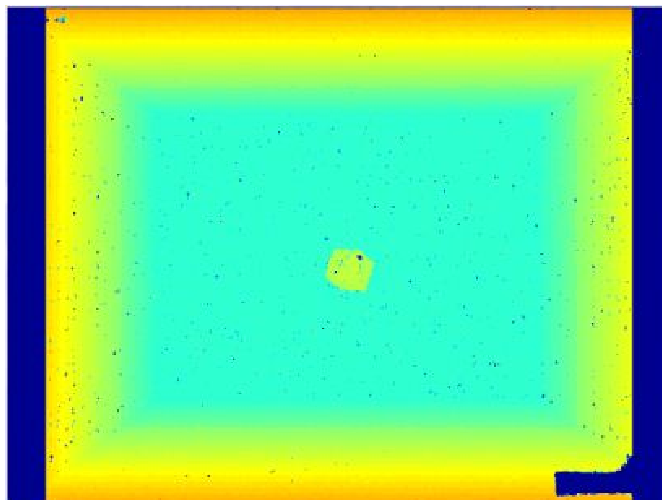

Figura 7.11: Mapa de disparidad obtenido con la versión definitiva del algoritmo de Lankton para imágenes del simulador y un filtrado por moda posterior

- Realizando un filtrado por media del mapa de disparidad

  Como ocurre en el caso anterior, de nuevo la imagen de la figura 7.12 es prácticamente idéntica a la imagen de la figura 7.10 con un tratamiento posterior con el filtro de la media. Con este filtro hemos conseguido un peor resultado que con el filtro de la moda figura 7.11.

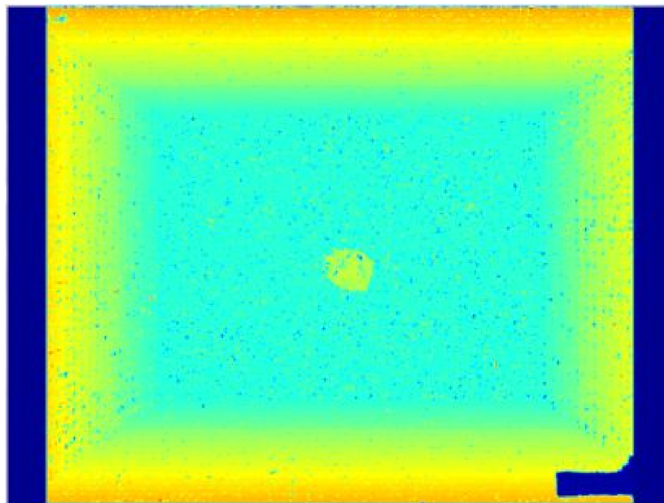

Figura 7.12: Mapa de disparidad obtenido con la versión definitiva del algoritmo de Lankton para imágenes del simulador y un filtrado por media posterior

7.2.3 Resultados con las imágenes reales

Los resultados obtenidos para las imágenes reales mostradas en la figura 7.3.

Resultados con la primera versión del algoritmo de Lankton

Como se puede observar en la imagen de la figura 7.13, el mapa de disparidad obtenido con la primera versión del algoritmo de Lankton no resulta ser satisfactorio.

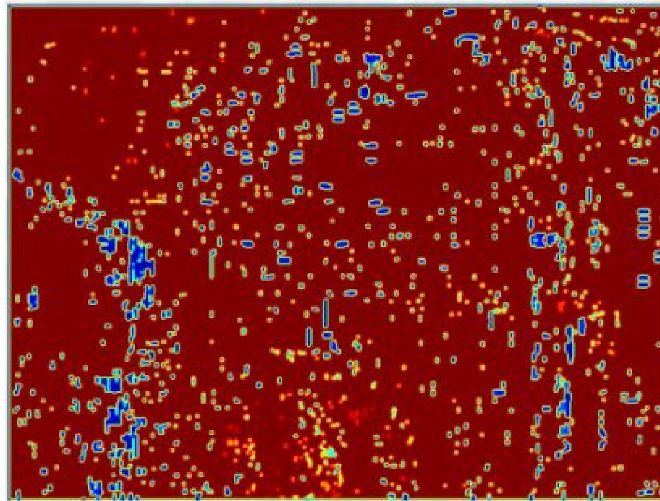

Figura 7.13: Mapa de disparidad obtenido con la primera versión del algoritmo de Lankton para imágenes reales

Resultados con la versión definitiva del algoritmo de Lankton

En el mapa de disparidad obtenido figura 7.14 se puede distinguir el balón en la escena ya que tiene valores de disparidad más altos que el resto (color rojo) aunque no posee valores de disparidad homogéneos debido a los brillos. Se puede ver también que el fondo no presenta un color homogéneo, esto es debido a que las dos imágenes de la figura 7.3 no tienen la misma intensidad. En el mapa de disparidad obtenido se puede observar que el balón tiene valores mayores de disparidad ya que en dicha imagen esta más próximo a las cámaras.

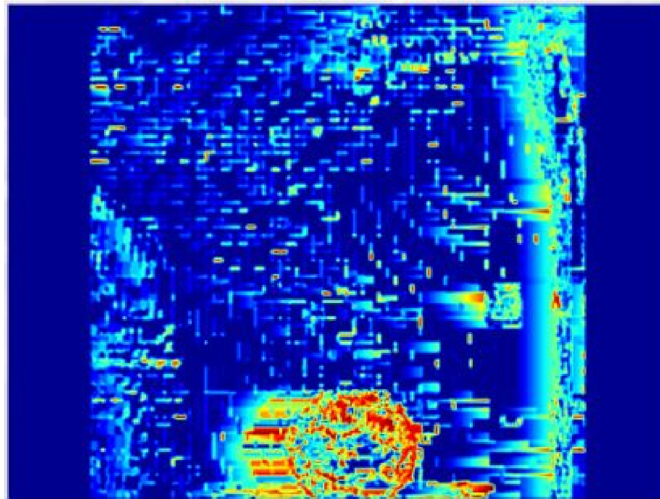

Figura 7.14: Mapa de disparidad obtenido con la versión definitiva del algoritmo de Lankton para imágenes reales

7.2.4 Resultados obtenidos para la detección de objetos

Los siguientes resultados se han obtenido para las imágenes de la figura 7.4. Todos los resultados obtenidos para la detección de los objetos se han realizado con los parámetros: Tipo de Píxel: "Bote"; Umbral: por defecto "20".

Como se puede observar en la imagen de la figura 7.15 el punto marcado por la señal verde en la imagen no está exactamente en el centro del bote sino desplazado hacía la parte inferior. Eso es debido a que en la imagen el bote aparece reflejado en el suelo y el algoritmo lo considera como parte del bote.

También podemos observar los resultados de disparidad para el bote. Tenemos una disparidad máxima de 56 en la zona del bote y una disparidad media de 23.

Figura 7.15: Resultado obtenido a partir del algoritmo de detección de objetos

Los siguientes resultados se han obtenido para las imágenes de la figura 7.5.

Figura 7.16: Resultado obtenido a partir del algoritmo de detección de objetos

Como se puede observar en la imagen de la figura 7.16 en este caso el punto marcado en la imagen esta desplazado ligeramente hacia la parte superior del bote con respecto al centro del mismo. Esto es debido a que el parecido de los píxeles de la parte inferior del bote con el píxel que le hemos pasado como referencia no es muy alto y el valor de similitud obtenido a partir de la ecuación 6.1 es mayor que el offset introducido (20) y por tanto estos píxeles no pertenecen a la región que se computa como bote en el algoritmo. También en esta imagen podemos observar que el reflejo del bote en el suelo no influye por el motivo previamente explicado. Además podemos observar los resultados de disparidad para el bote. Tenemos una disparidad máxima de 80 en la zona del bote y una disparidad media de 37.

Los resultados obtenidos 7.15 y 7.16 son suficientemente buenos para el fin propuesto ya que sólo necesitábamos saber si existía algún obstáculo de esas características en la escena 3D para informar al algoritmo de navegación del otro grupo coordinado.

Apéndice A

Diseño técnico completo. Diagramas de secuencia

A.1 Diagramas del algoritmo de correlación

En este apéndice, se presentan los siguientes diagramas de secuencia relacionados con el algoritmo de correlación en el siguiente orden:

- *Algoritmo*
- *rellenarVentana*
- *calcularDistancia*
- *calcularMedia*
- *varianza*
- *covarianza*

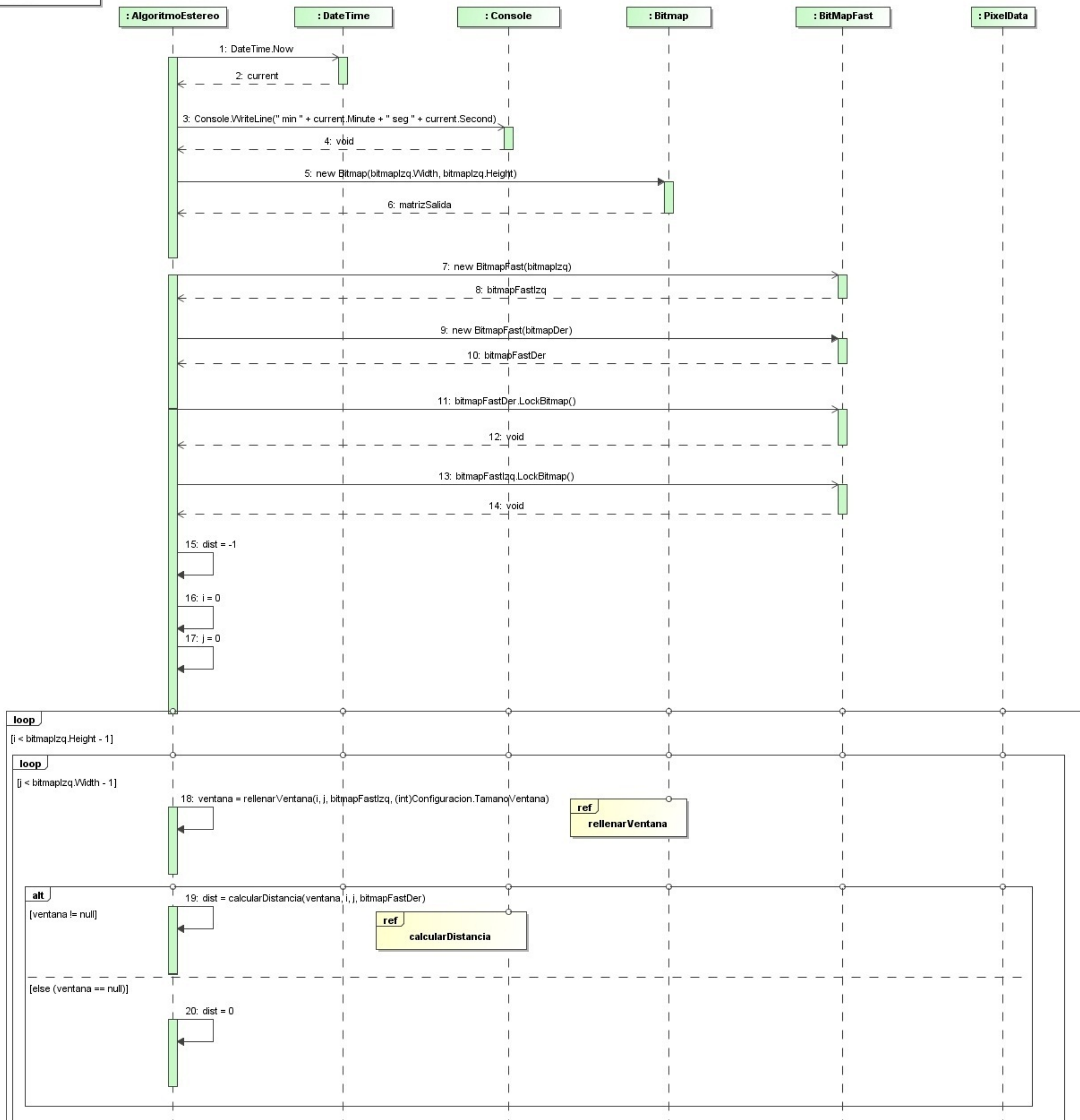

Figura A.1: Algoritmo de correlación. Primera parte de la función *Algoritmo*

Figura A.2: Algoritmo de correlación. Segunda parte de la función *Algoritmo*

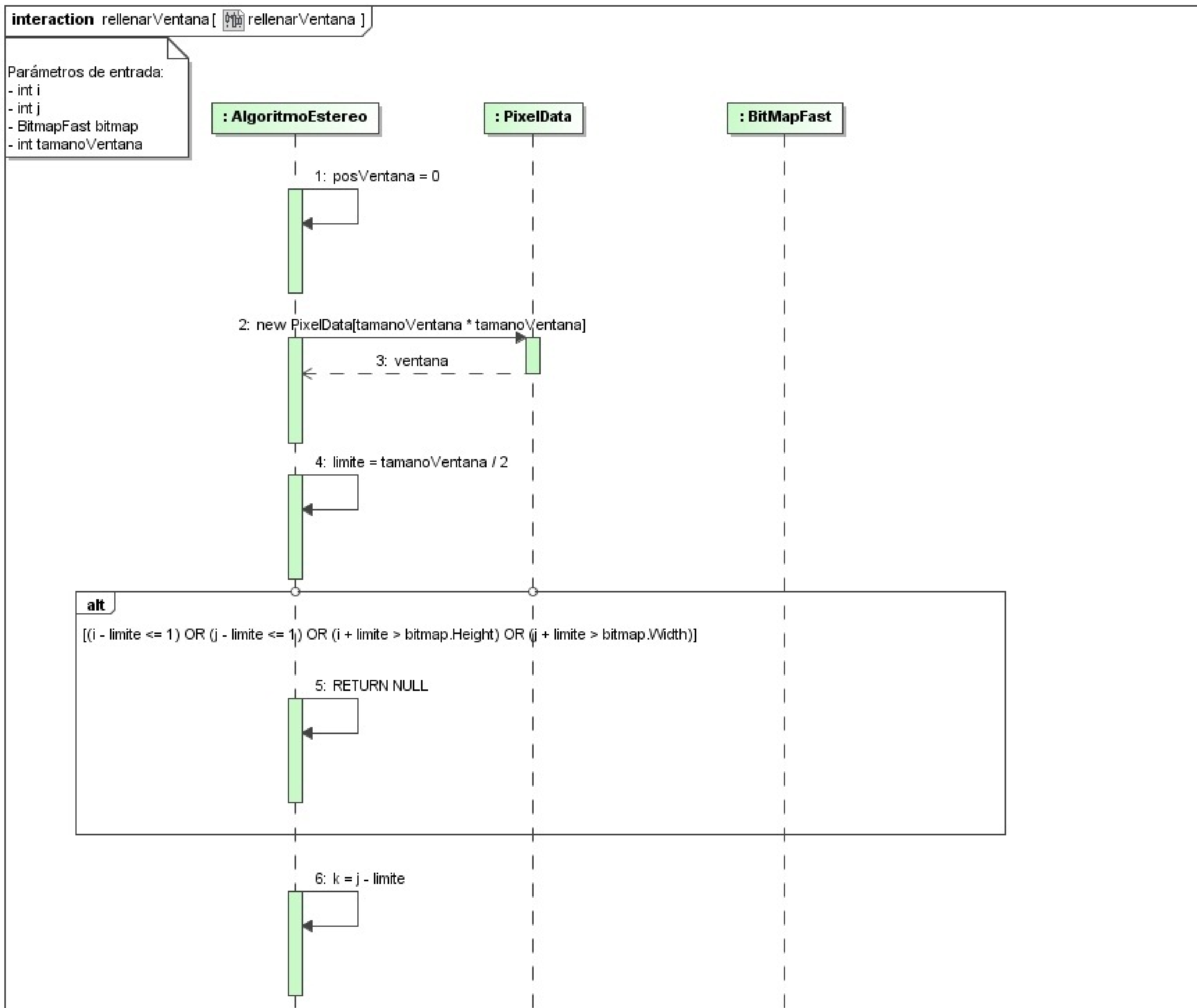

Figura A.3: Algoritmo de correlación. Primera parte de la función *rellenarVentana*

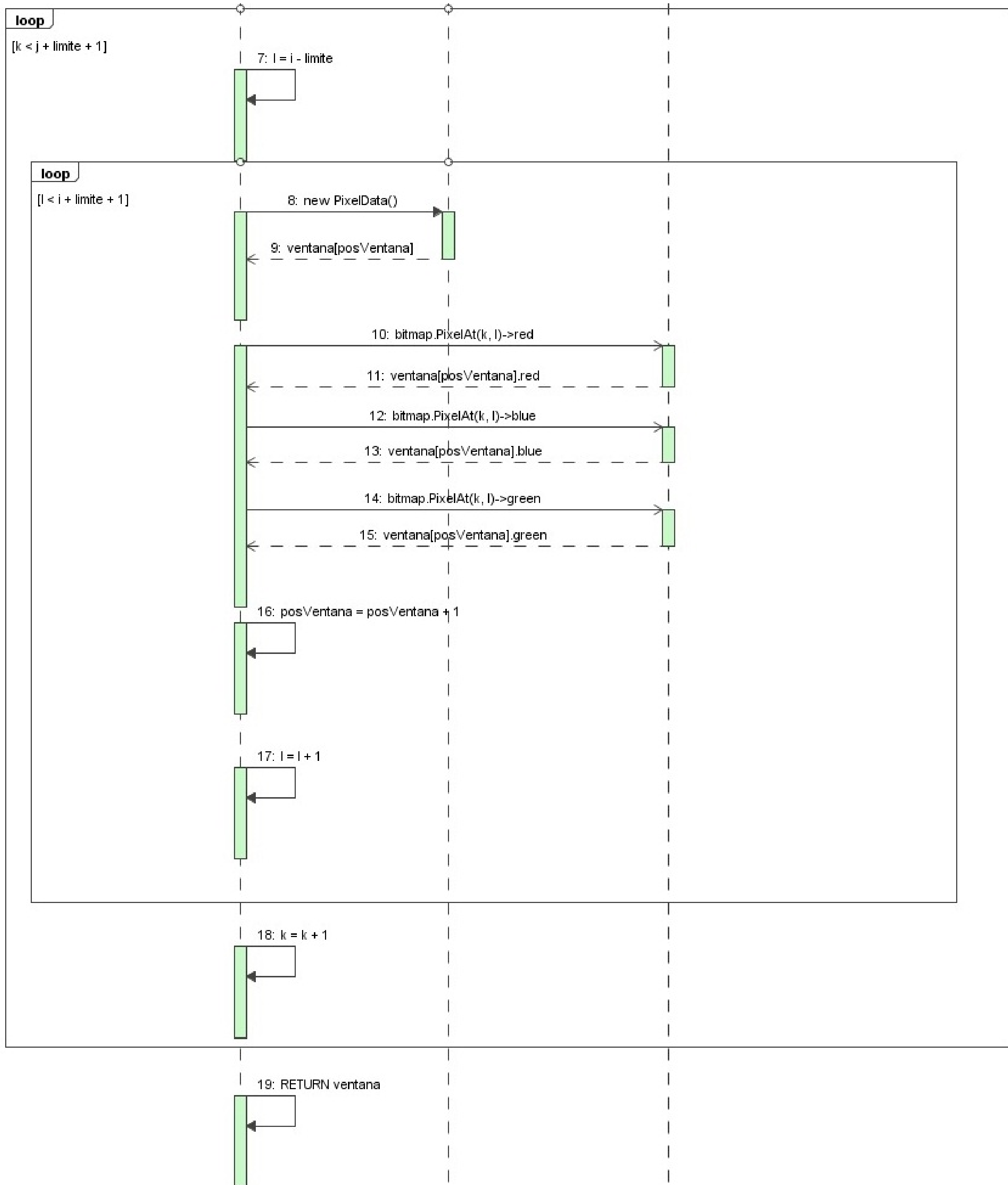

Figura A.4: Algoritmo de correlación. Segunda parte de la función *rellenarVentana*

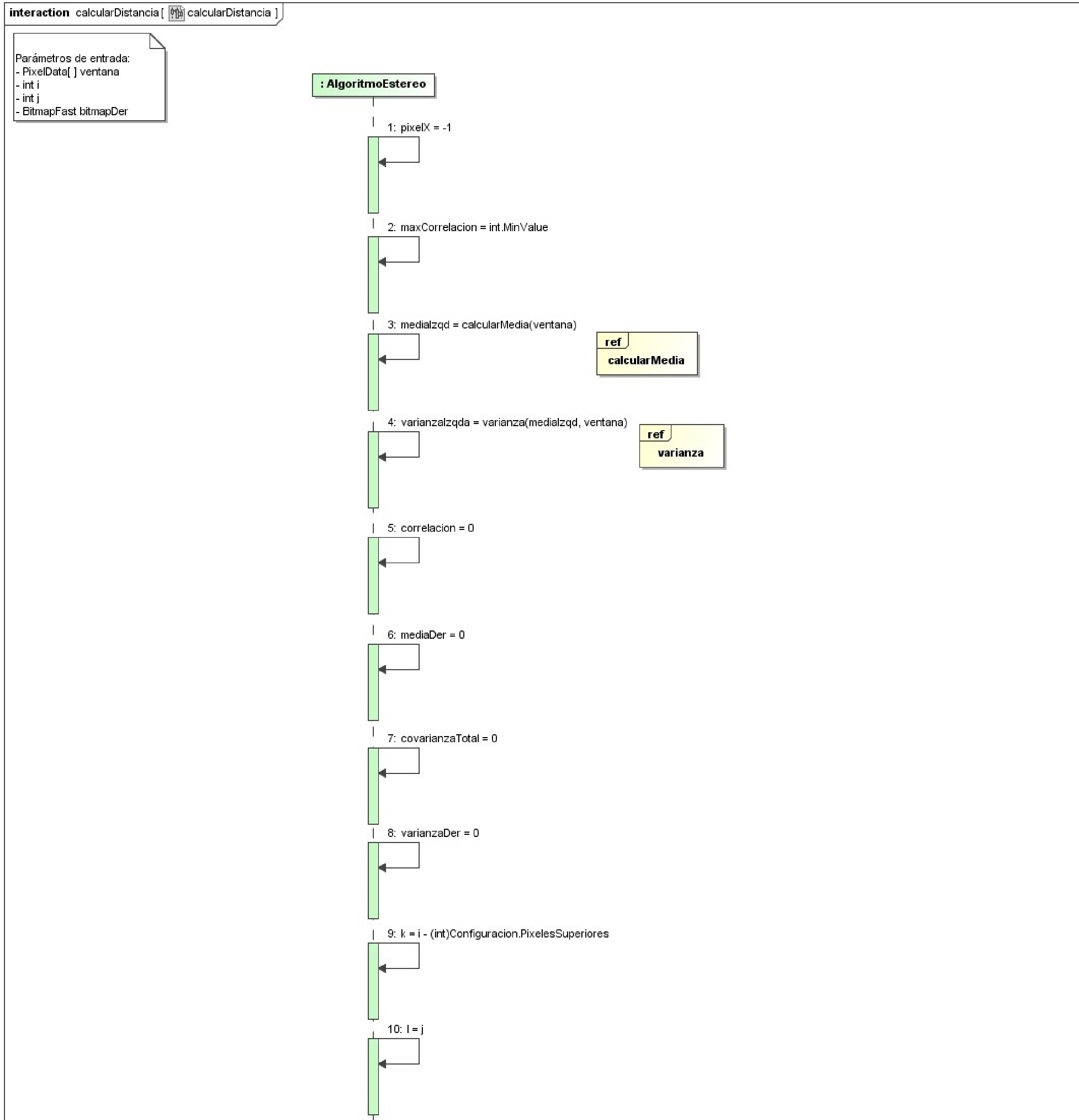

Figura A.5: Algoritmo de correlación. Primera parte de la función *calcularDistancia*

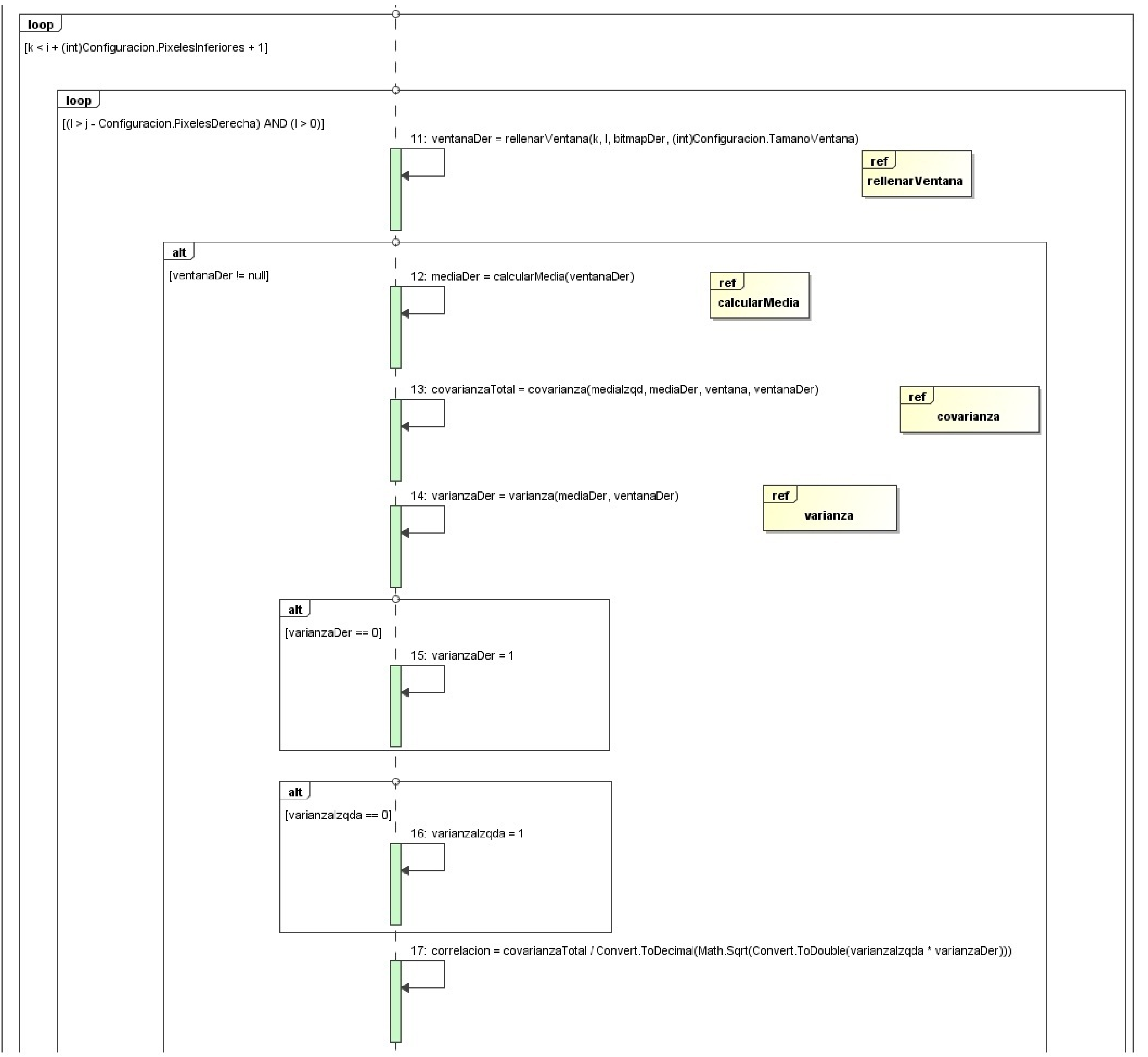

Figura A.6: Algoritmo de correlación. Segunda parte de la función *calcularDistancia*

Figura A.7: Algoritmo de correlación. Tercera parte de la función *calcularDistancia*

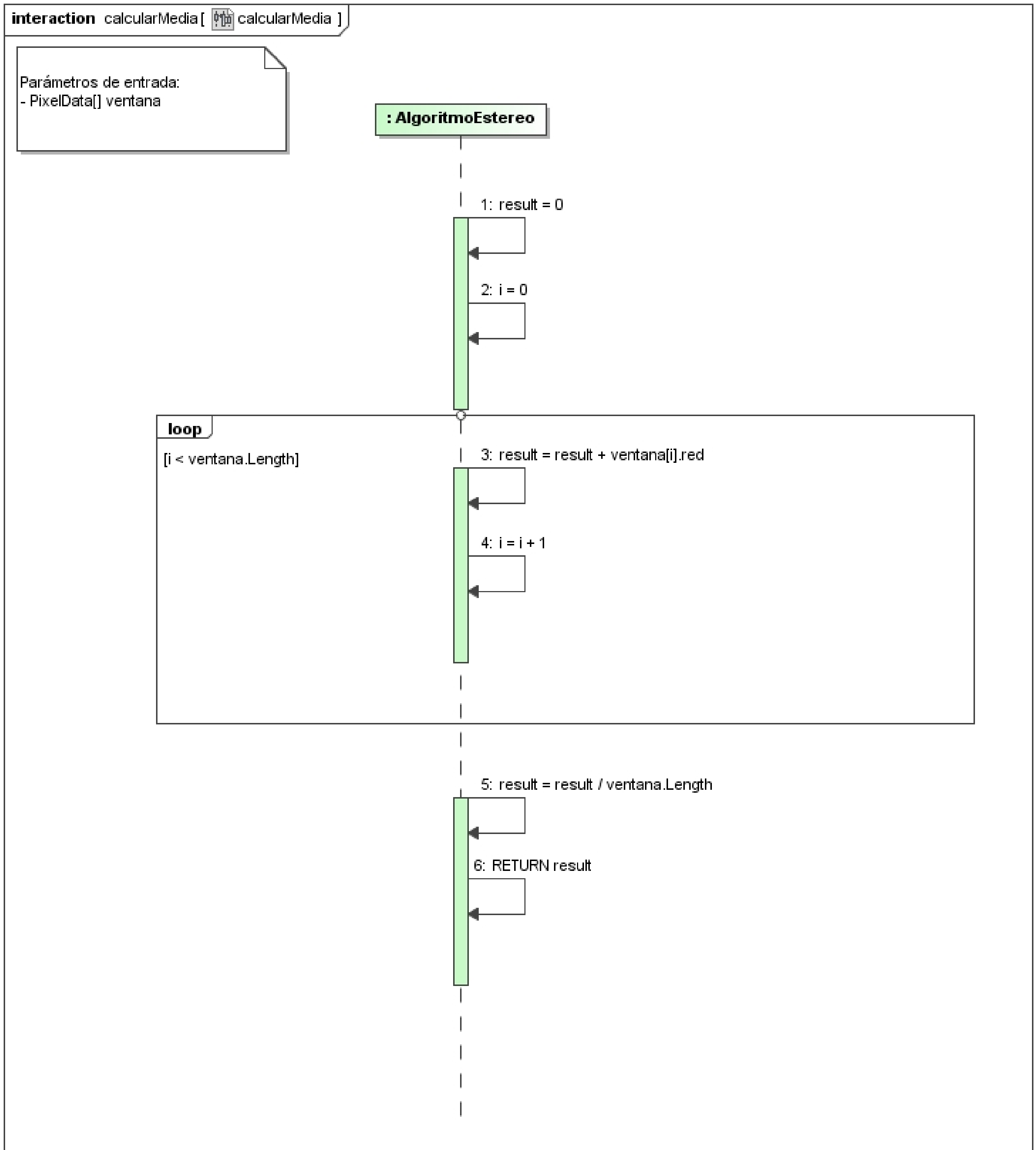

Figura A.8: Algoritmo de correlación. Función *calcularMedia*

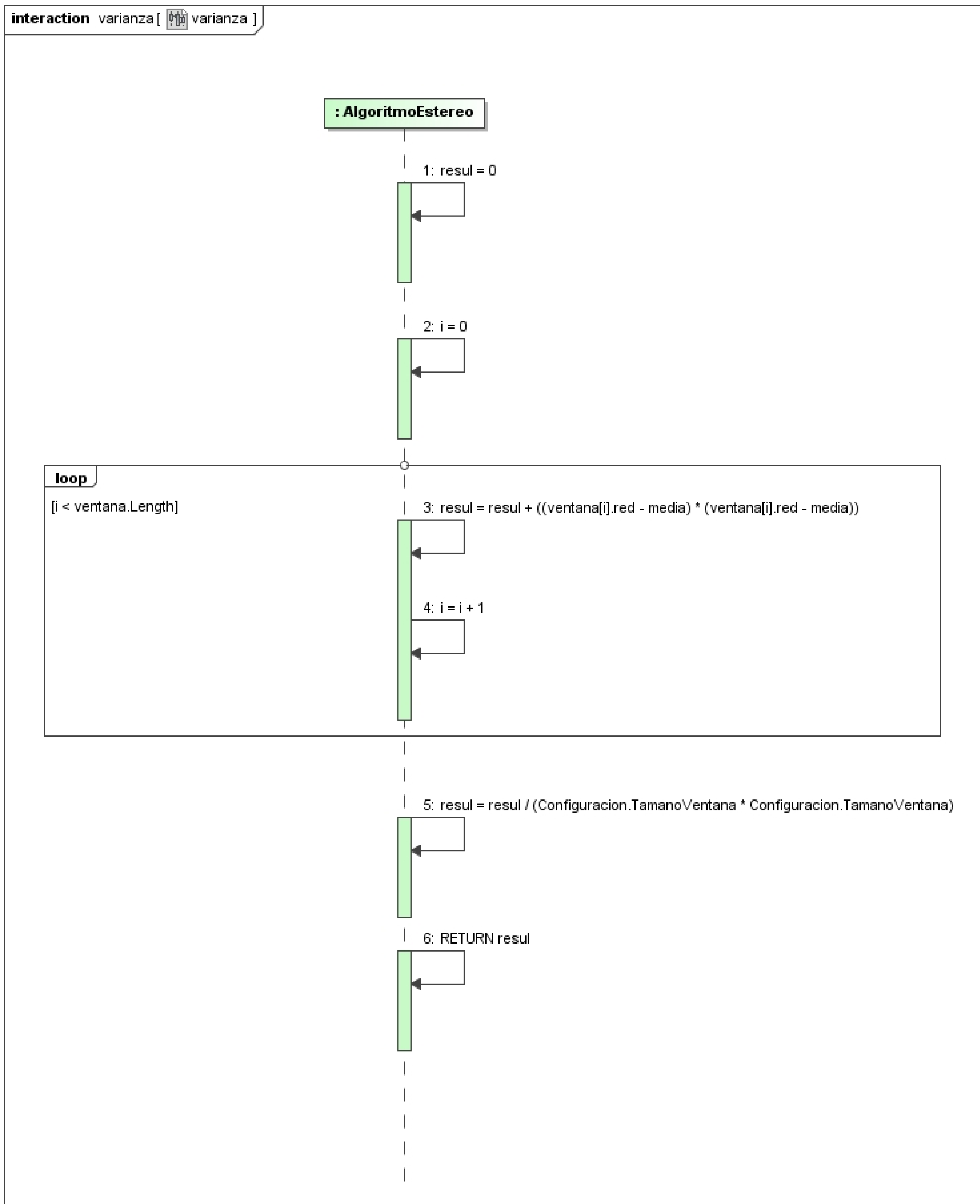

Figura A.9: Algoritmo de correlación. Función *varianza*

interaction covarianza [covarianza]

Parámetros de entrada:
- decimal mediaIzqd
- decimal mediaDer
- PixelData[] ventanaIzqd
- PixelData[] ventanaDer

: AlgoritmoEstereo

1: resul = 0

2: i = 0

loop

[i < ventanaDer.Length]

3: resul = resul + ((ventanaIzqd[i].red - mediaIzqd) * (ventanaDer[i].red - mediaDer))

4: i = i + 1

5: resul = resul / (Configuracion.TamanoVentana * Configuracion.TamanoVentana)

6: RETURN resul

Figura A.10: Algoritmo de correlación. Función *covarianza*

A.2 Diagramas del algoritmo de Lankton y de su versión mejorada

En este apartado, podemos contemplar los diagramas de las funciones desarrolladas para el correcto funcionamiento del algoritmo de visión estereoscópica de Lankton. Aparecen los siguientes diagramas en el mismo orden que se presentan en la lista:

- *NuevoAlgoritmo*
- *DesplazarVerticalmente*
- *DeslizarImagenes*
- *elegirMejorResultado*
- *BitmapToDouble*:

  Pasa la imagen a una matriz de números cogiendo la componente roja de cada pixel.
- *clone*:

  Esta función crea una copia de la imagen para no modificar la original.
- *DesplazarImagenes*:

  Se utiliza para desplazar las imágenes tantos píxeles como se desee hacia arriba o hacia abajo. Llama a la función *Scroll*.
- *ElevarAlCuadrado*:

  Sirve para elevar al cuadrado todas las componentes de los pixeles de una imagen dada.
- *Scroll*:

  Se encarga de desplazar la imagen a la izquierda (o a la derecha) y hacia arriba (o hacia abajo) tantos píxeles como le indiquemos con los parámetros *dx* y *dy*.
- *winnerTakeAll*:

  Se encarga de seleccionar el mejor deslizamiento de una imagen sobre la otra teniendo en cuenta la disparidad mínima que se permite.

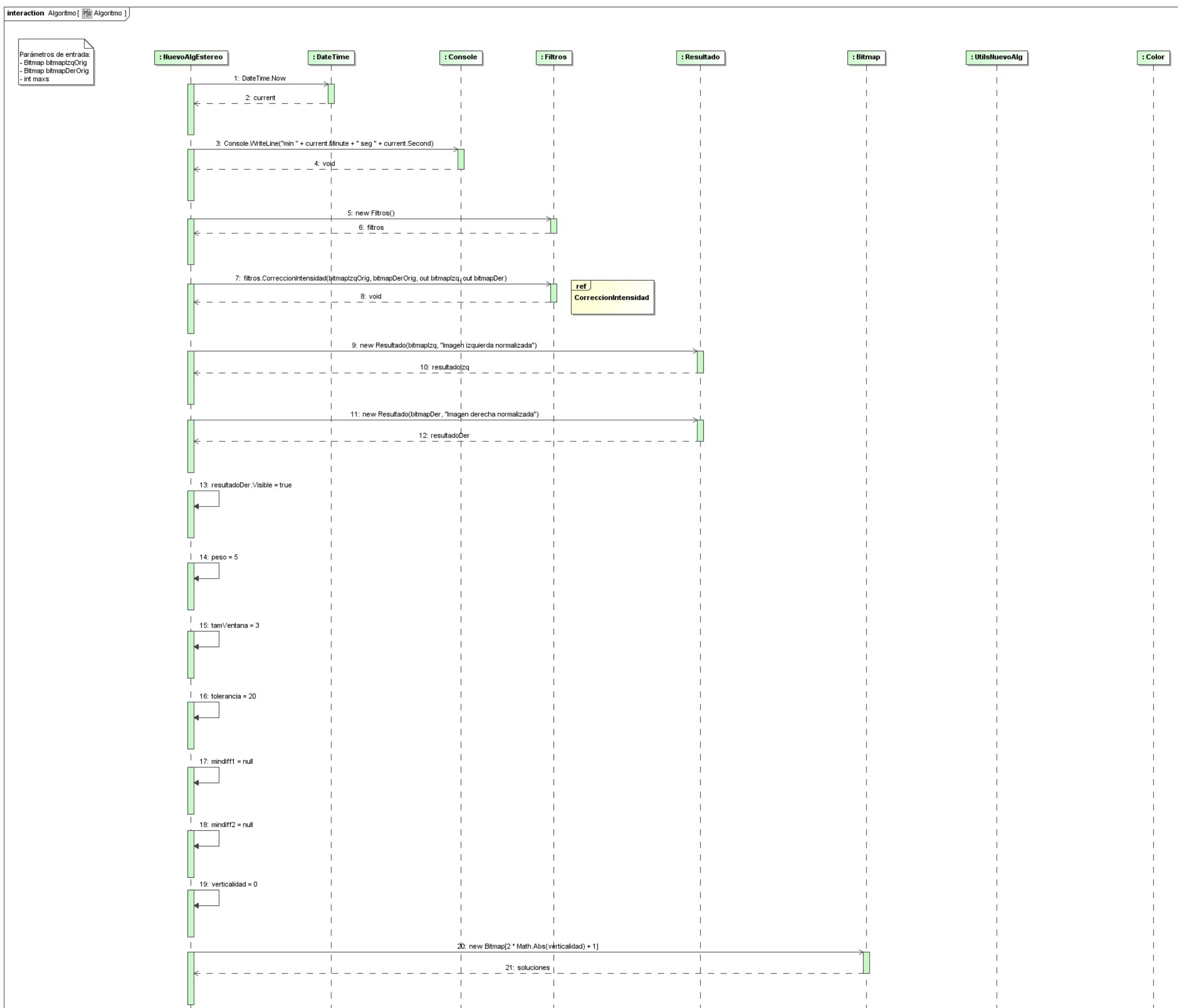

Figura A.11: Algoritmo de Lankton. Primera parte de la función *NuevoAlgoritmo*

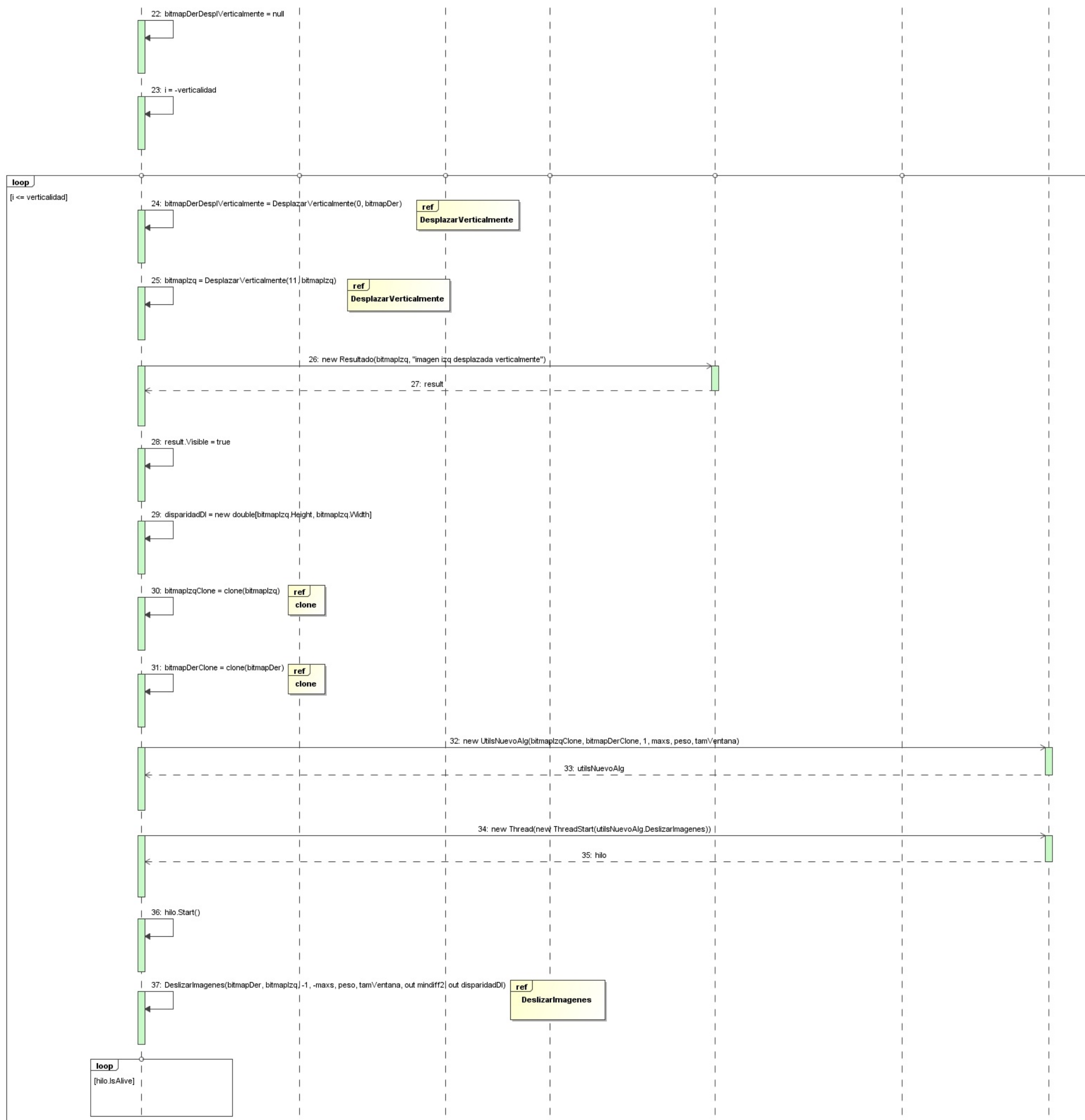

Figura A.12: Algoritmo de Lankton. Segunda parte de la función *NuevoAlgoritmo*

Figura A.13: Algoritmo de Lankton. Tercera parte de la función *NuevoAlgoritmo*

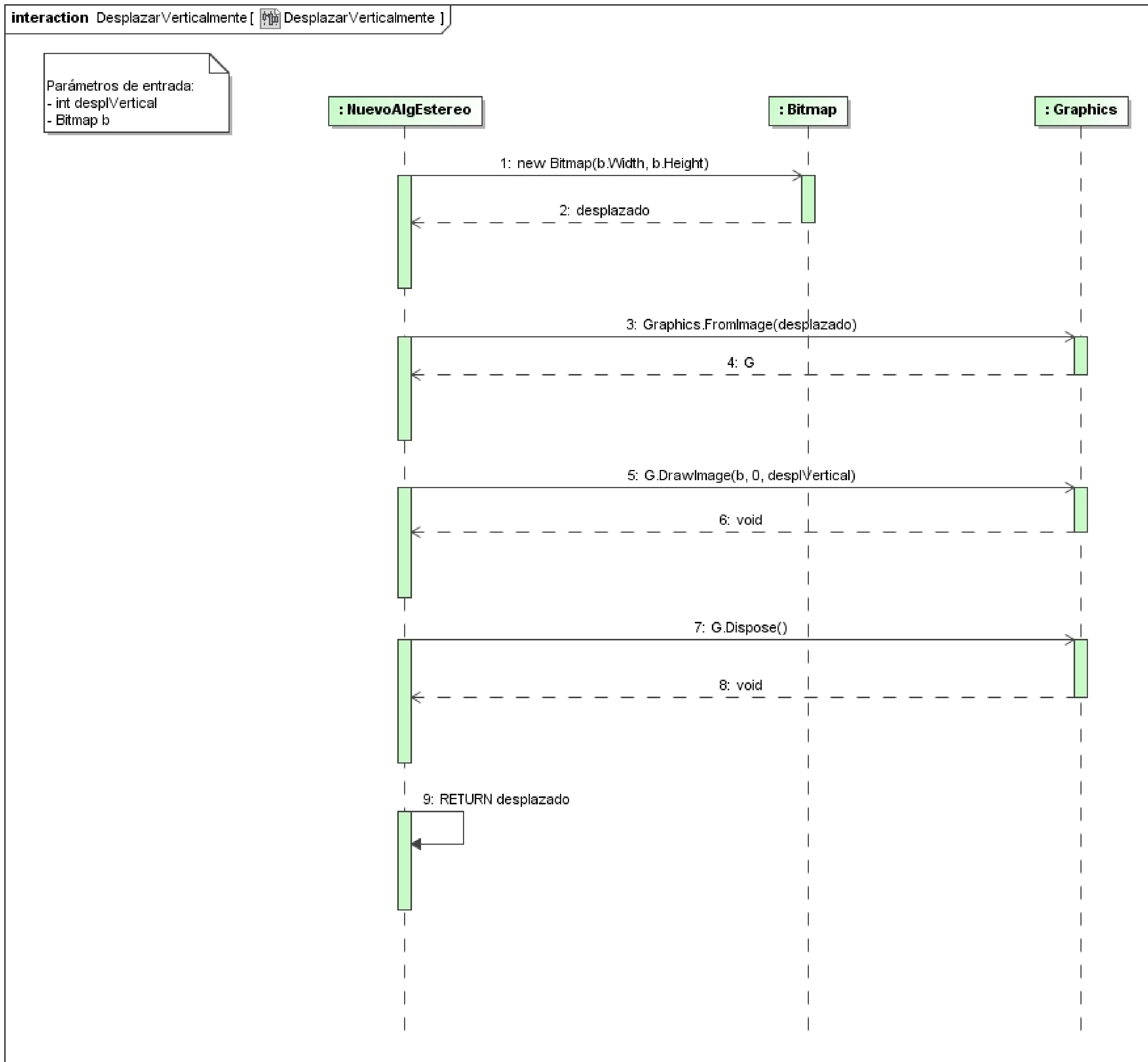

Figura A.14: Algoritmo de Lankton. Función *DesplazarVerticalmente*

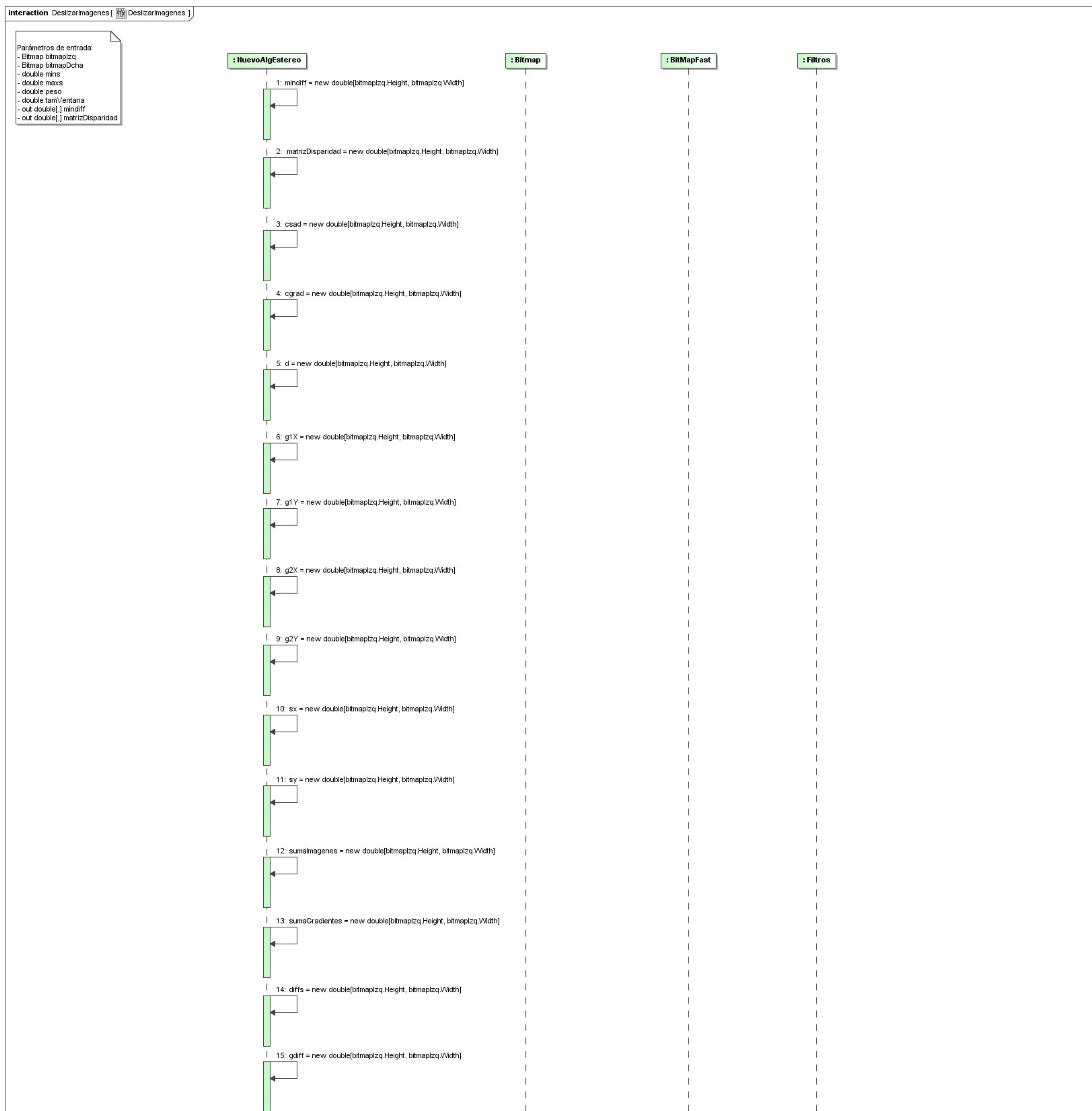

Figura A.15: Algoritmo de Lankton. Primera parte de la función *DeslizarImagenes*

16: new Bitmap(bitmapIzq)
17: izqClone
18: new Bitmap(bitmapDcha)
19: drchClone
20: new BitmapFast(drchClone)
21: bitmapFastDcha
22: bitmapFastDcha.LockBitmap()
ref LockBitmap
23: void
24: new BitmapFast(izqClone)
25: bitmapFastIzq
26: bitmapFastIzq.LockBitmap()
ref LockBitmap
27: void
28: aux = null
29: bitmapDesplazado = null
30: h = new double[bitmapIzq.Height, bitmapIzq.Width]
31: i = 0
32: j = 0
loop [i < bitmapIzq.Height]
loop [j < bitmapIzq.Width]
33: mindiff[i, j] = Double.MaxValue
34: h[i, j] = 1 / Math.Pow(tamVentana, 2)
35: j = j + 1
36: i = i + 1

Figura A.16: Algoritmo de Lankton. Segunda parte de la función *DeslizarImagenes*

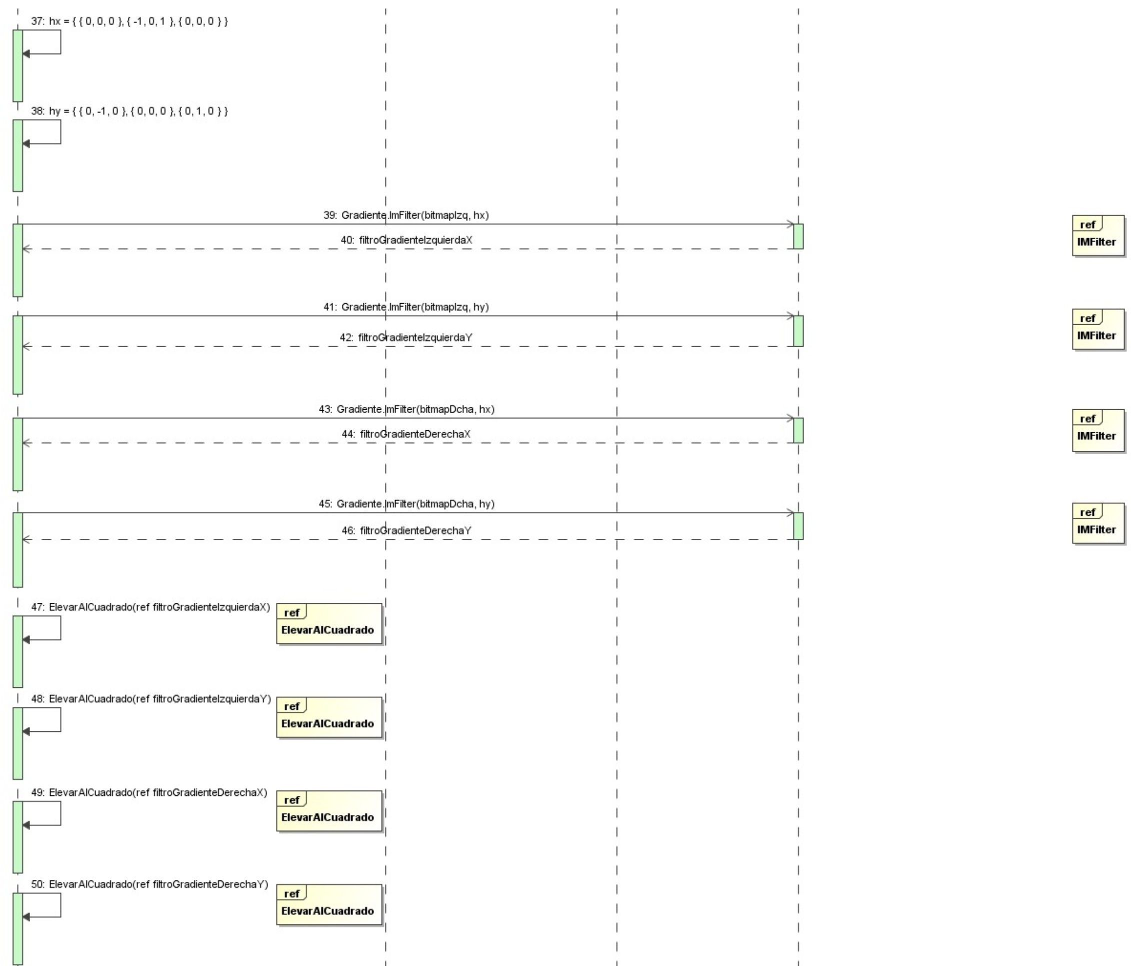

Figura A.17: Algoritmo de Lankton. Tercera parte de la función *DeslizarImagenes*

Figura A.18: Algoritmo de Lankton. Cuarta parte de la función *DeslizarImagenes*

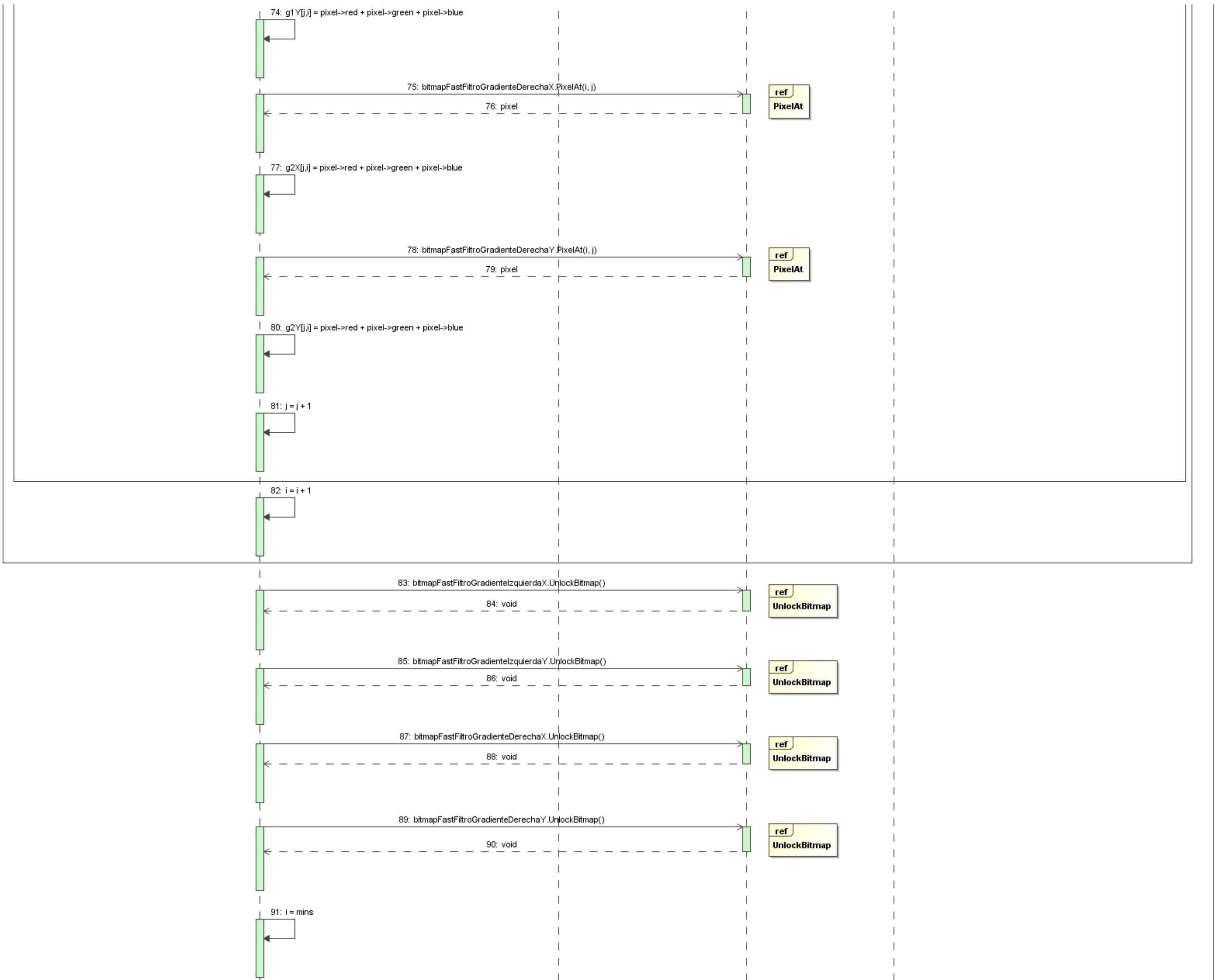

Figura A.19: Algoritmo de Lankton. Quinta parte de la función *DeslizarImagenes*

Figura A.20: Algoritmo de Lankton. Sexta parte de la función *DeslizarImagenes*

Figura A.21: Algoritmo de Lankton. Séptima parte de la función *DeslizarImagenes*

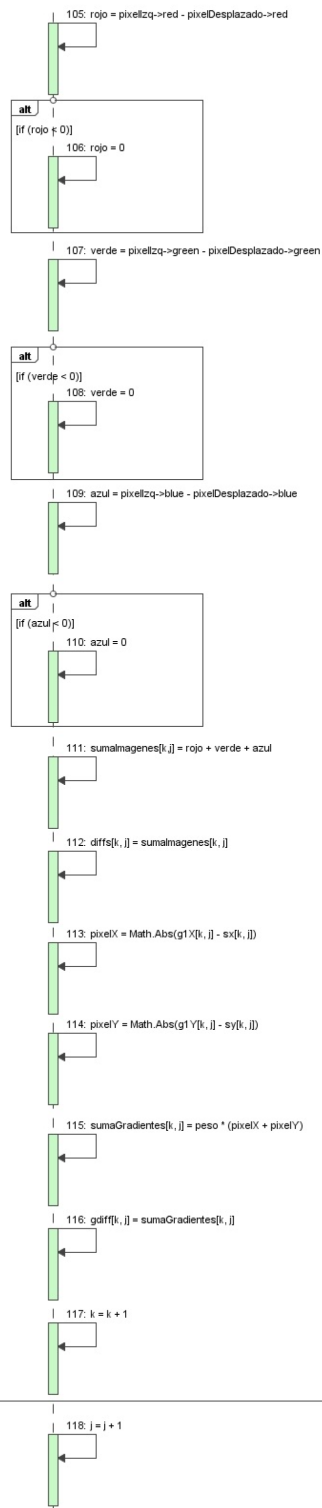

Figura A.22: Algoritmo de Lankton. Octava parte de la función *DeslizarImagenes*

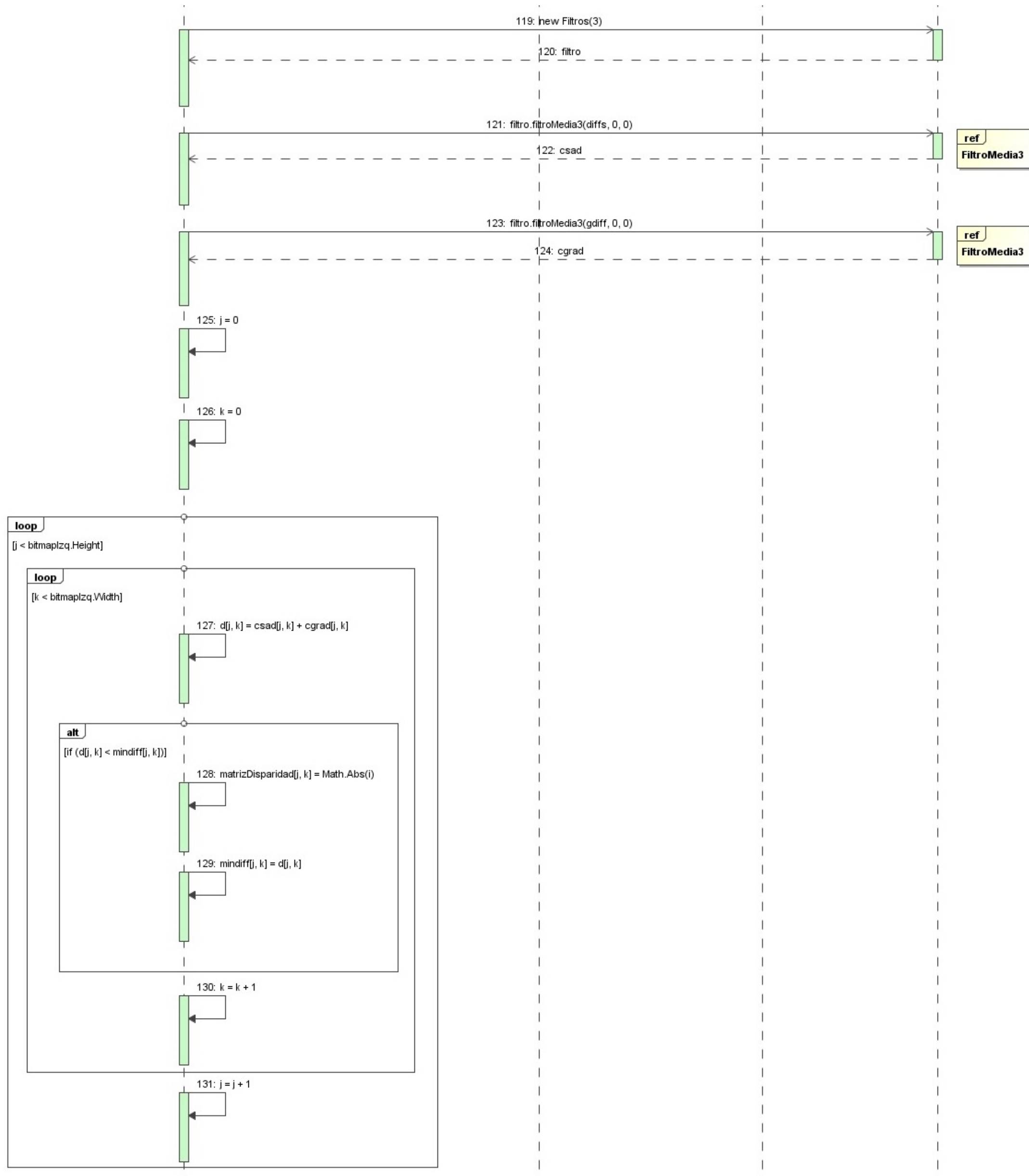

Figura A.23: Algoritmo de Lankton. Novena parte de la función *DeslizarImagenes*

Figura A.24: Algoritmo de Lankton. Décima parte de la función *DeslizarImagenes*

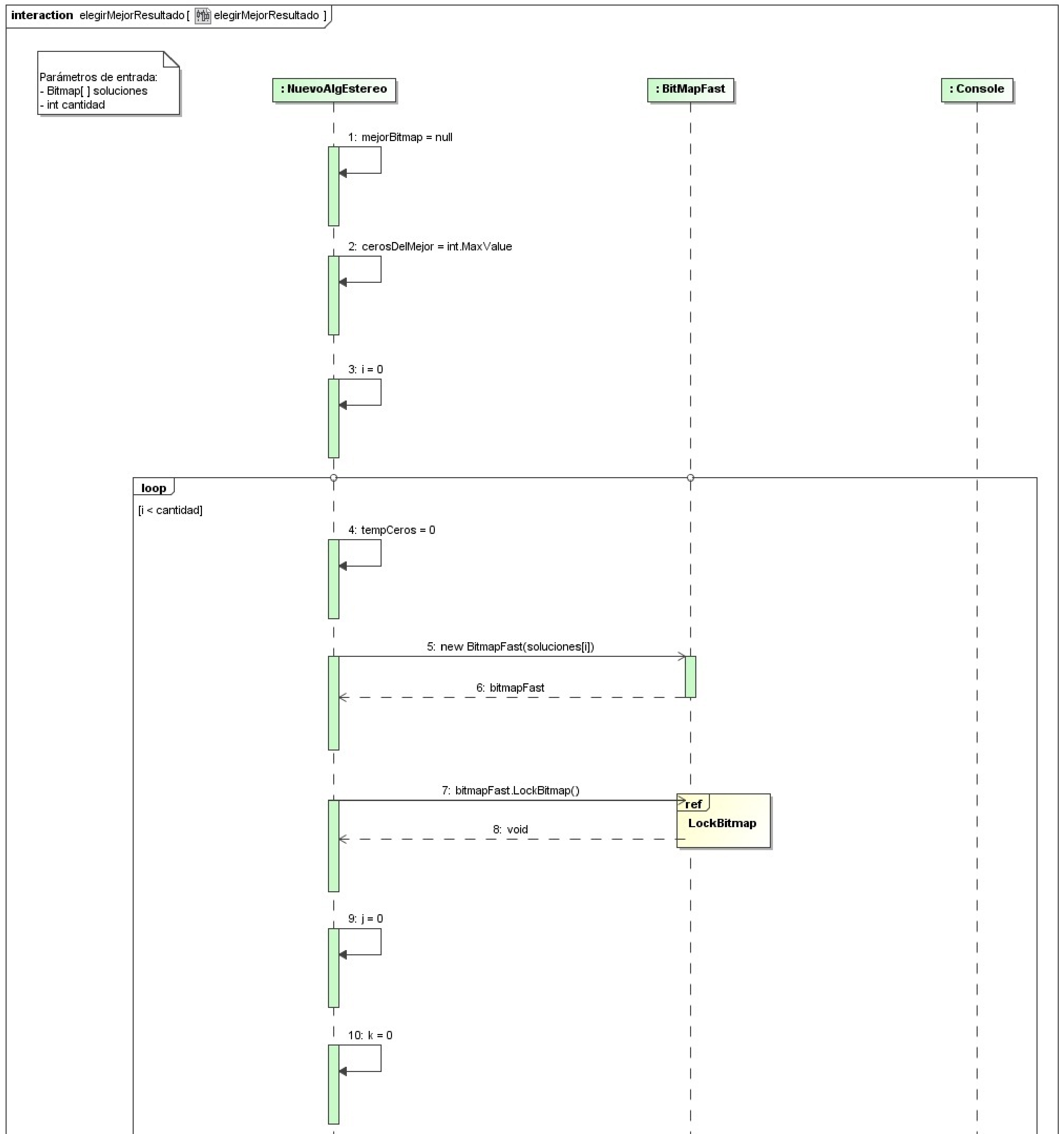

Figura A.25: Algoritmo de Lankton. Primera parte de la función *elegirMejorResultado*

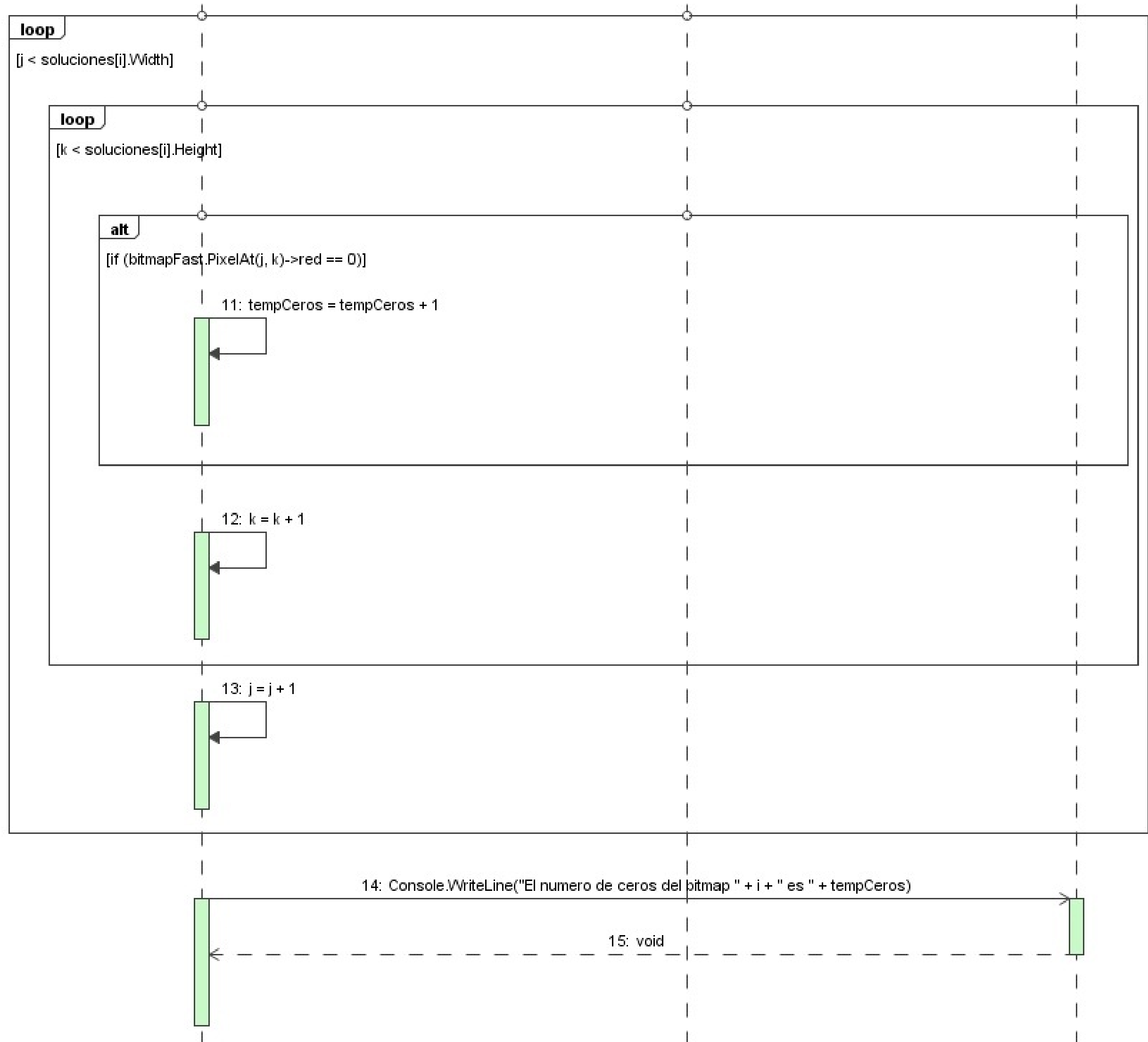

Figura A.26: Algoritmo de Lankton. Segunda parte de la función *elegirMejorResultado*

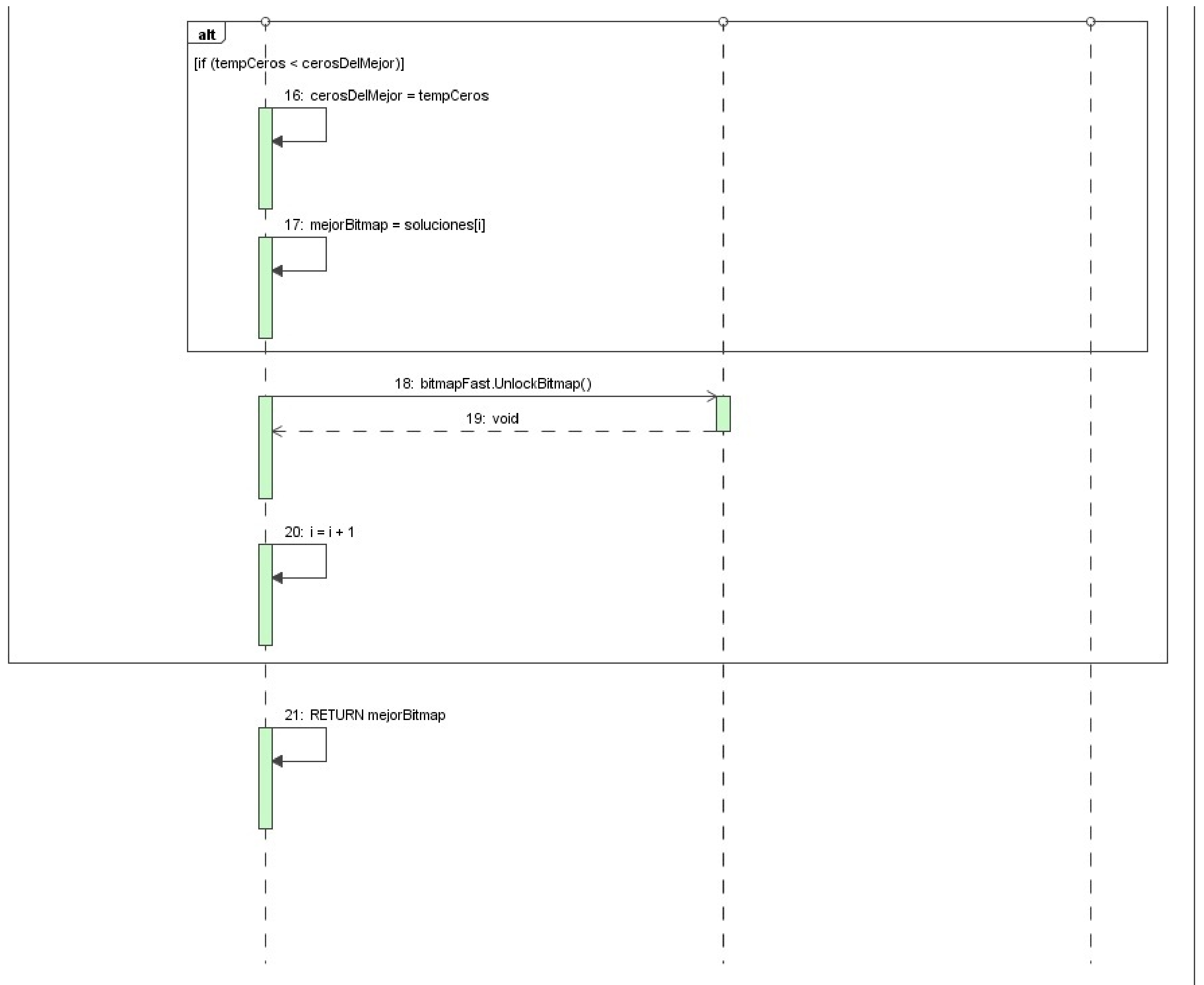

Figura A.27: Algoritmo de Lankton. Tercera parte de la función *elegirMejorResultado*

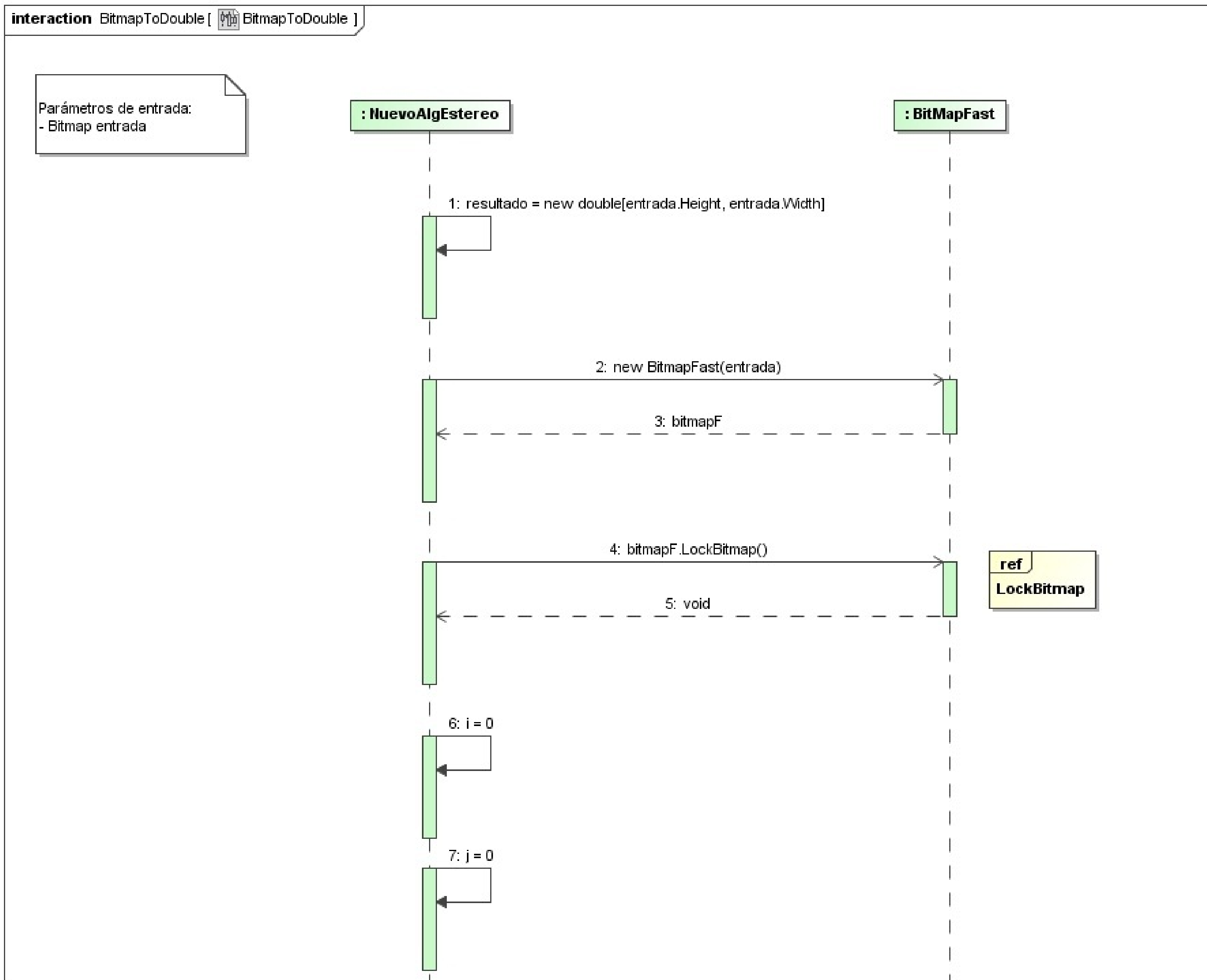

Figura A.28: Algoritmo de Lankton. Primera parte de la función *BitmapToDouble*.

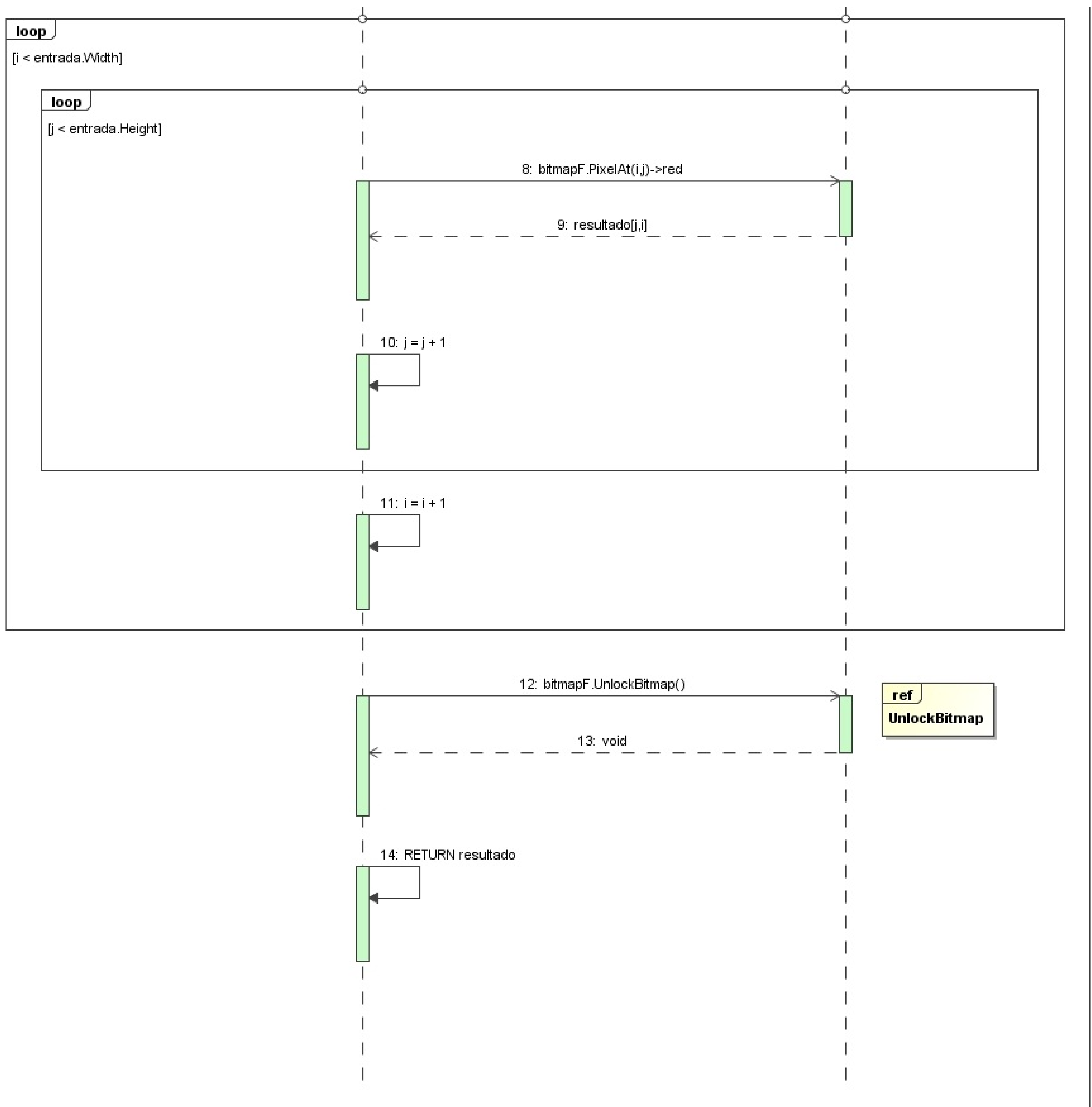

Figura A.29: Algoritmo de Lankton. Segunda parte de la función *BitmapToDouble*.

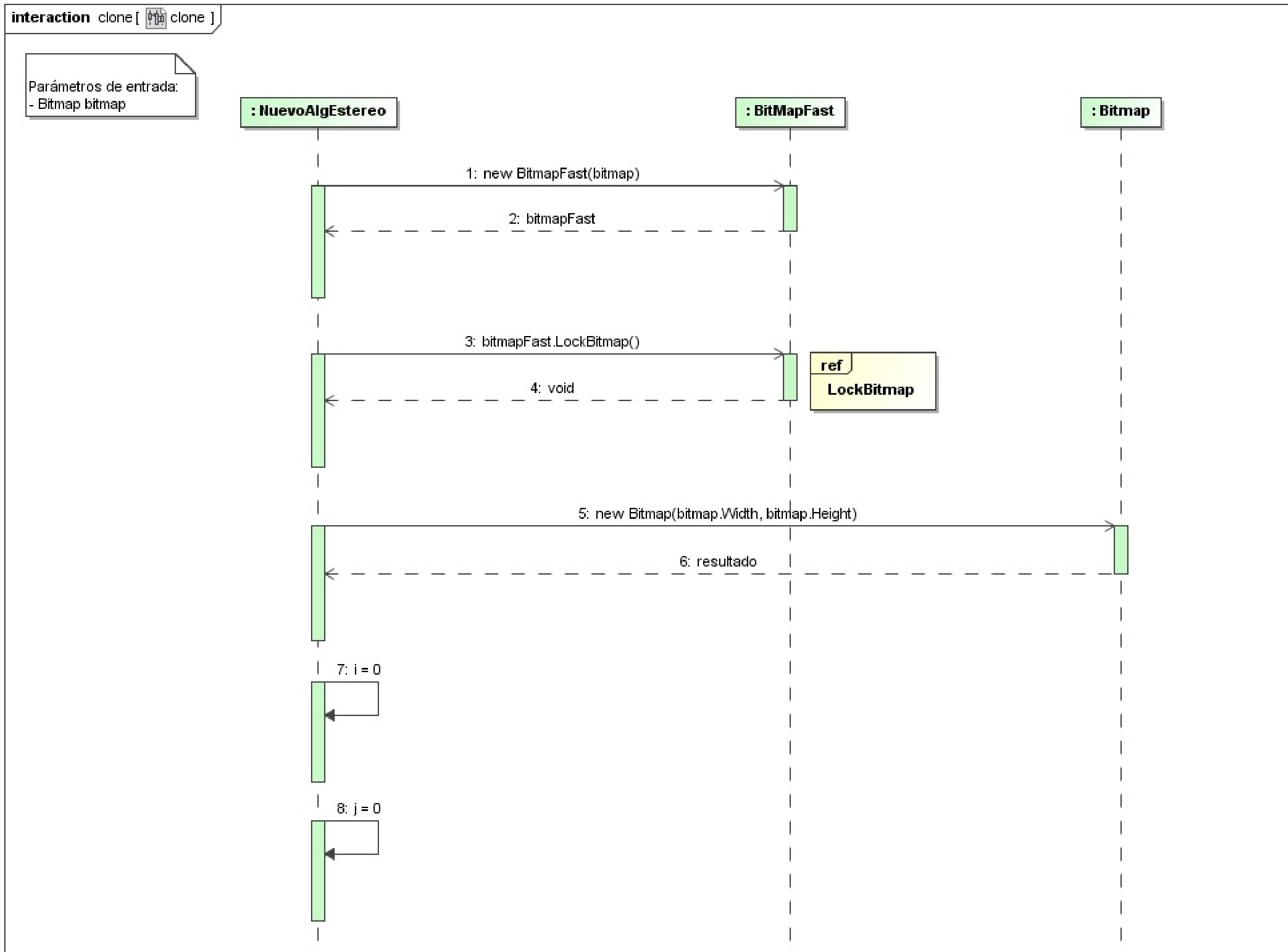

Figura A.30: Algoritmo de Lankton. Primera parte de la función *clone*.

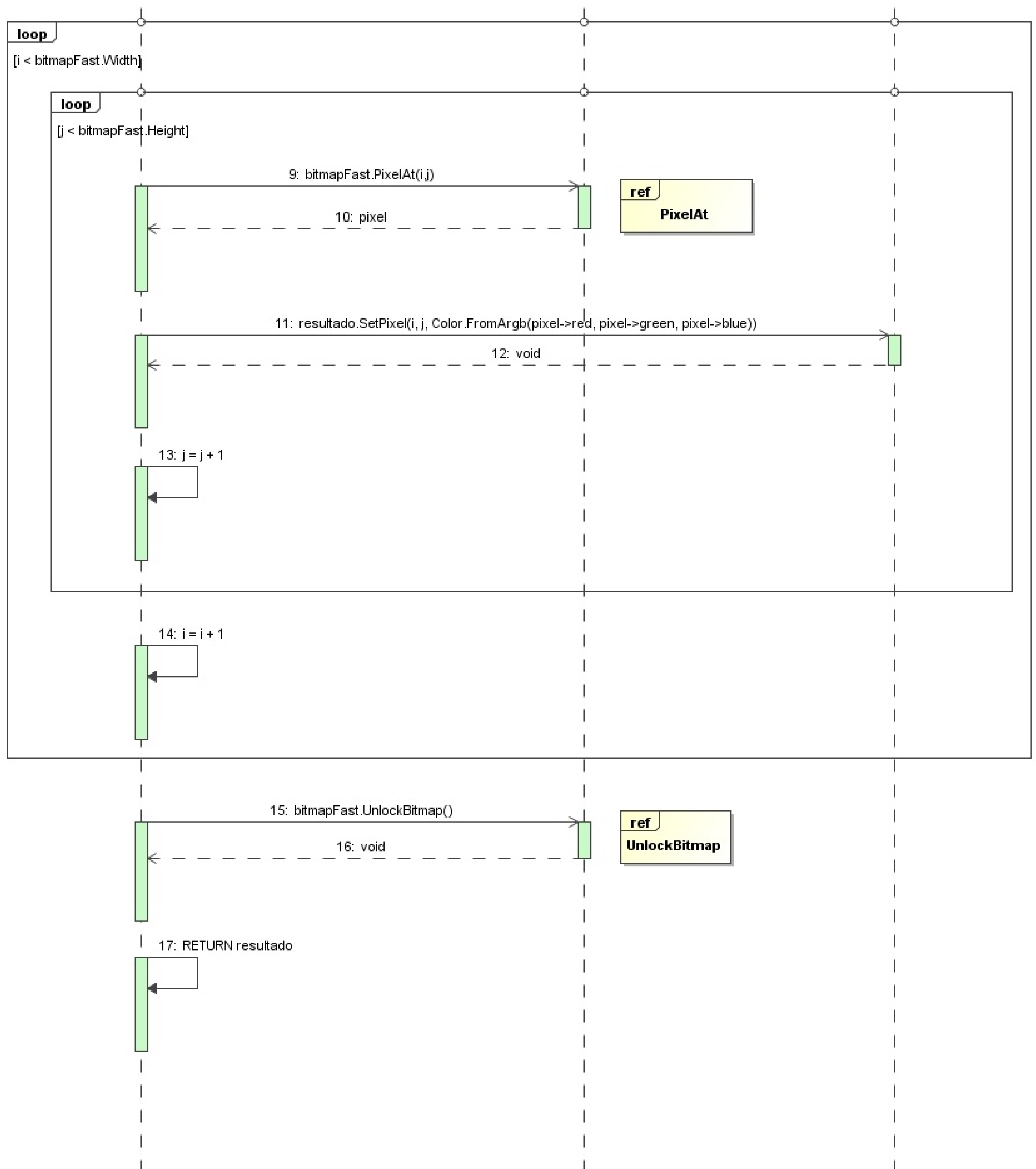

Figura A.31: Algoritmo de Lankton. Segunda parte de la función *clone*.

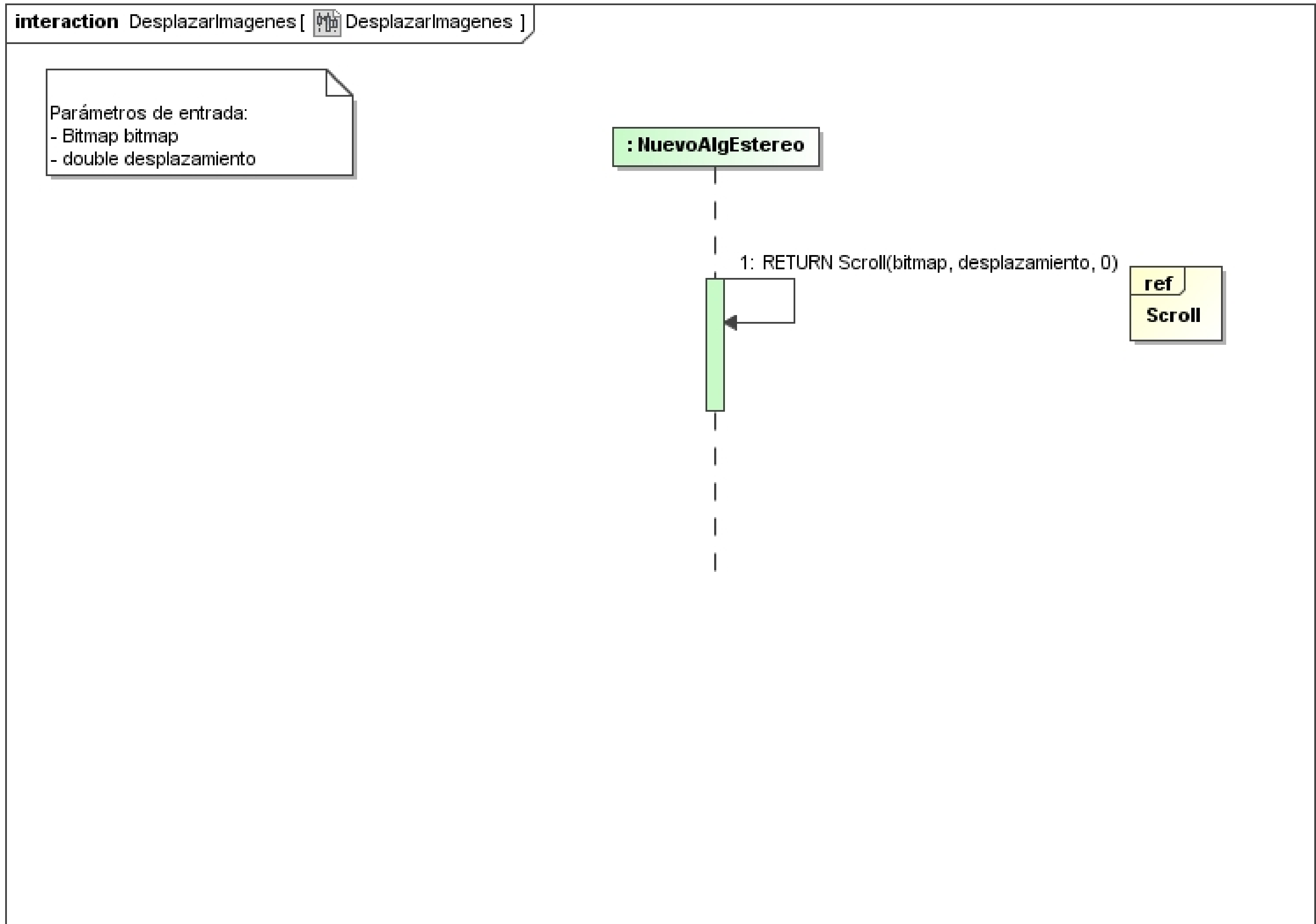

Figura A.32: Algoritmo de Lankton. Función *DesplazarImagenes*.

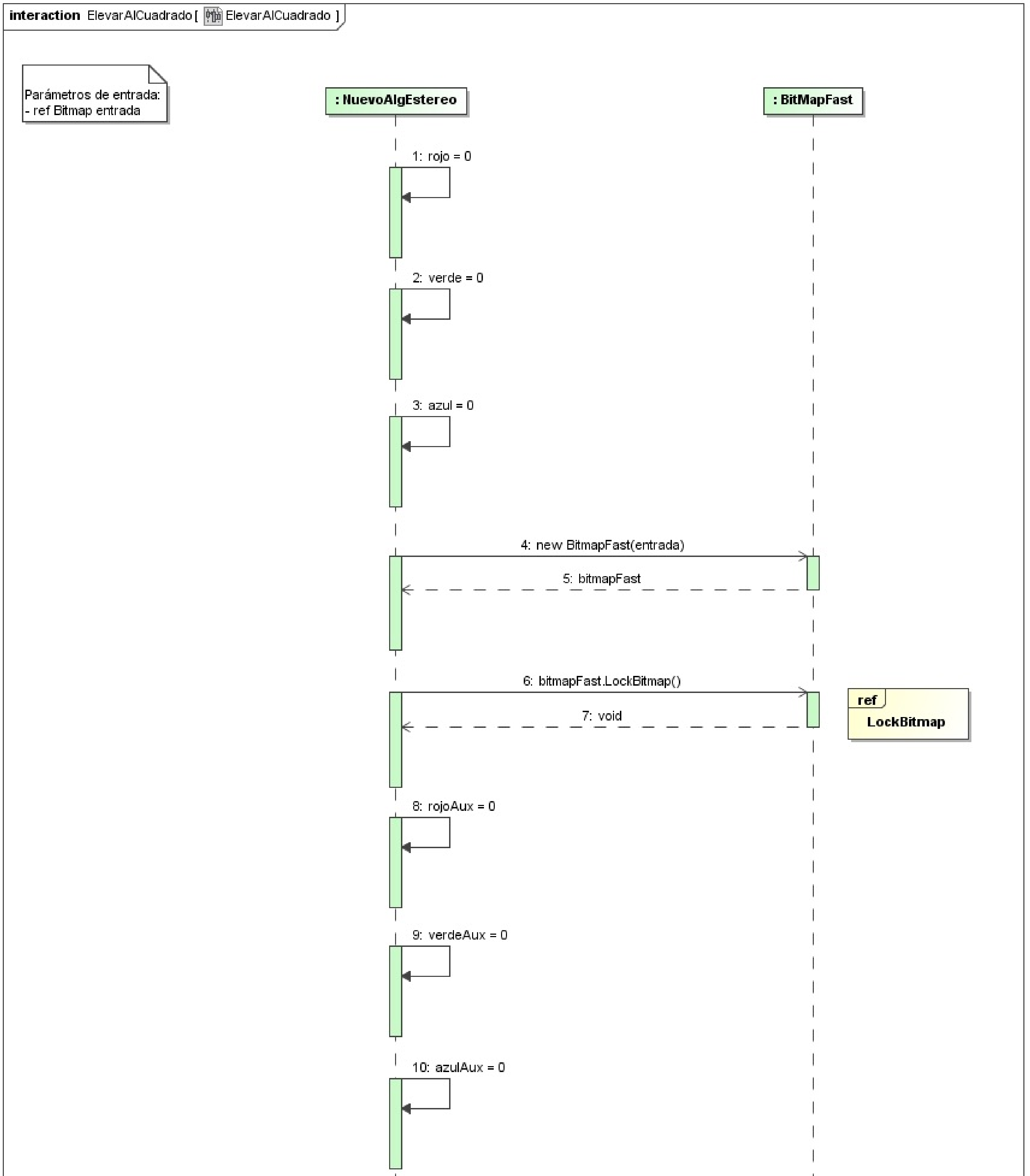

Figura A.33: Algoritmo de Lankton. Primera parte de la función *ElevarAlCuadrado*.

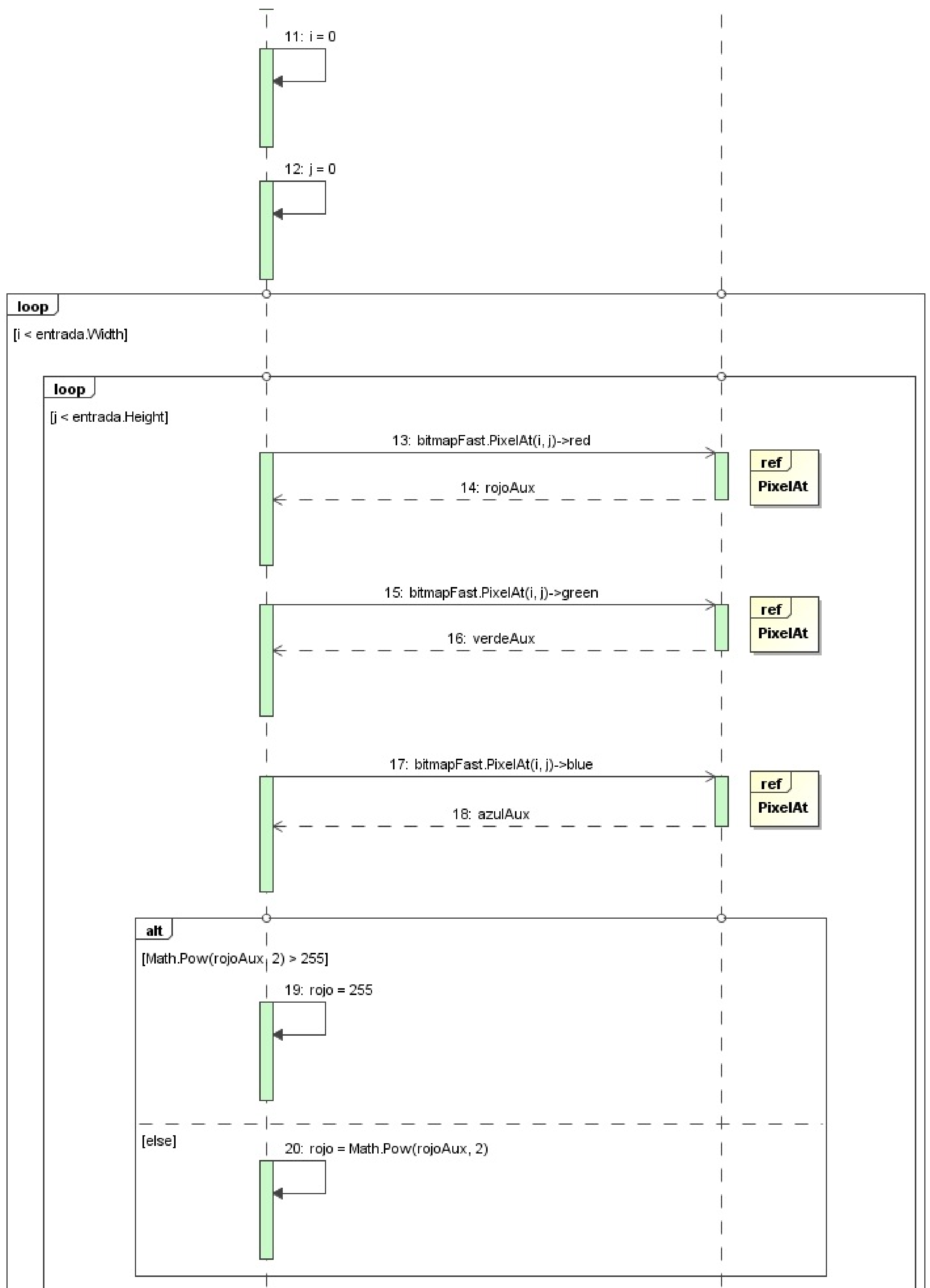

Figura A.34: Algoritmo de Lankton. Segunda parte de la función *ElevarAlCuadrado*

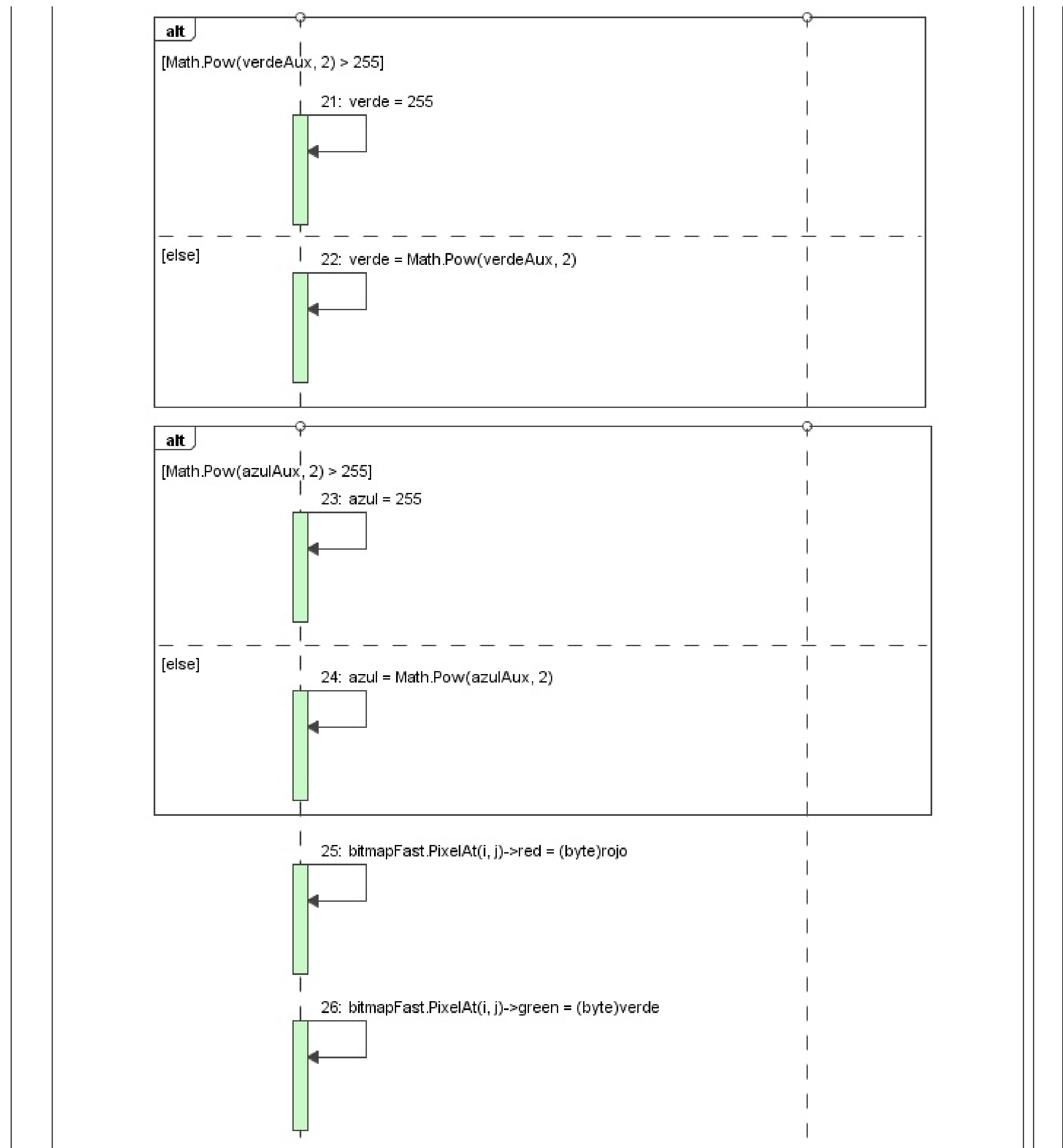

Figura A.35: Algoritmo de Lankton. Tercera parte de la función *ElevarAlCuadrado*

Figura A.36: Algoritmo de Lankton. Cuarta parte de la función *ElevarAlCuadrado*

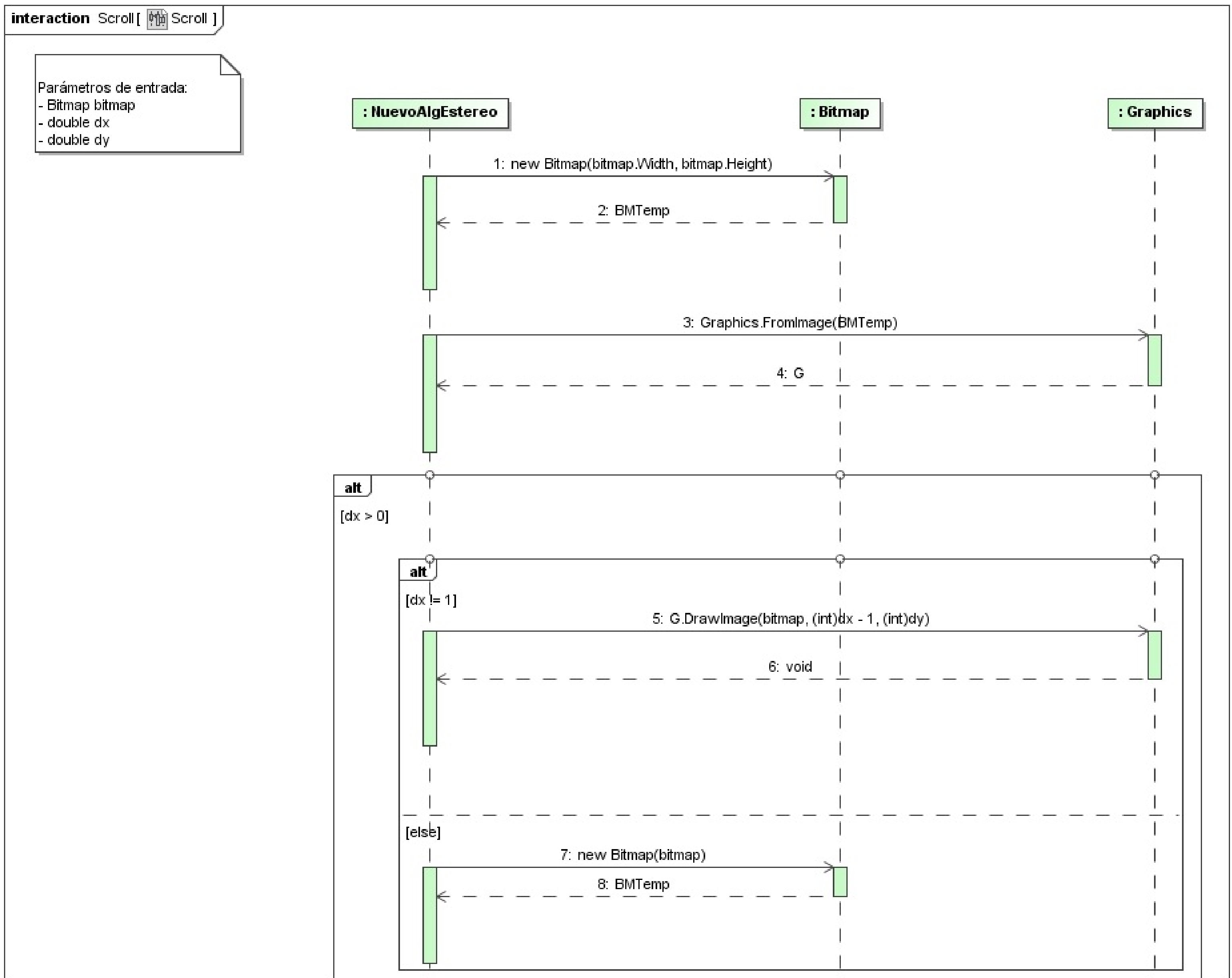

Figura A.37: Algoritmo de Lankton. Primera parte de la función *Scroll*.

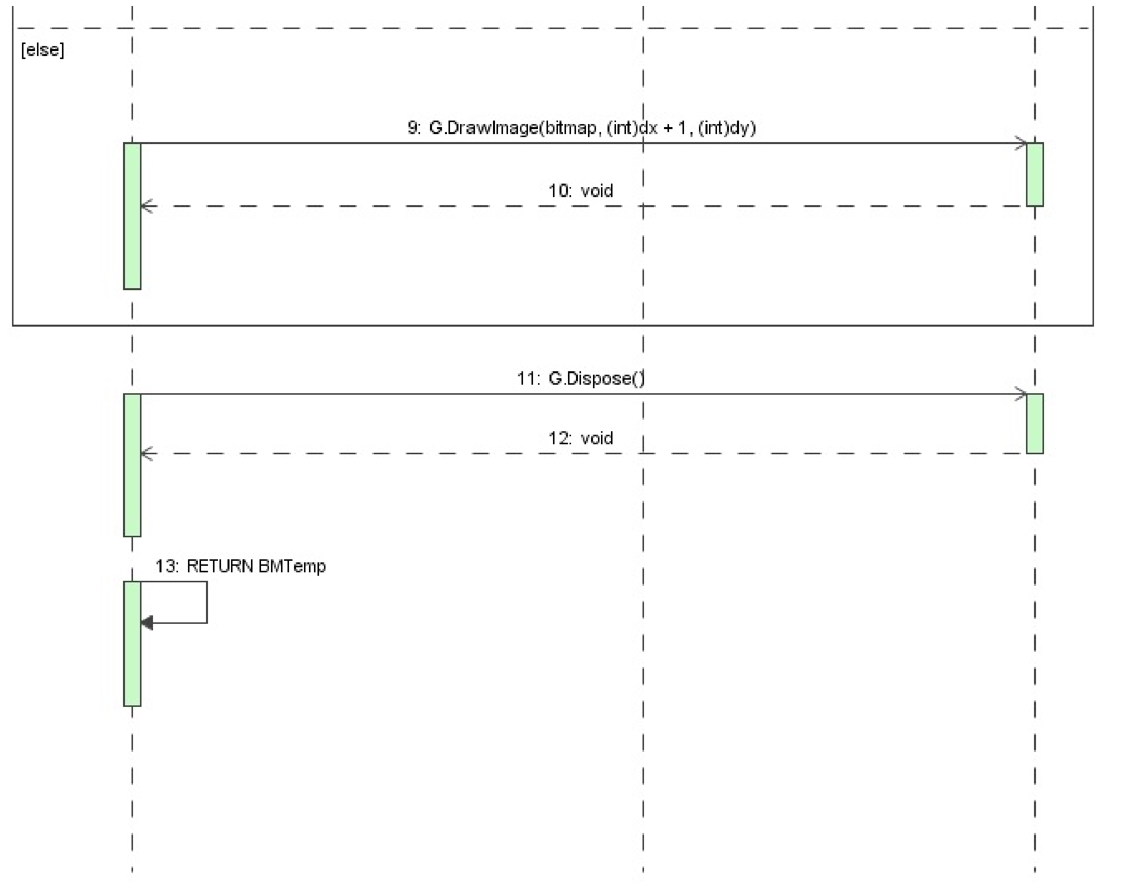

Figura A.38: Algoritmo de Lankton. Segunda parte de la función *Scroll*

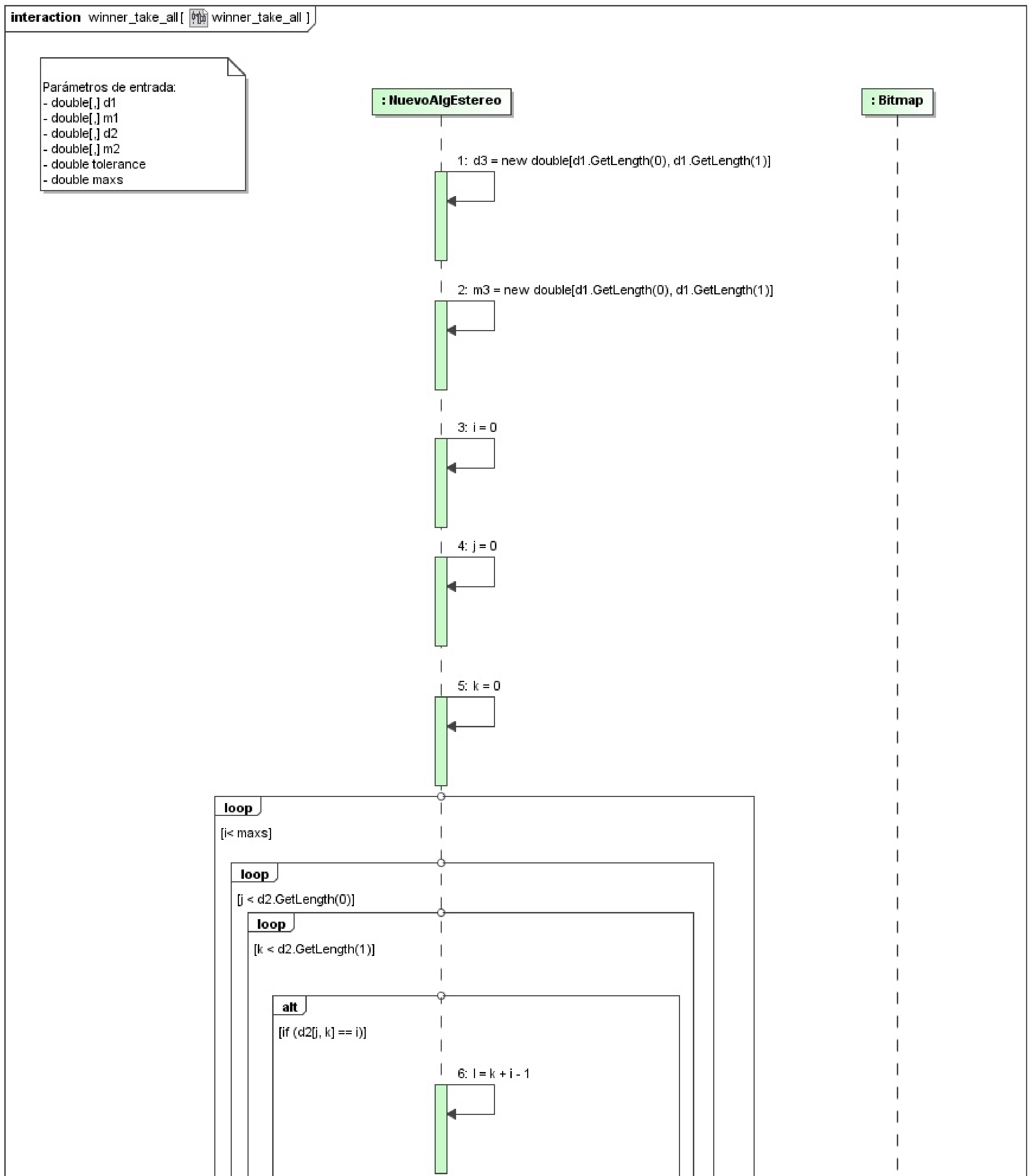

Figura A.39: Algoritmo de Lankton. Primera parte de la función *winnerTakeAll*.

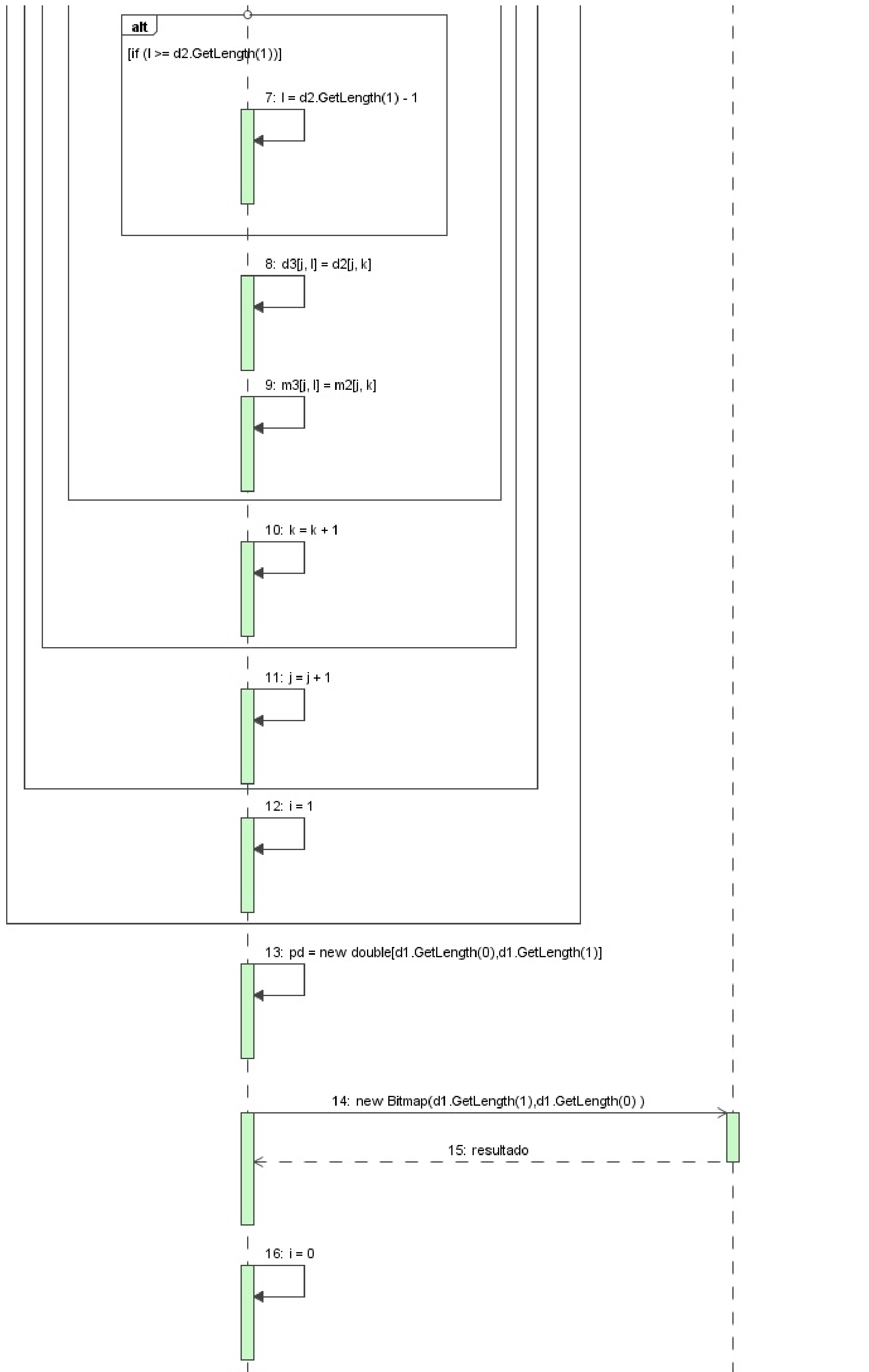

Figura A.40: Algoritmo de Lankton. Segunda parte de la función *winnerTakeAll*

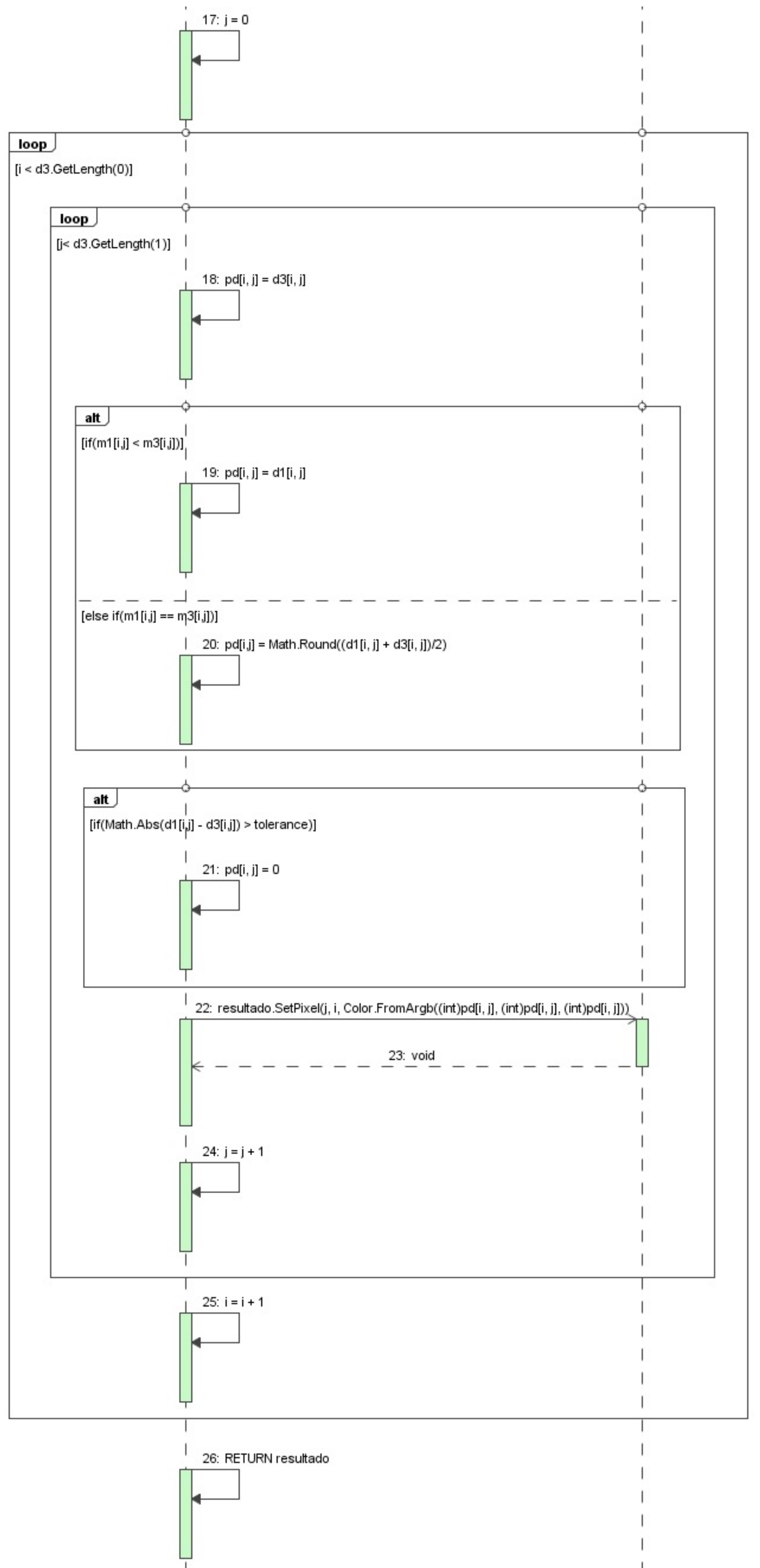

Figura A.41: Algoritmo de Lankton. Tercera parte de la función *winnerTakeAll*

A.3 Diagramas de la clase BitMapFast

A continuación, los diagramas de la clase *BitMapFast*. Tan solo son tres, y se presentan en el siguiente orden:

- *LockBitmap*:

  Sirve para bloquear un Bitmap que va a ser modificado mediante accesos a memoria.

- *PixelAt*:

  Devuelve el pixel solicitado mediante las posiciones x e y.

- *UnlockBitmap*:

  Sirve para desbloquear un Bitmap que ha sido modificado mediante accesos a memoria.

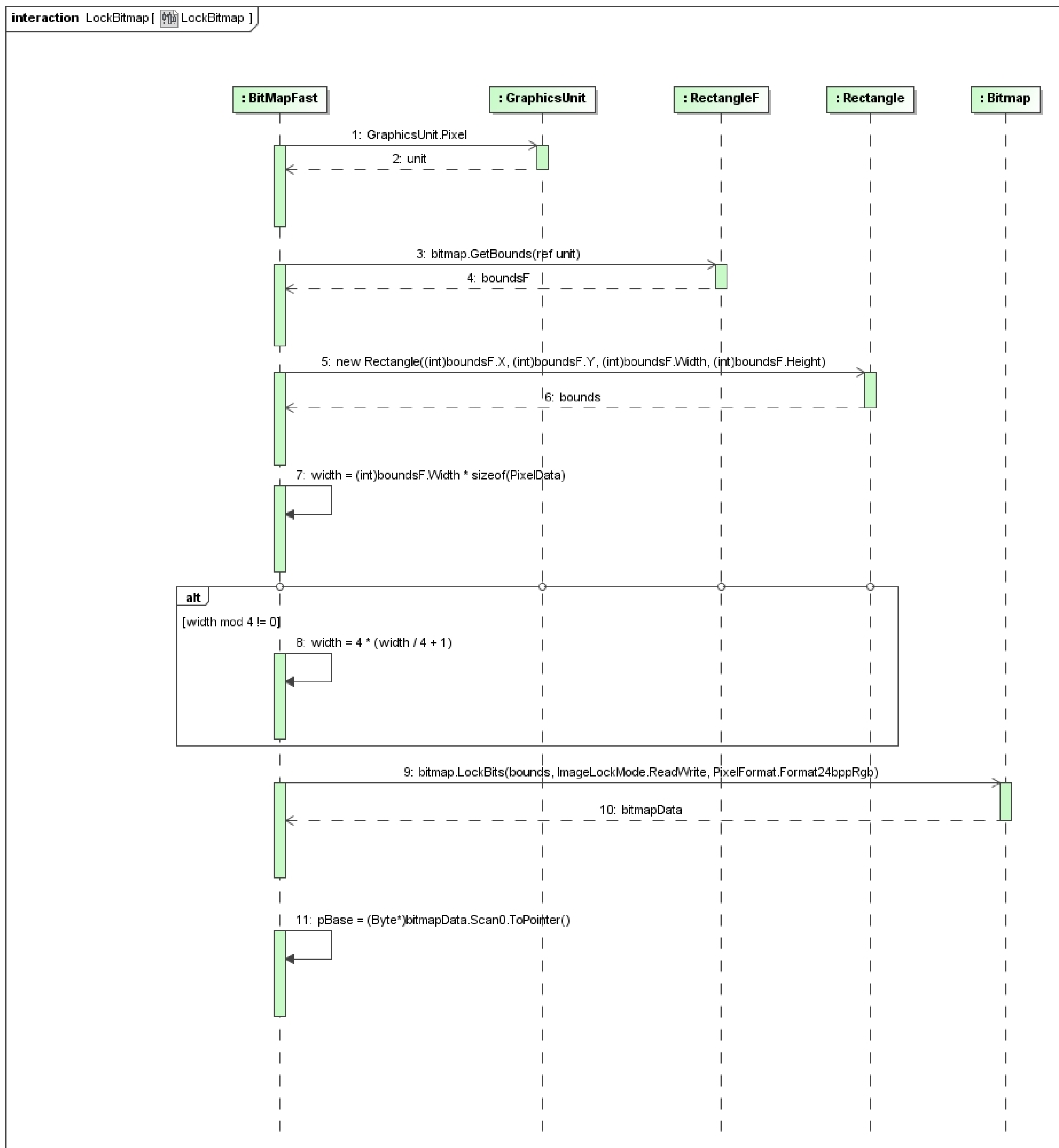

Figura A.42: Clase BitMapFast. Función LockBitmap.

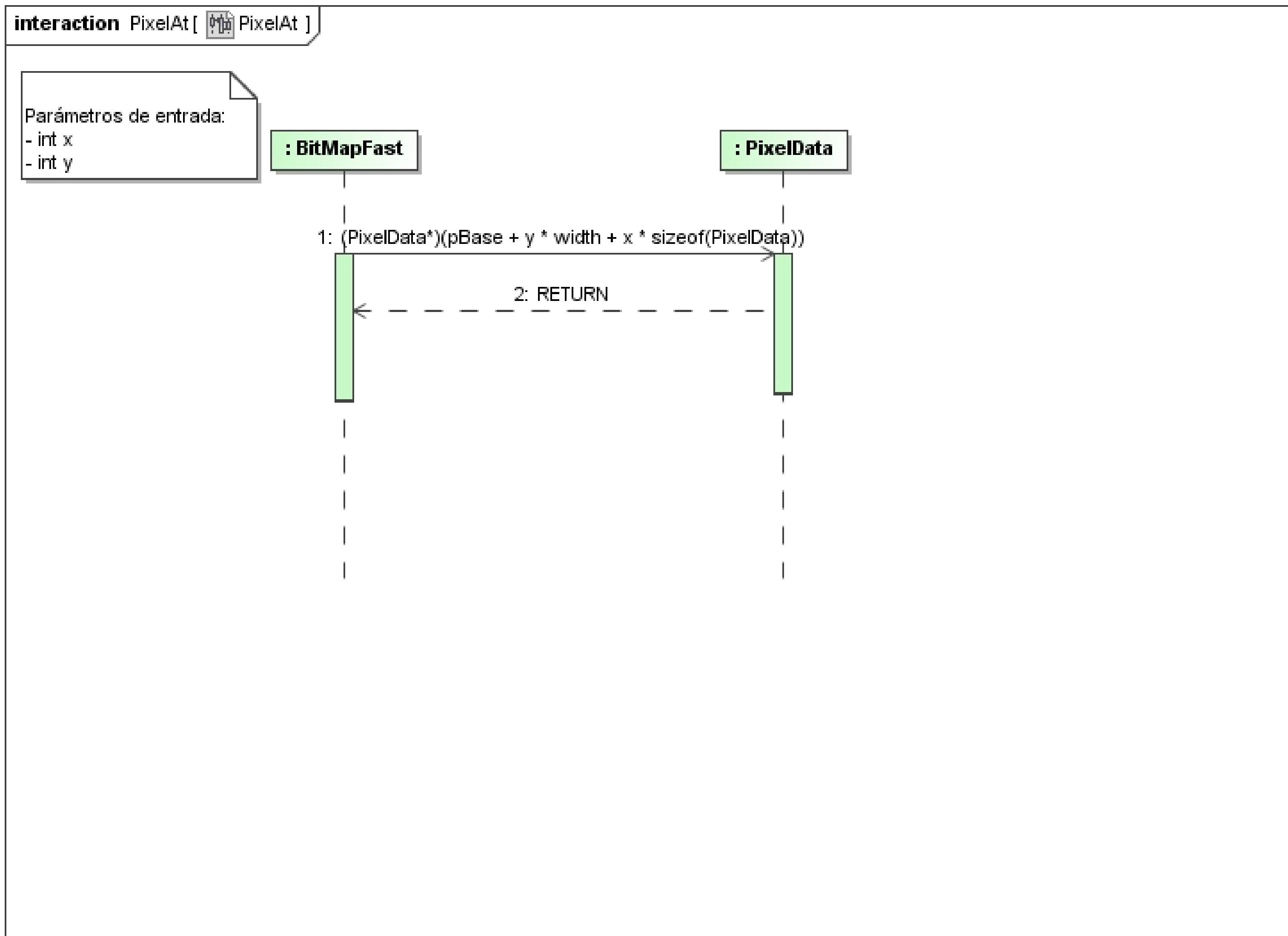

Figura A.43: Clase BitMapFast. Función PixelAt.

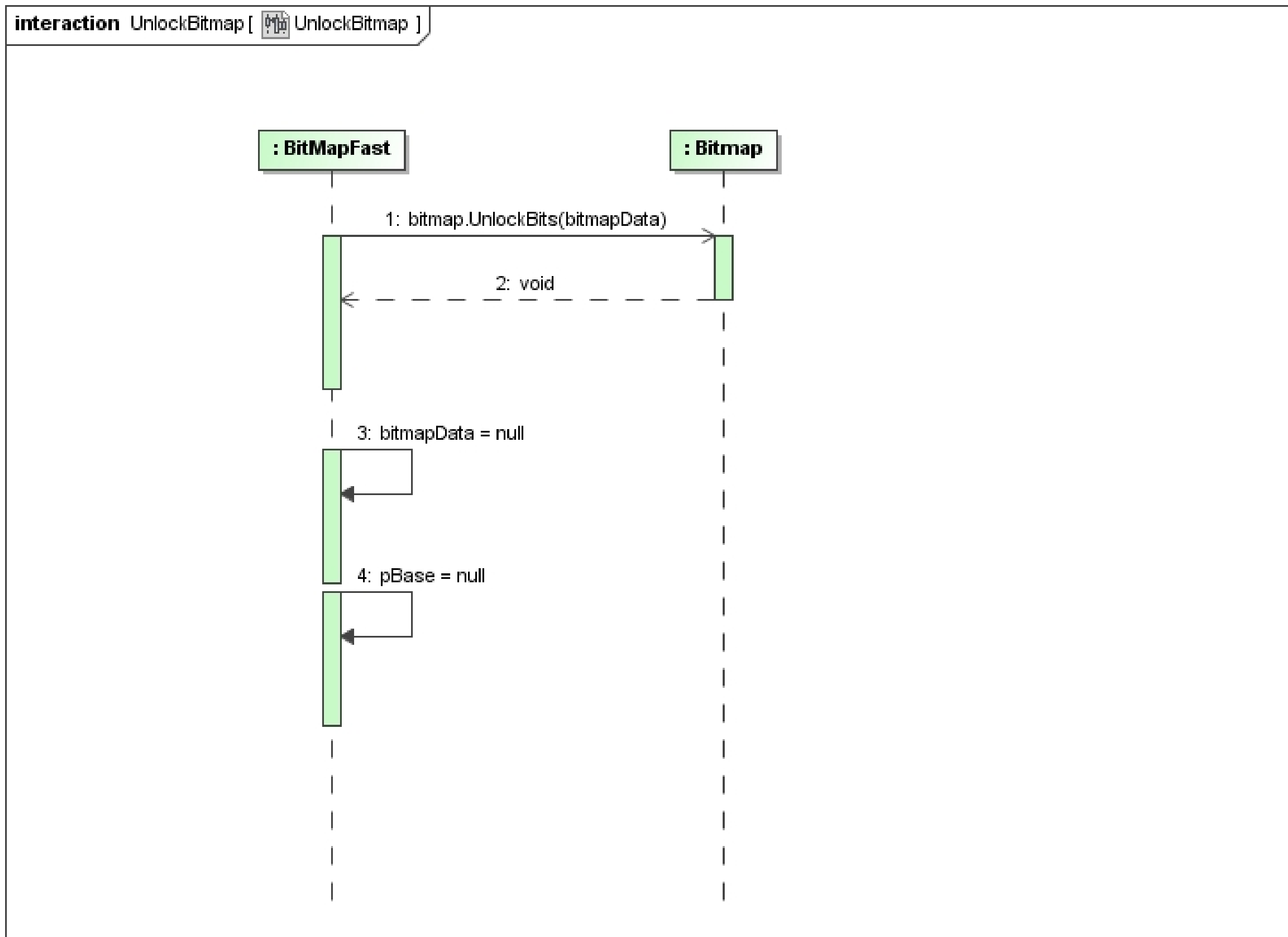

Figura A.44: Clase BitMapFast. Función *UnlockBitmap*.

A.4 Diagramas de la clase Filtros

Los diagramas pertenecientes a la clase *Filtros* son un total de diez. Se refieren a distintas operaciones que se aplican a las imagenes o a sus píxeles. Se presentan ordenados tal como vienen enumerados a continuación:

- *aumentarODisminuirLuminiscenciaMatrizHSL*:

  Como el propio nombre indica, esta función es utilizada para aumentar o disminuir la luminiscencia de la matriz con formato HSL.

- *BitmapHSL2RGB*:

  Su utilidad es convertir un Bitmap con formato HSL en uno con formato RGB modificando píxel a píxel.

- *BitmapRGB2HSL*:

  Su utilidad es convertir un Bitmap con formato RGB en uno con formato HSL modificando píxel a píxel.

- *FiltroMedia*:

  Esta función aplica el filtro de la media a un Bitmap dado.

- *FiltroMedia3*:

  El resultado que produce esta función es una matriz de valores en la que cada componente es el resultado de hallar la media de los valores entre el valor contenido en la componente correspondiente de la matriz original con los valores de su alrededor.

- *FiltroMedia3Gradiente*:

  El resultado que produce esta función es una matriz de valores en la que cada componente es el resultado de hallar la media de los valores entre el valor contenido en la componente correspondiente de la matriz original con los valores de su alrededor.

- *FiltroModa*:

  Aplica el filtro de la moda sobre un Bitmap fuente pasado por parámetro.

- *Moda*:

  Aplica la moda a una serie de valores partiendo del que viene indicado por parámetro mediante su posición en el Bitmap original con un límite máximo de disparidad.

- *MostrarDisparidad*:

  Como el propio nombre indica, esta función es la responsable de mostrar la disparidad final.

- *normalizacionIntensidad*:

  Se utiliza para corregir la luminiscencia de forma que las dos imágenes pasadas por parámetro tengan valores de la misma lo más semejantes posible.

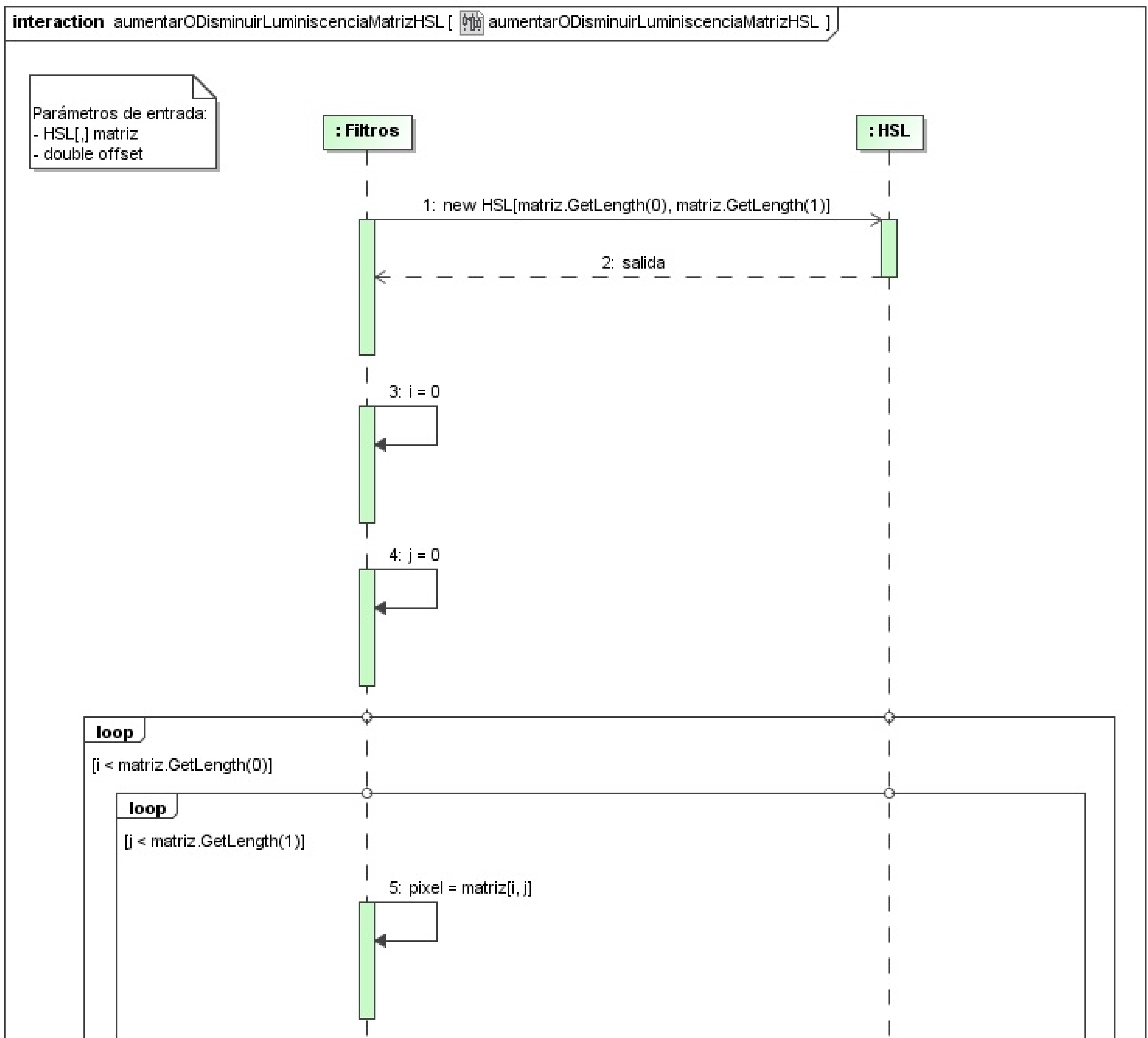

Figura A.45: Clase Filtros. Primera parte de la función *aumentarODisminuirLuminiscenciaMatrizHSL*.

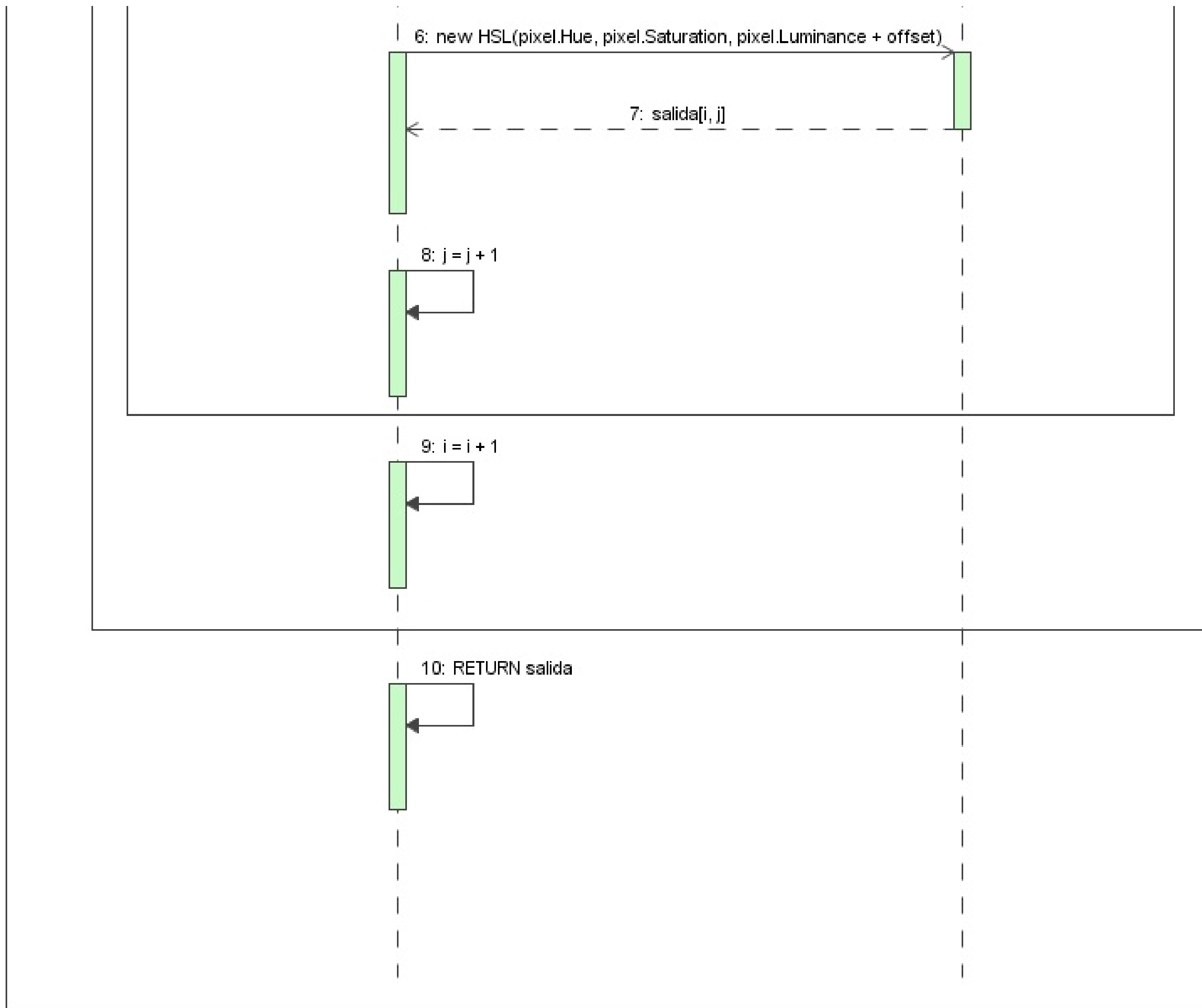

Figura A.46: Clase Filtros. Segunda parte de la función *aumentarODisminuirLuminiscenciaMatrizHSL*.

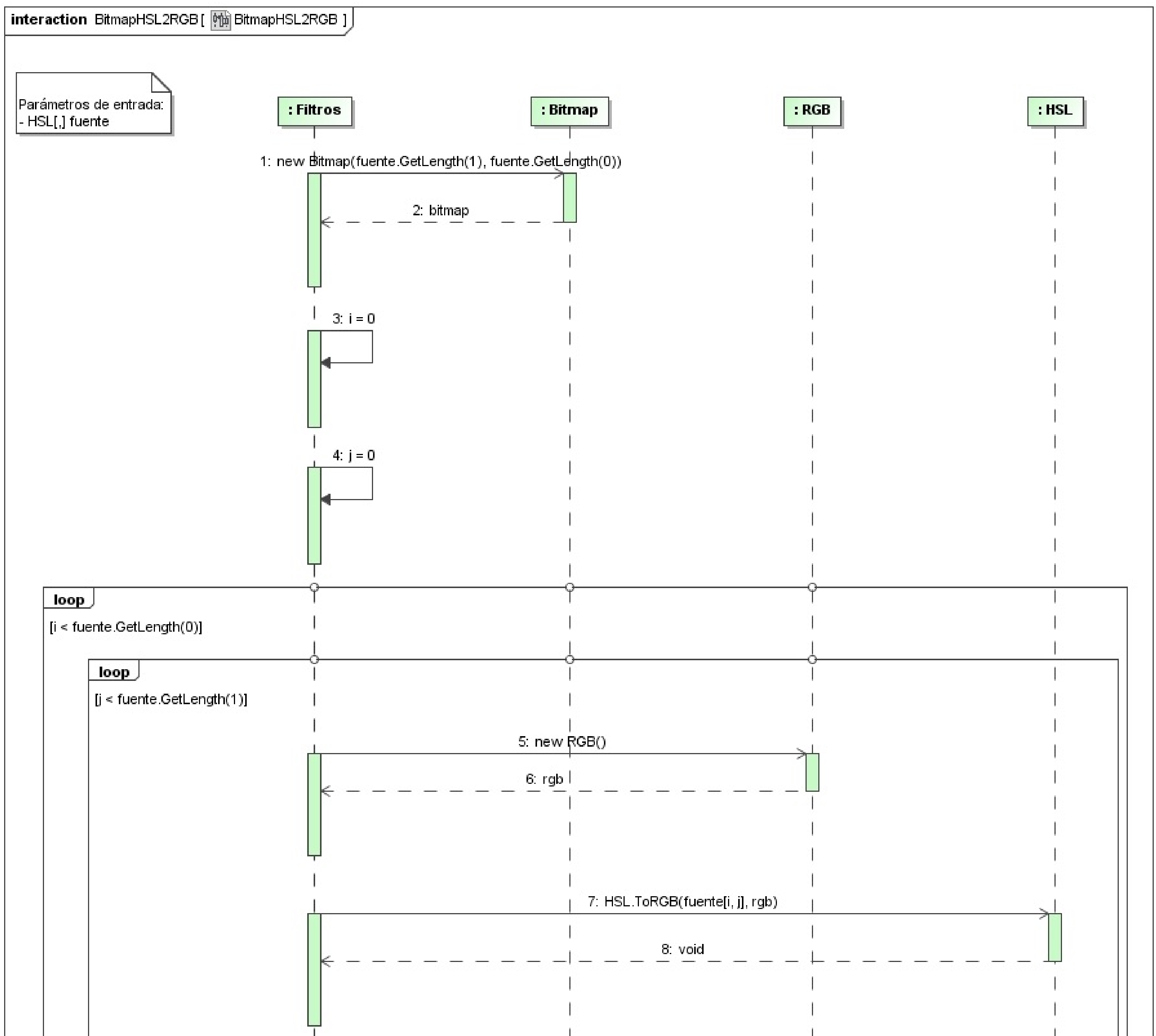

Figura A.47: Clase Filtros. Primera parte de la función *BitmapHSL2RGB*.

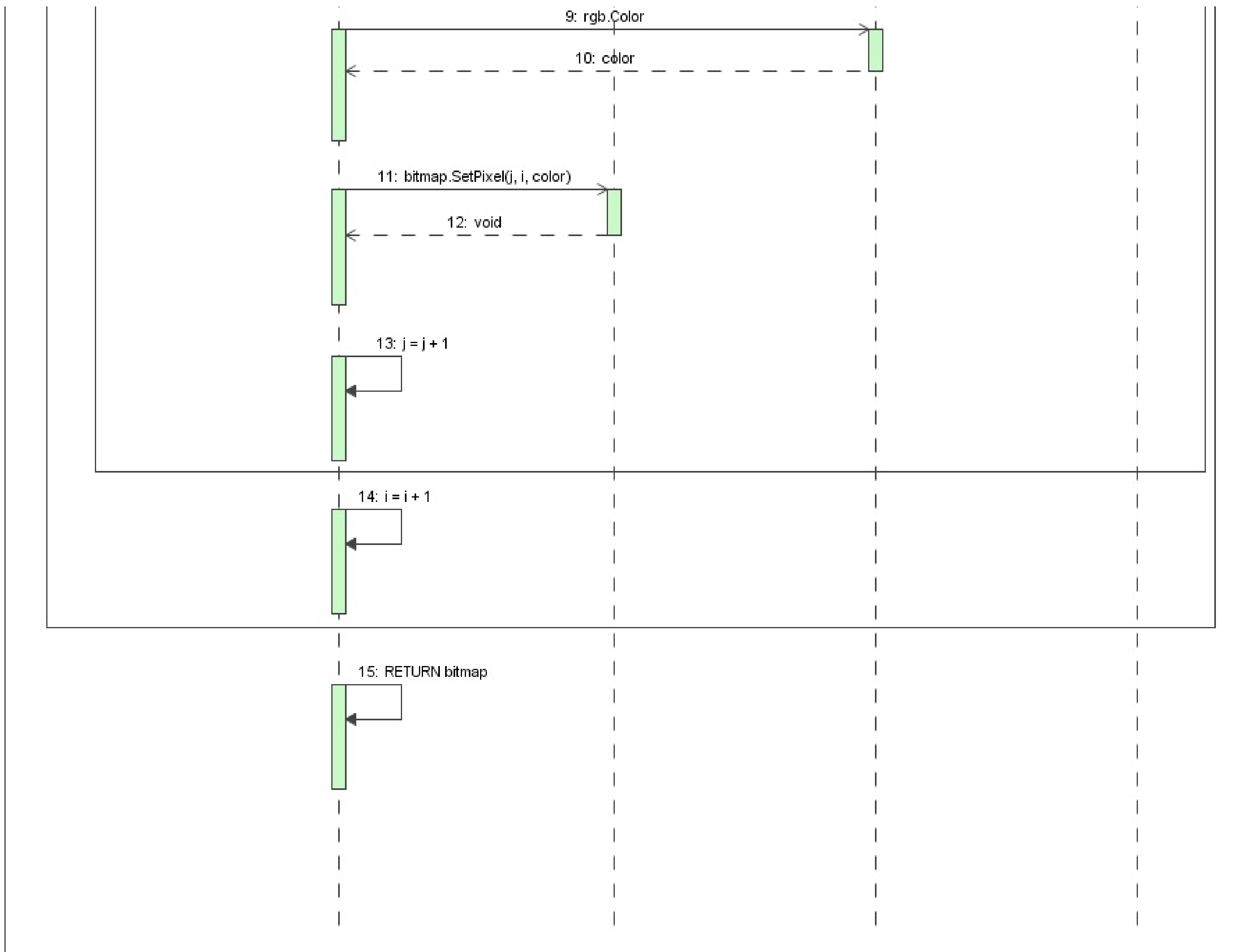

Figura A.48: Clase Filtros. Segunda parte de la función *BitmapHSL2RGB*.

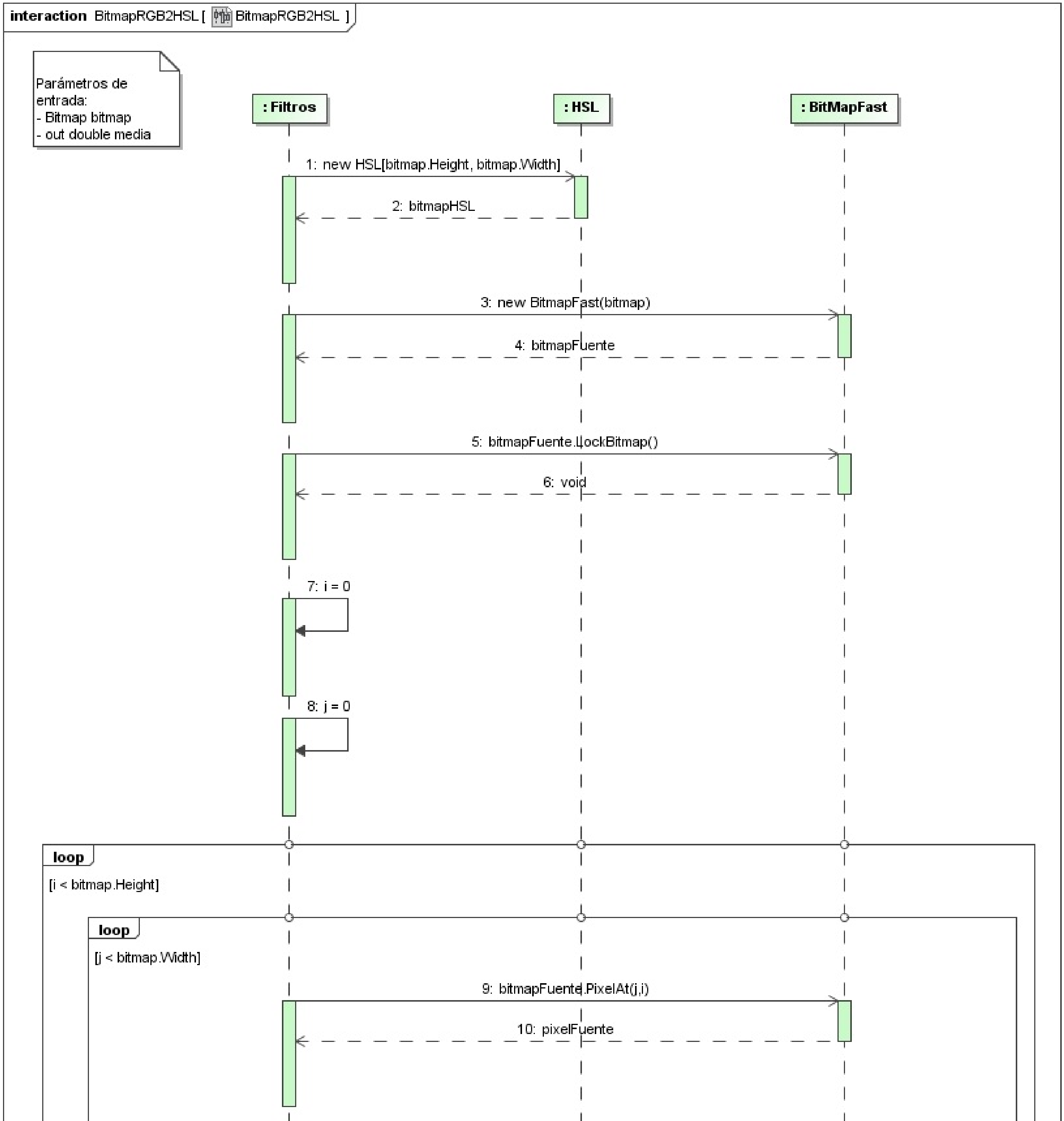

Figura A.49: Clase Filtros. Primera parte de la función *BitmapRGB2HSL*.

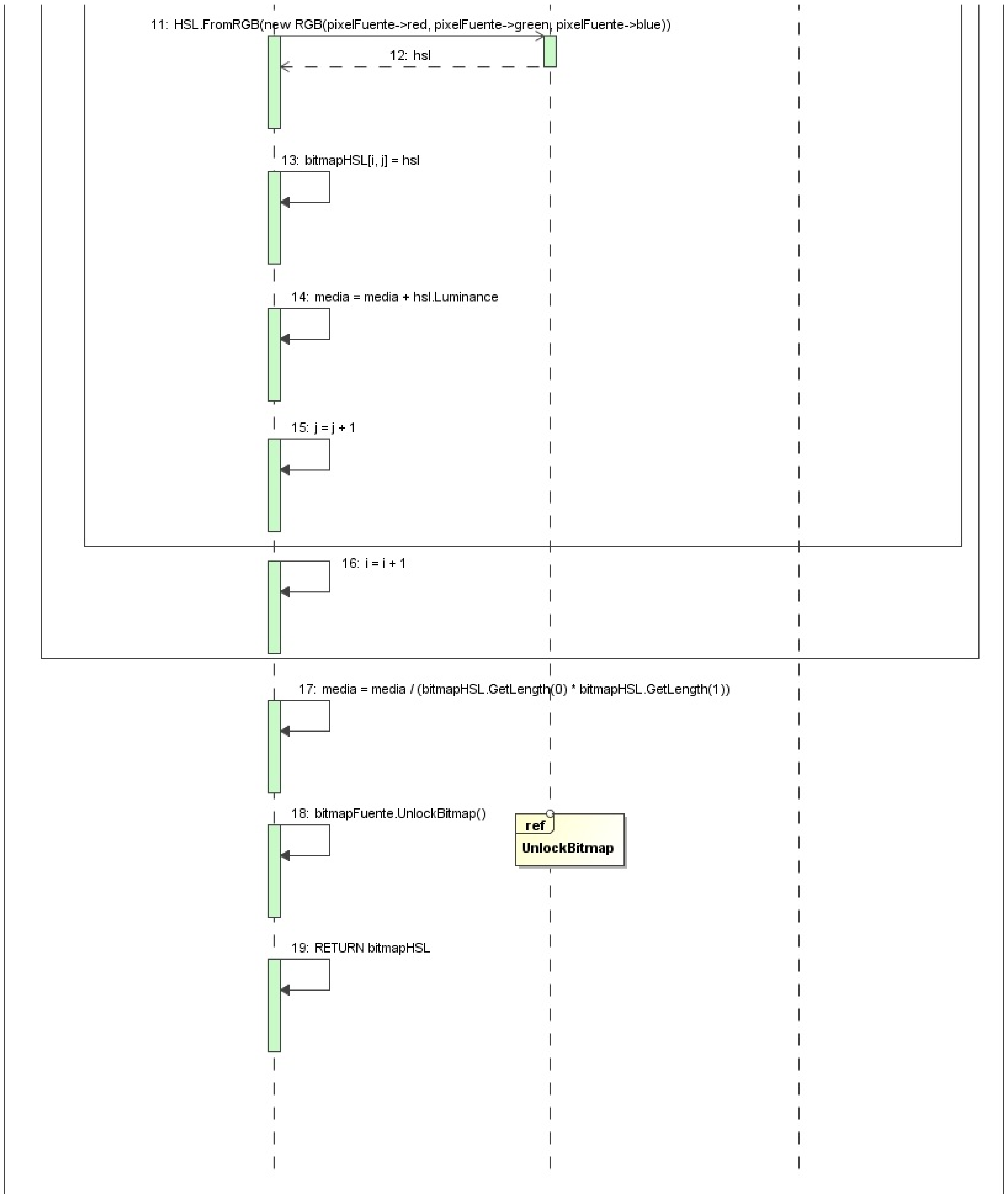

Figura A.50: Clase Filtros. Segunda parte de la función *BitmapRGB2HSL*.

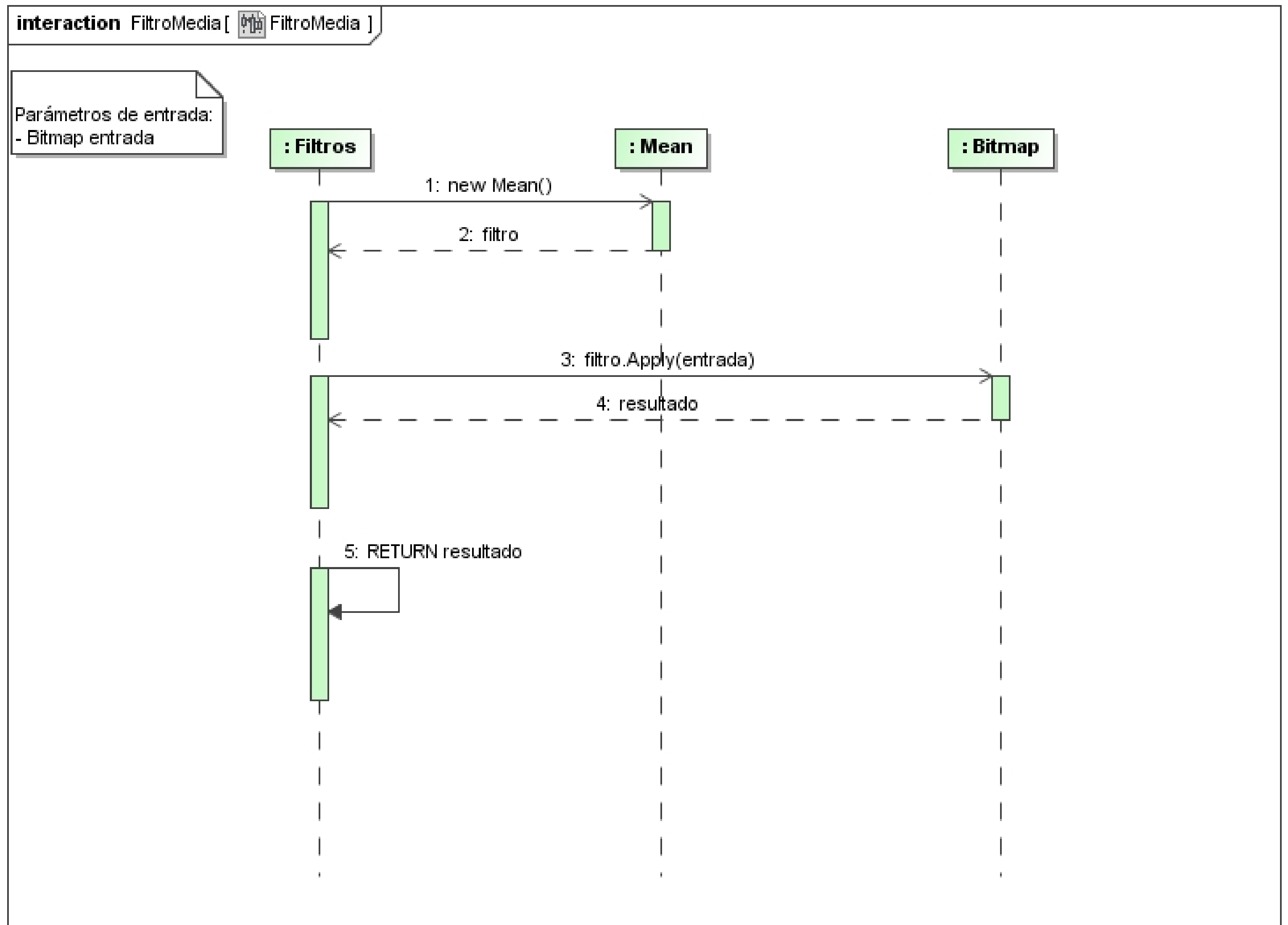

Figura A.51: Clase Filtros. Función *FiltroMedia*.

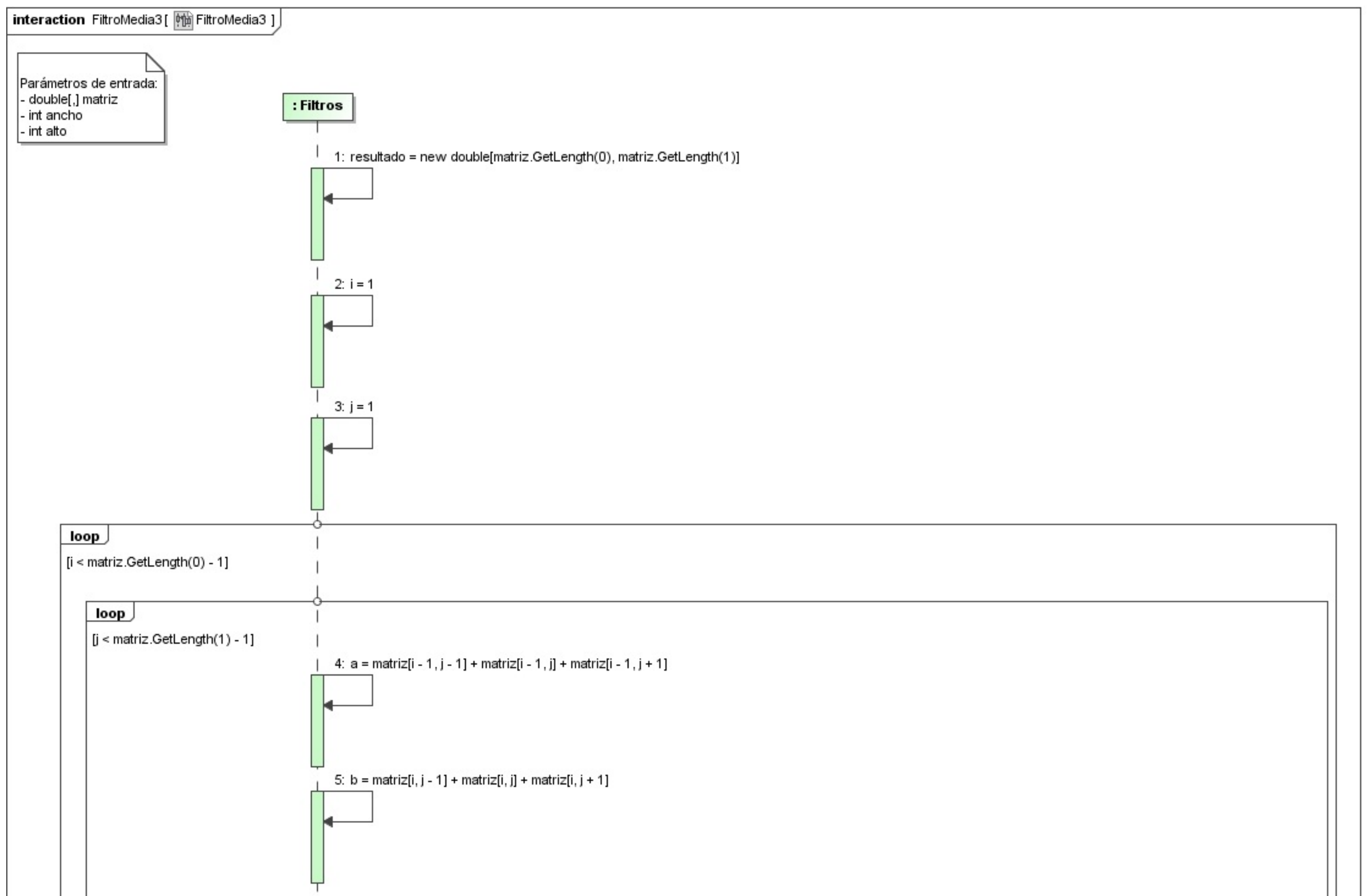

Figura A.52: Clase Filtros. Primera parte de la función *FiltroMedia3*.

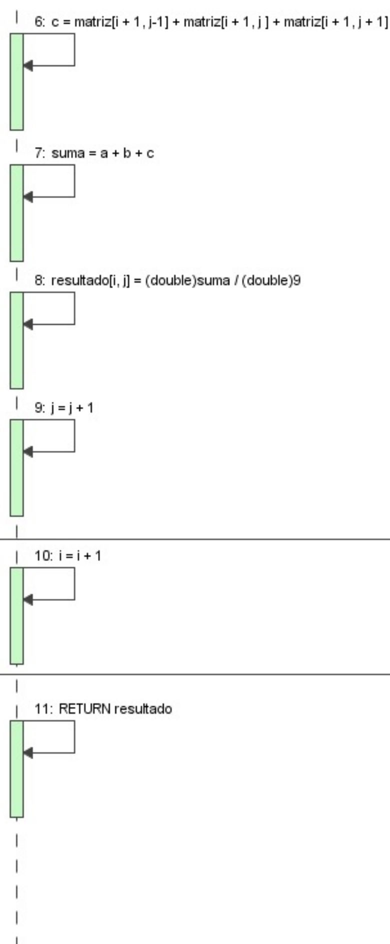

Figura A.53: Clase Filtros. Segunda parte de la función *FiltroMedia3*.

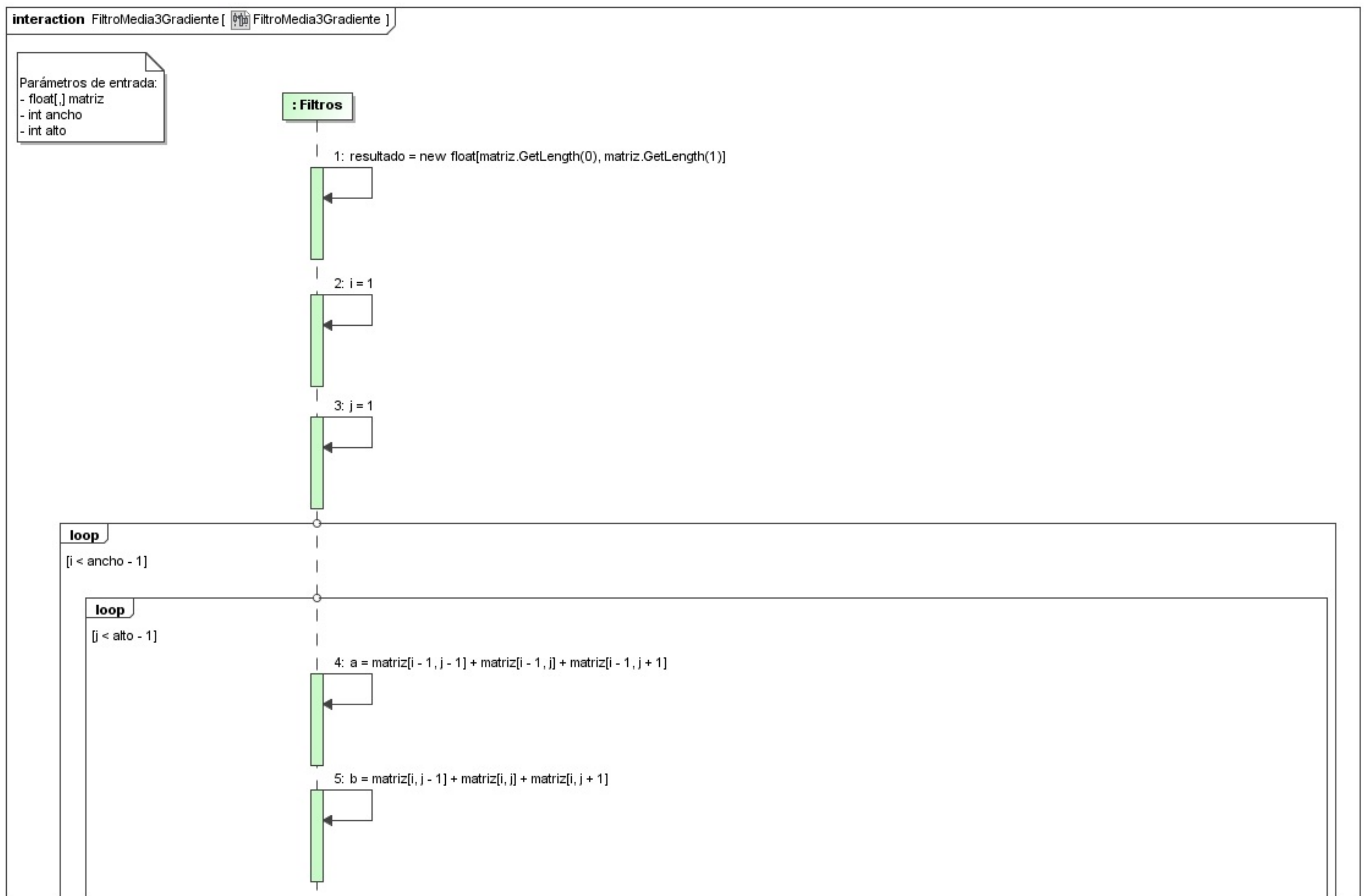

Figura A.54: Clase Filtros. Primera parte de la función *FiltroMedia3Gradiente*.

Figura A.55: Clase Filtros. Segunda parte de la función *FiltroMedia3Gradiente*.

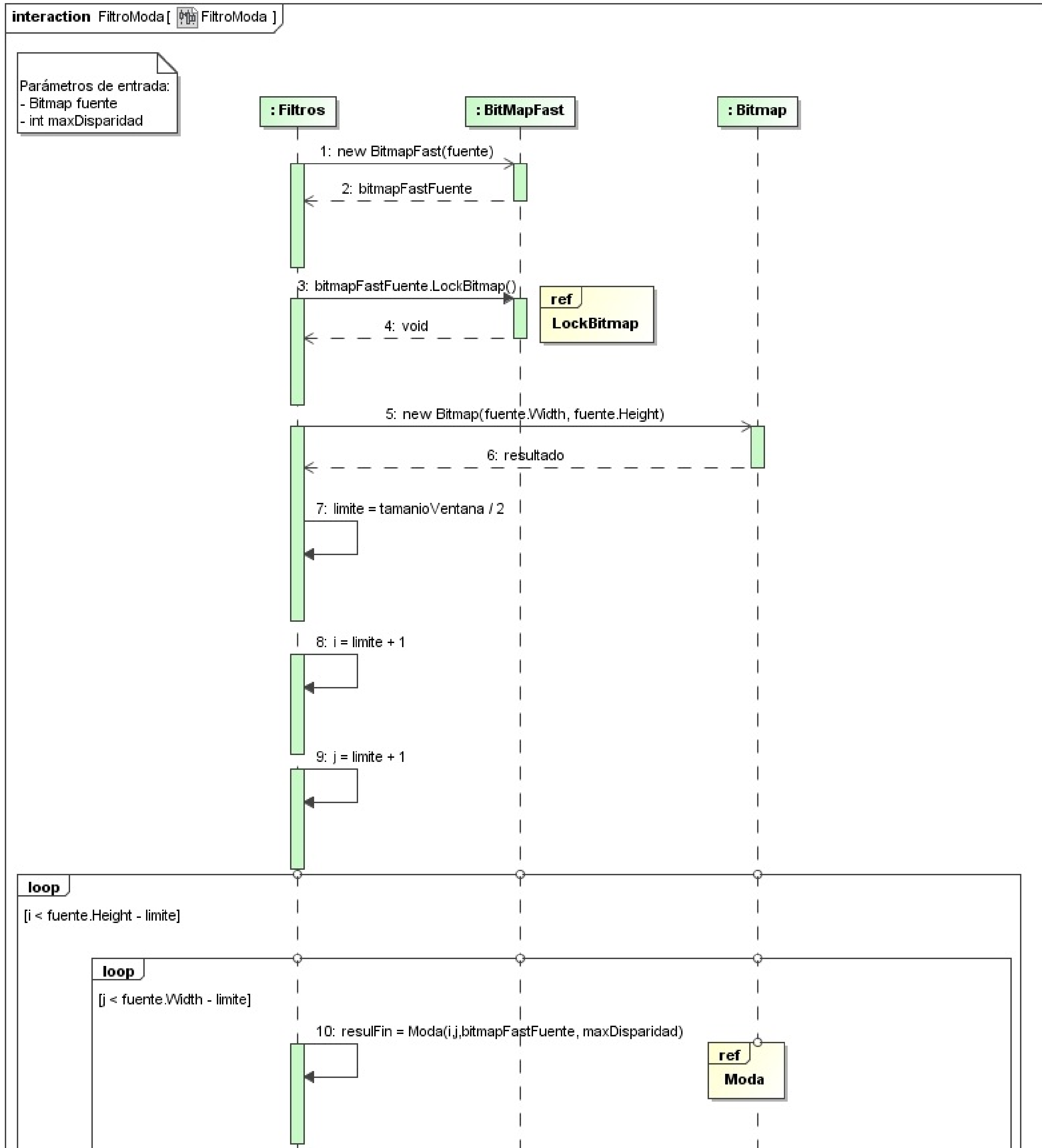

Figura A.56: Clase Filtros. Primera parte de la función *FiltroModa*.

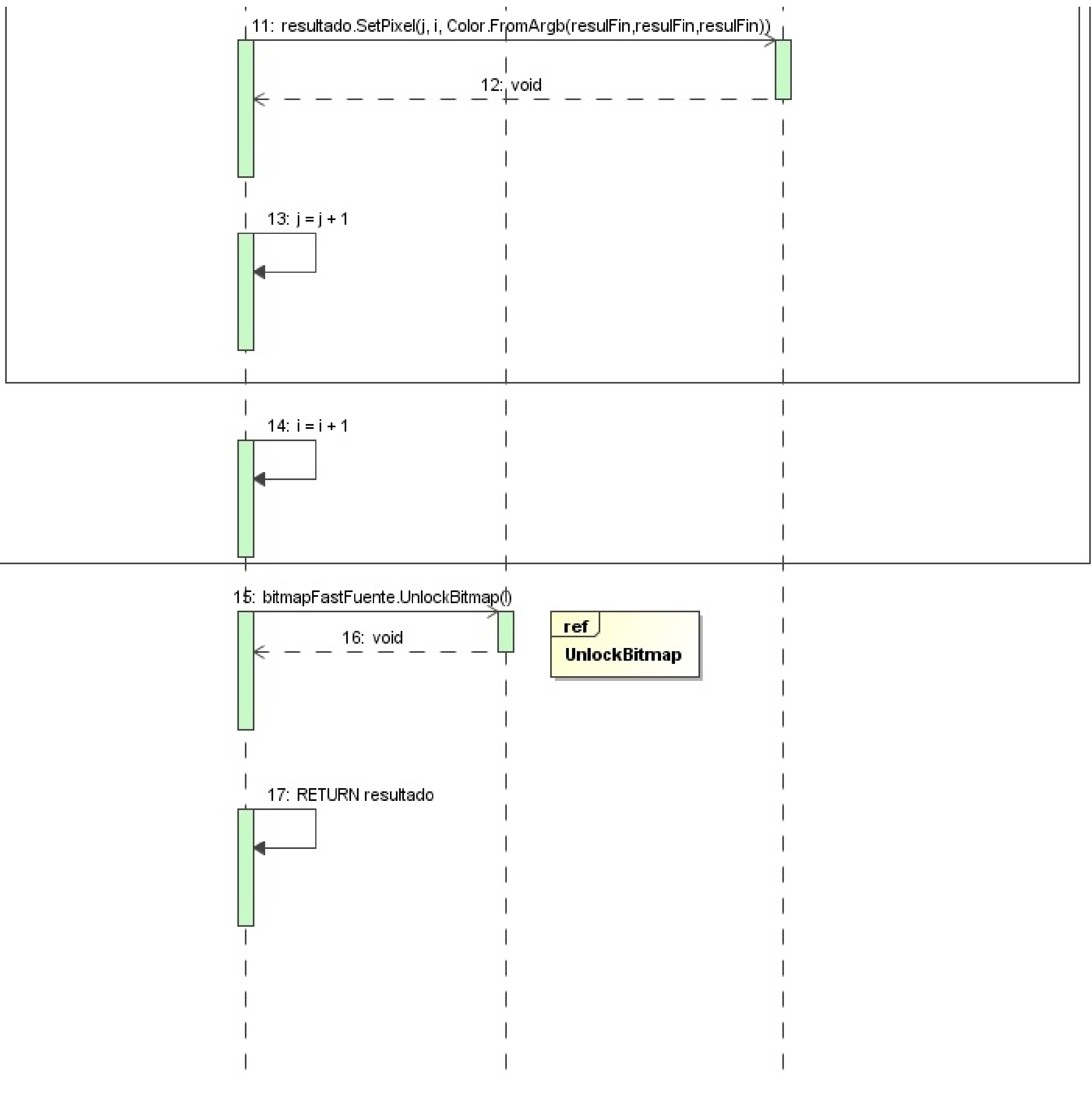

Figura A.57: Clase Filtros. Segunda parte de la función *FiltroModa*.

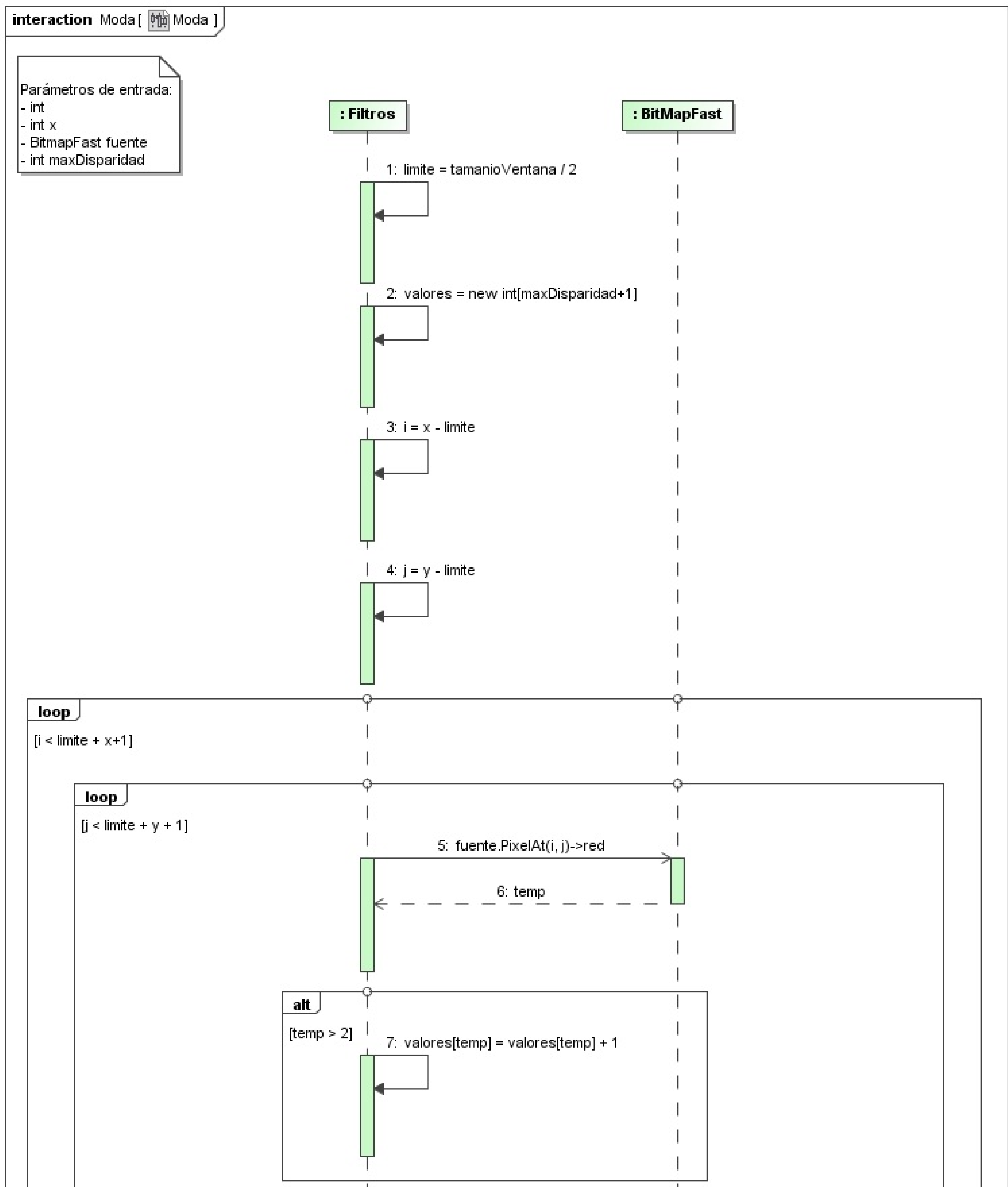

Figura A.58: Clase Filtros. Primera parte de la función *Moda*.

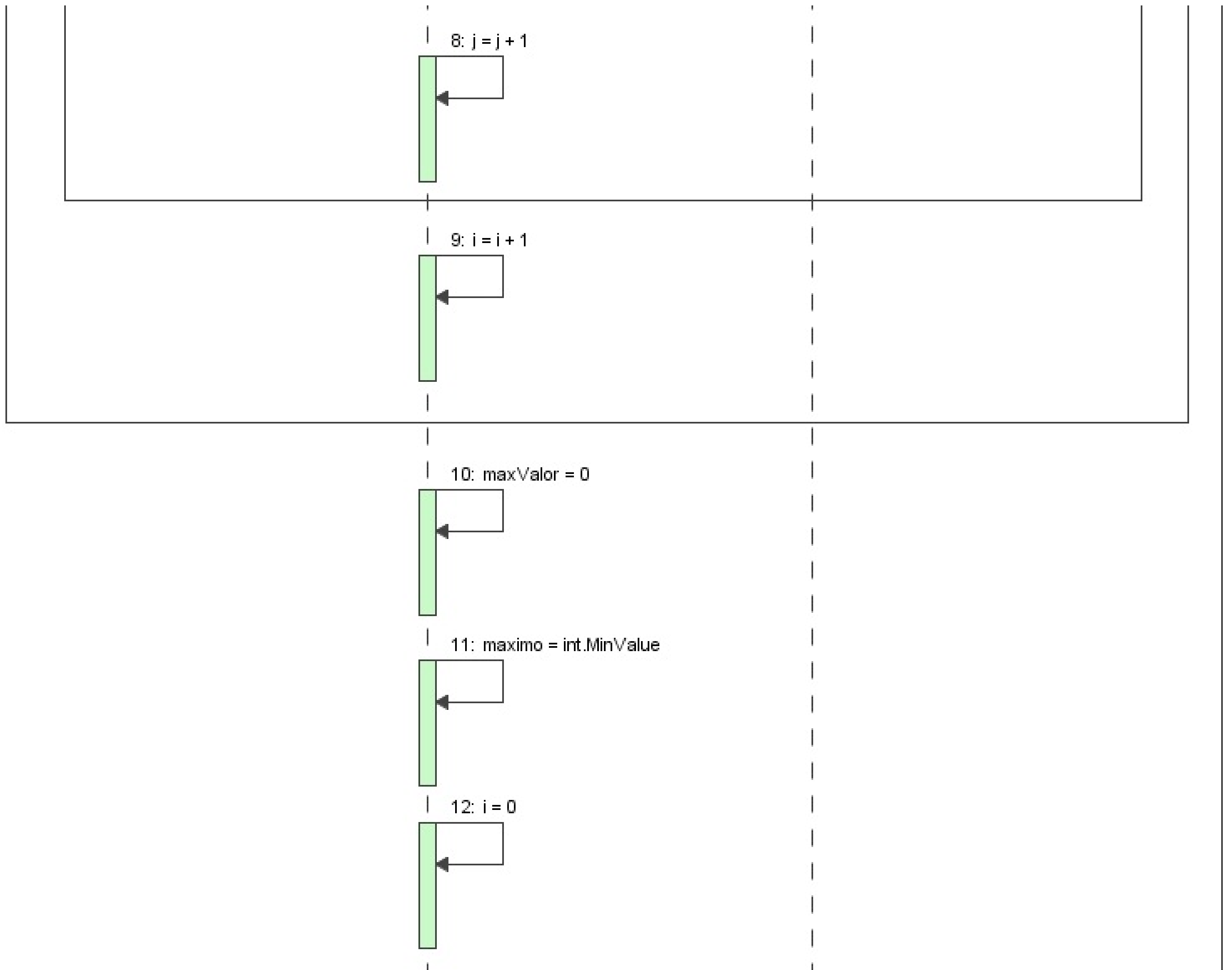

Figura A.59: Clase Filtros. Segunda parte de la función *Moda*.

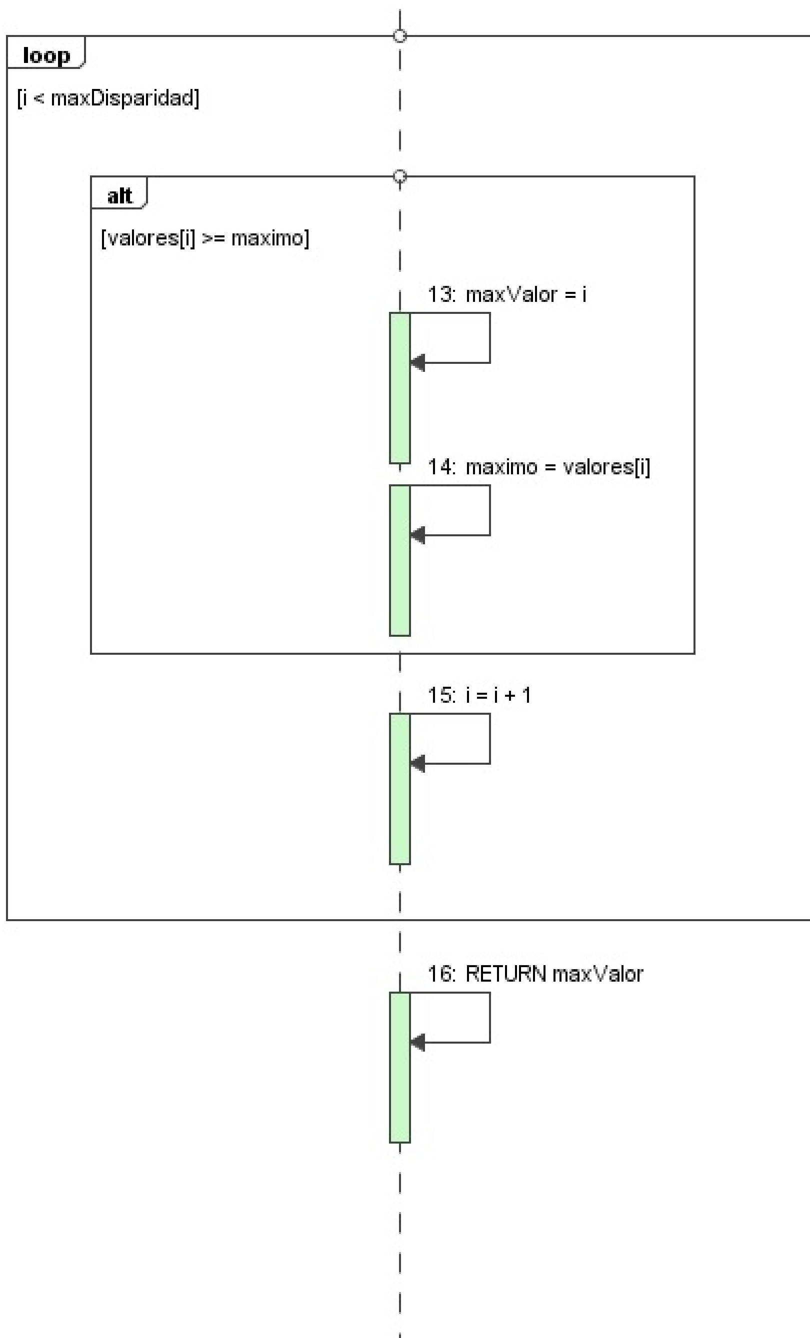

Figura A.60: Clase Filtros. Tercera parte de la función *Moda*.

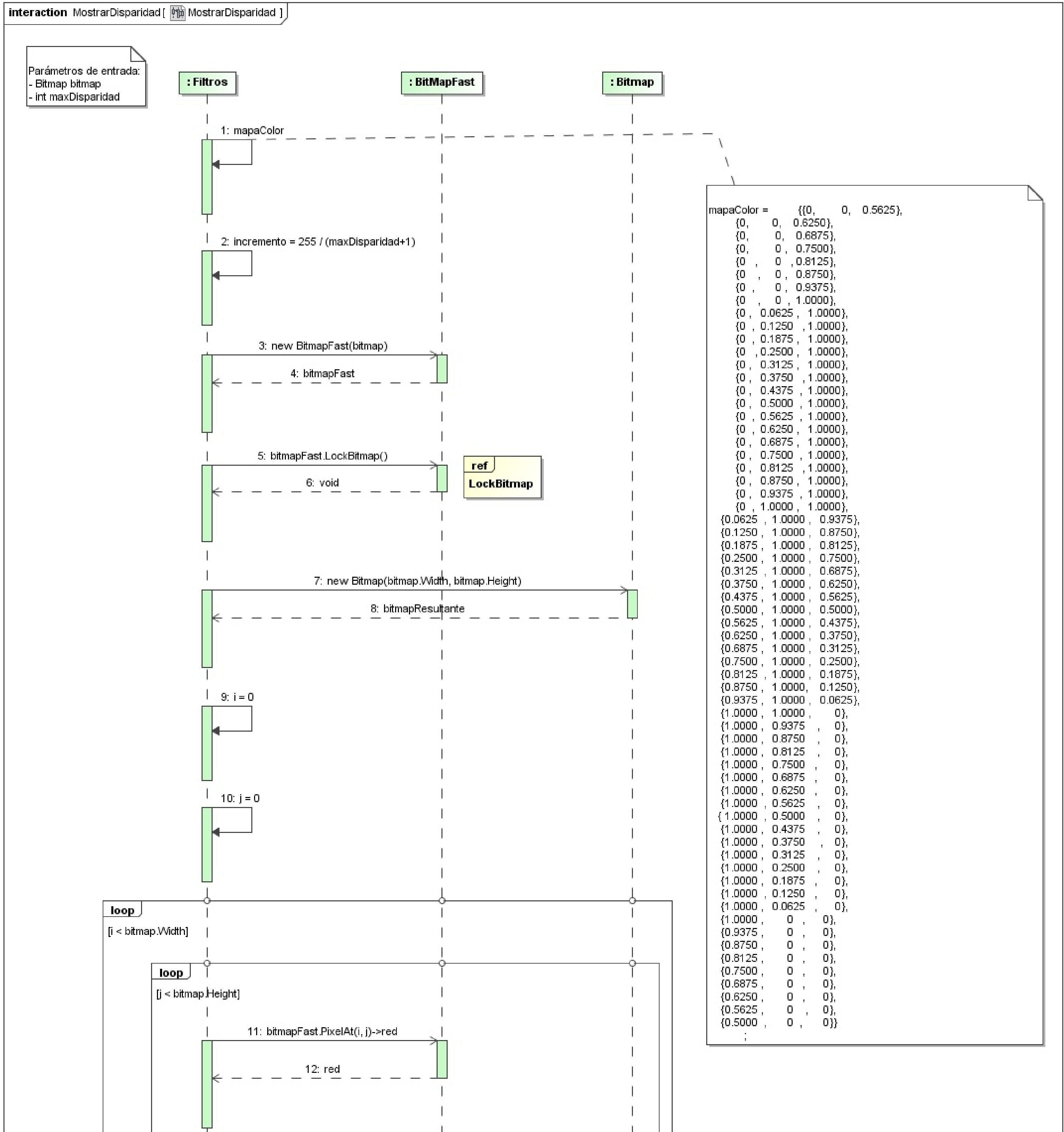

Figura A.61: Clase Filtros. Primera parte de la función *MostrarDisparidad*.

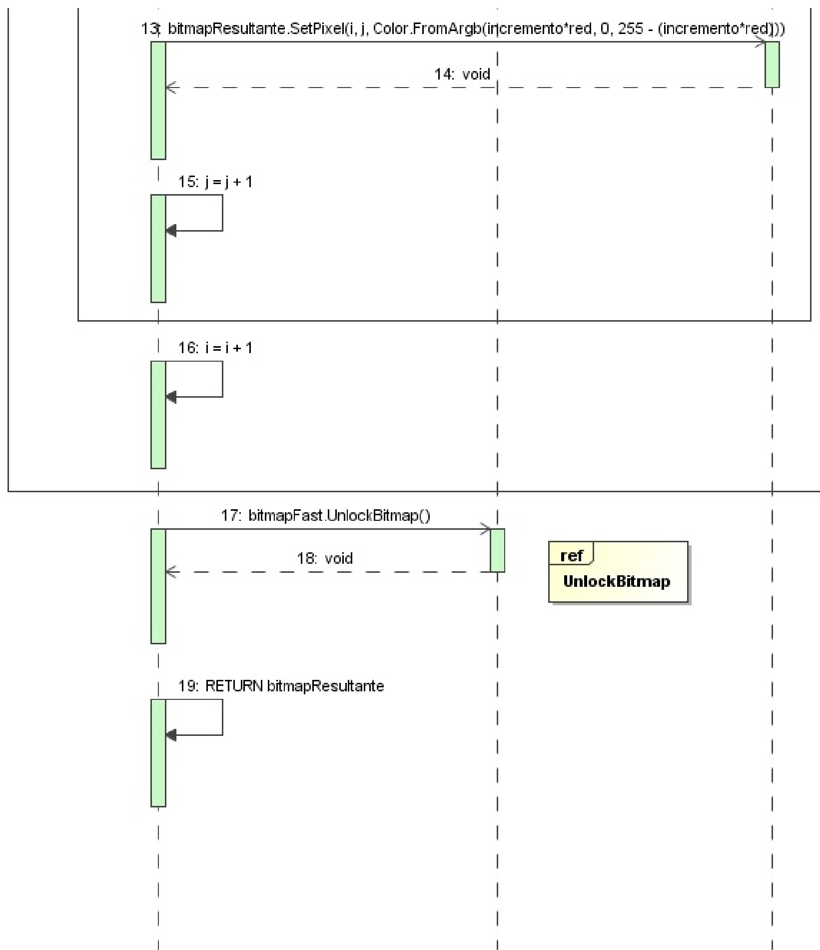

Figura A.62: Clase Filtros. Segunda parte de la función *MostrarDisparidad*.

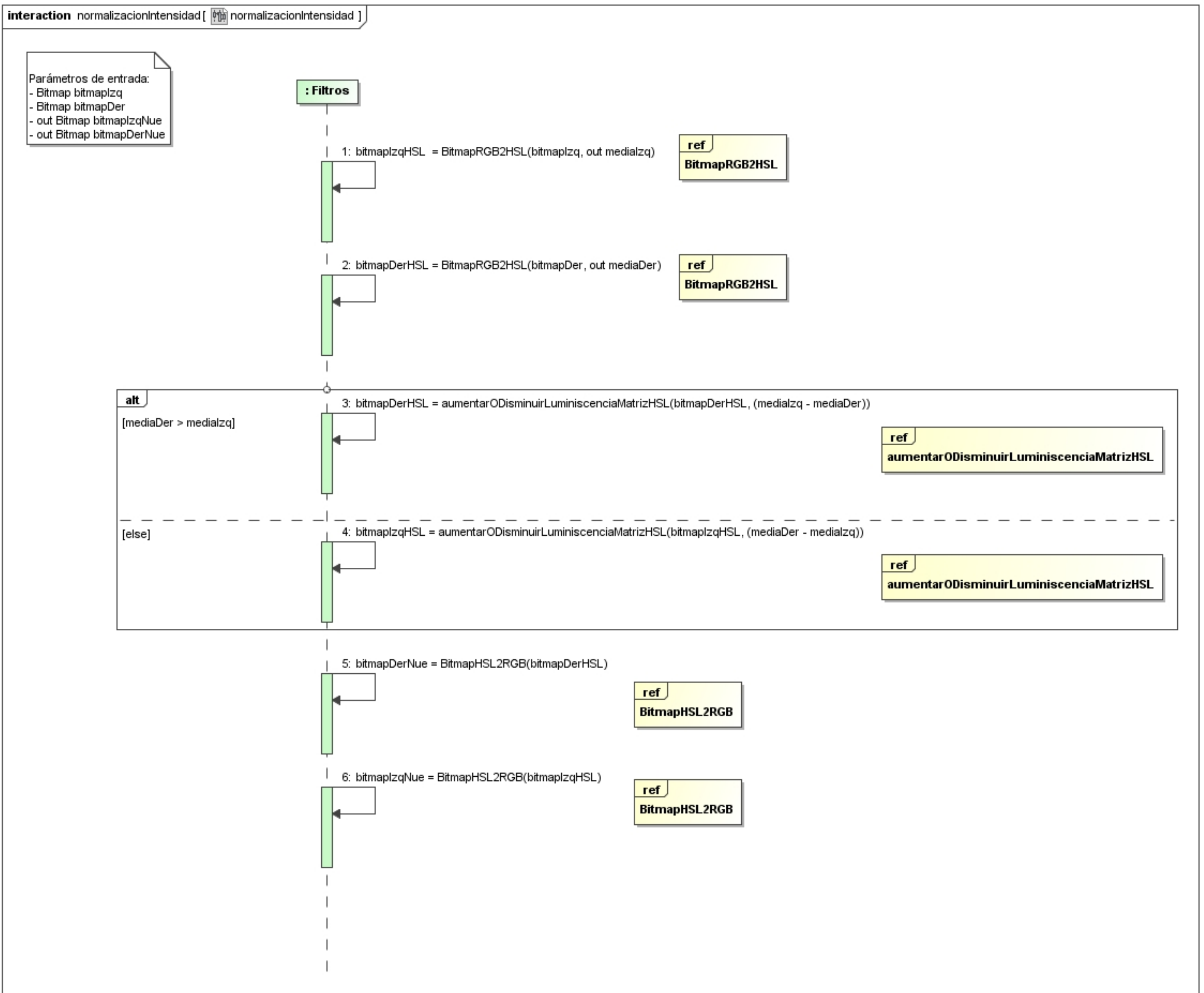

Figura A.63: Clase Filtros. Función *normalizacionIntensidad*.

A.5 Diagramas de la clase Gradiente

Los diagramas de la clase *Gradiente* son bastante similares entre sí, por lo que solo mostraremos tres diagramas, los cuales permiten deducir fácilmente cómo serán los demás:

- *IMFilter*:

  Aplica el filtro de la convolución a la imagen que le pasemos como parámetro de entrada.

- *gradienteXR*:

  Halla el gradiente de la imagen según la componente X y sobre la componente R de los píxeles. Es análogo para las componentes G y B.

- *gradienteYR*:

  Halla el gradiente de la imagen según la componente Y y sobre la componente R de los píxeles. Es análogo para las componentes G y B.

Los diagramas gradienteXG y gradienteXB son similares a gradienteXR pero usando las componentes verde y azul de cada pixel respectivamente. Los diagramas gradienteYG y gradienteYB son similares a gradienteYR pero usando las componentes verde y azul de cada pixel respectivamente.

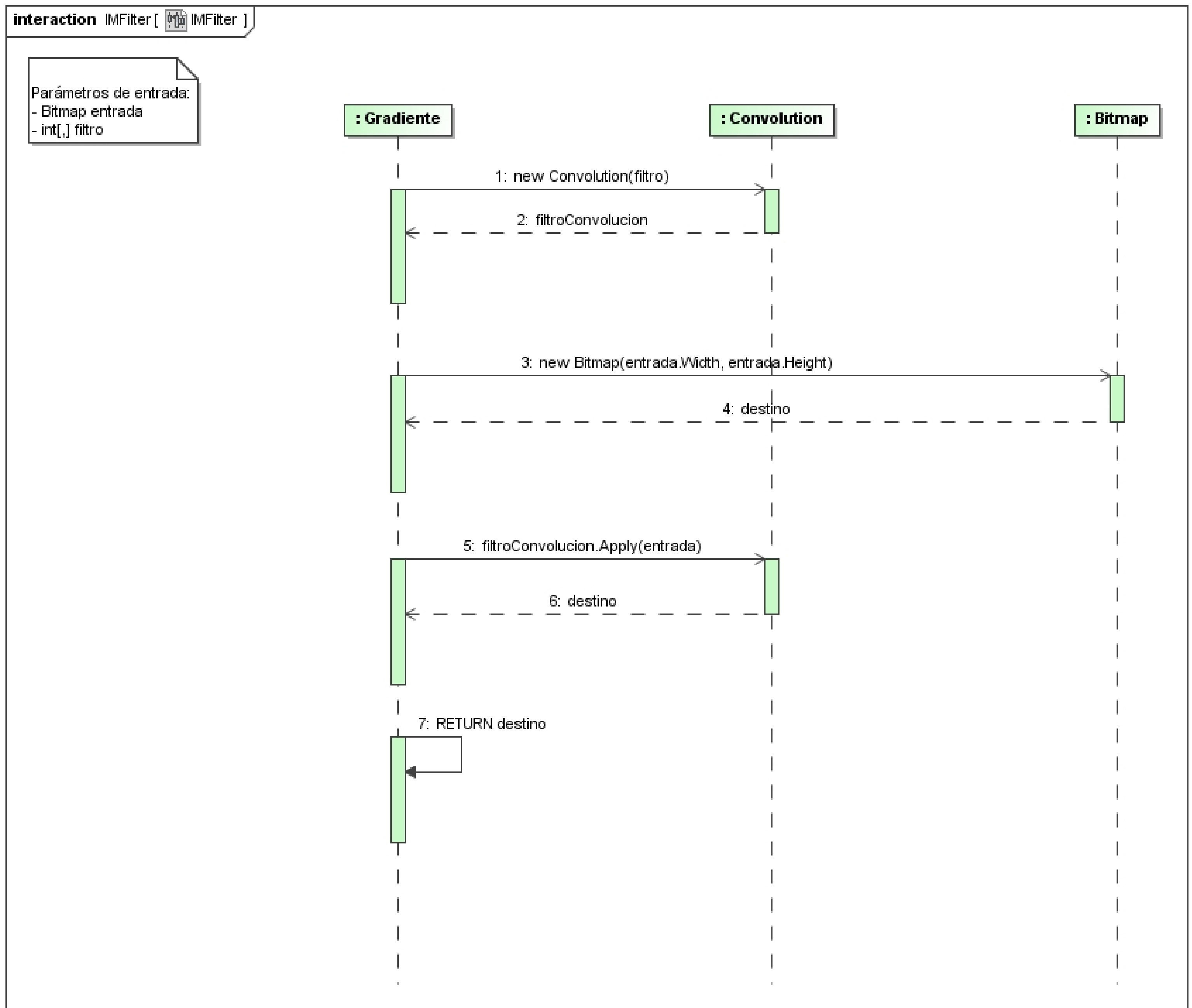

Figura A.64: Clase Gradiente. Función *IMFilter*.

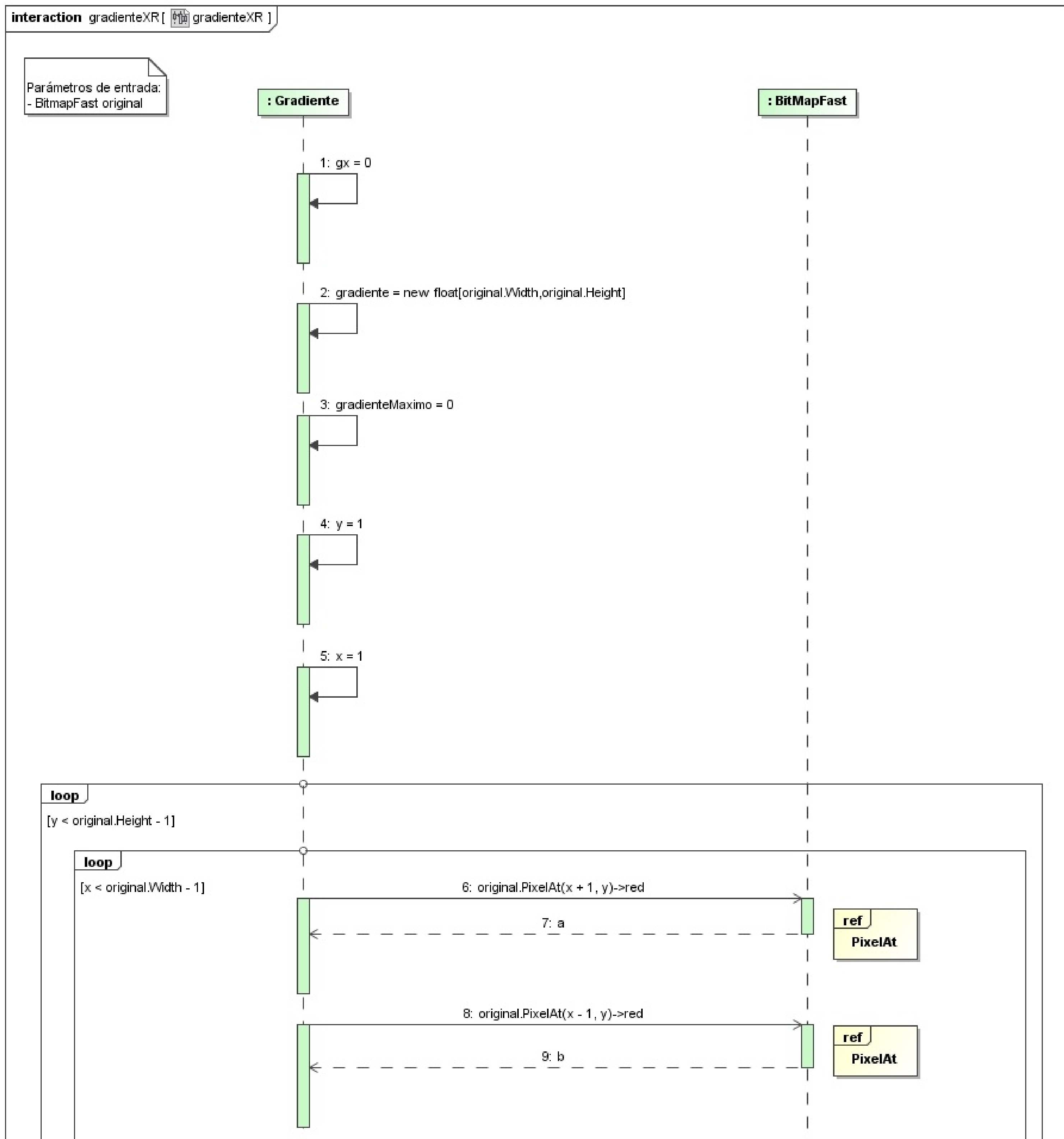

Figura A.65: Clase Gradiente. Primera parte de la función *gradienteXR*.

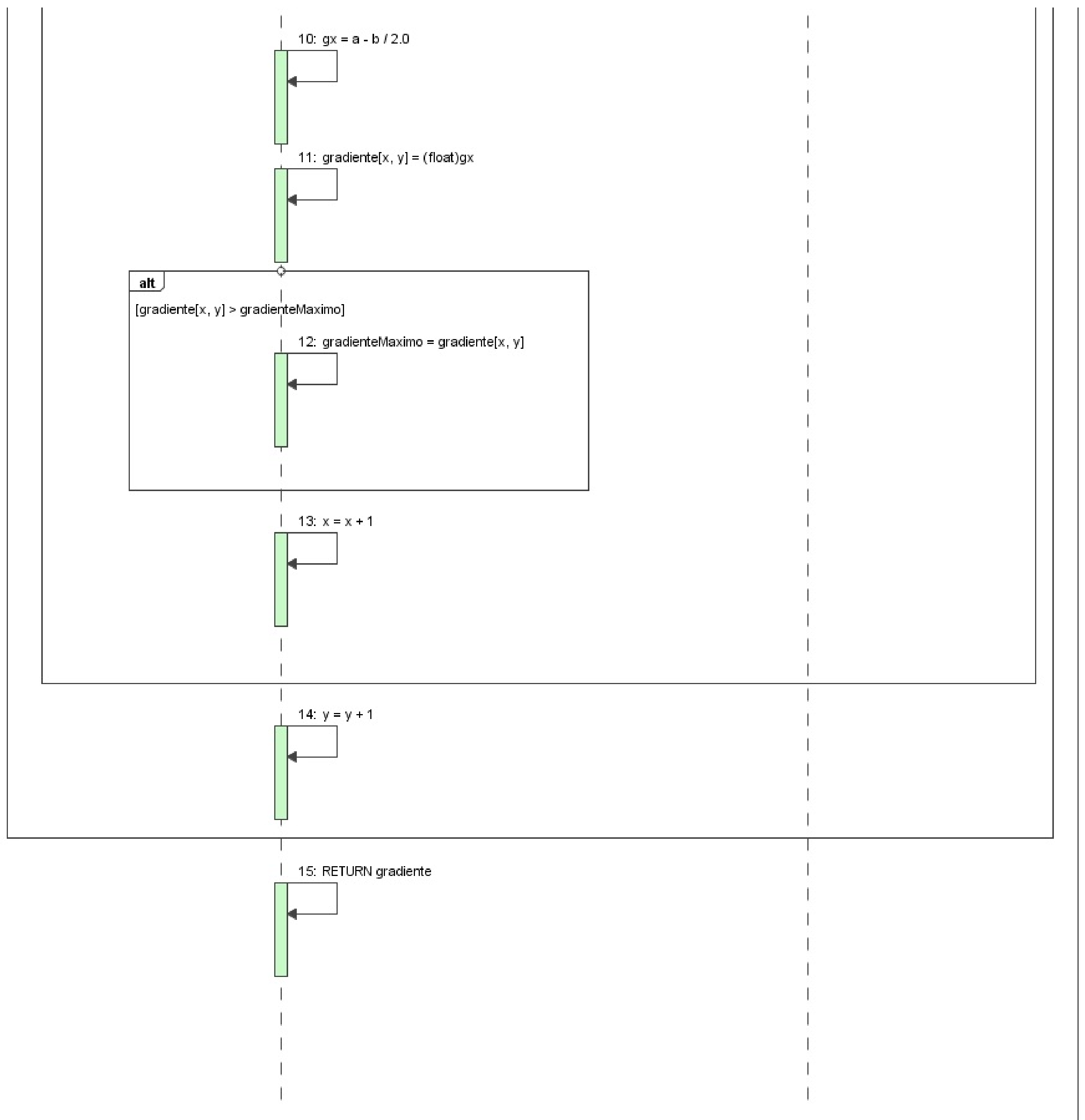

Figura A.66: Clase Gradiente. Segunda parte de la función *gradienteXR*.

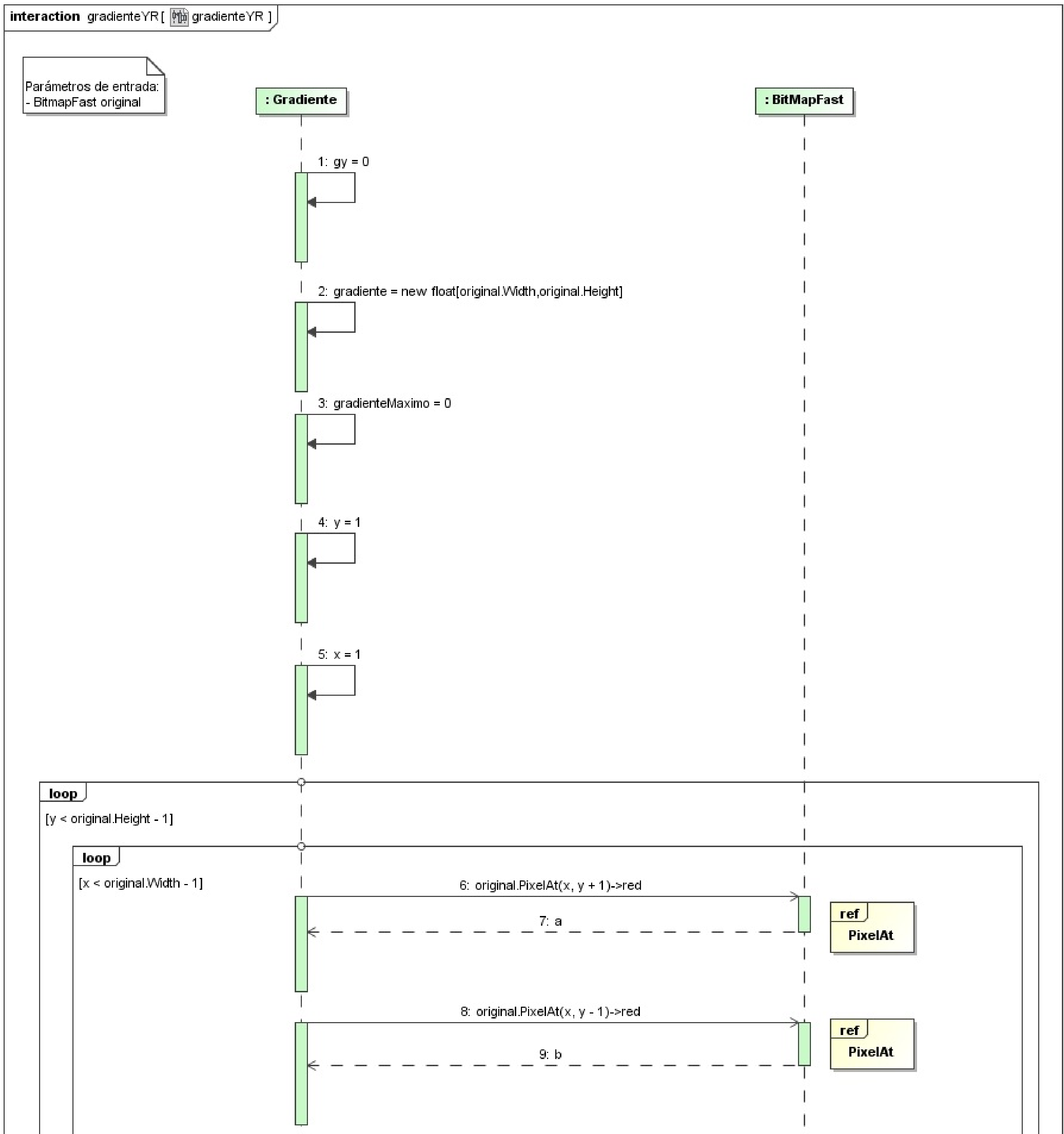

Figura A.67: Clase Gradiente. Primera parte de la función *gradienteYR*.

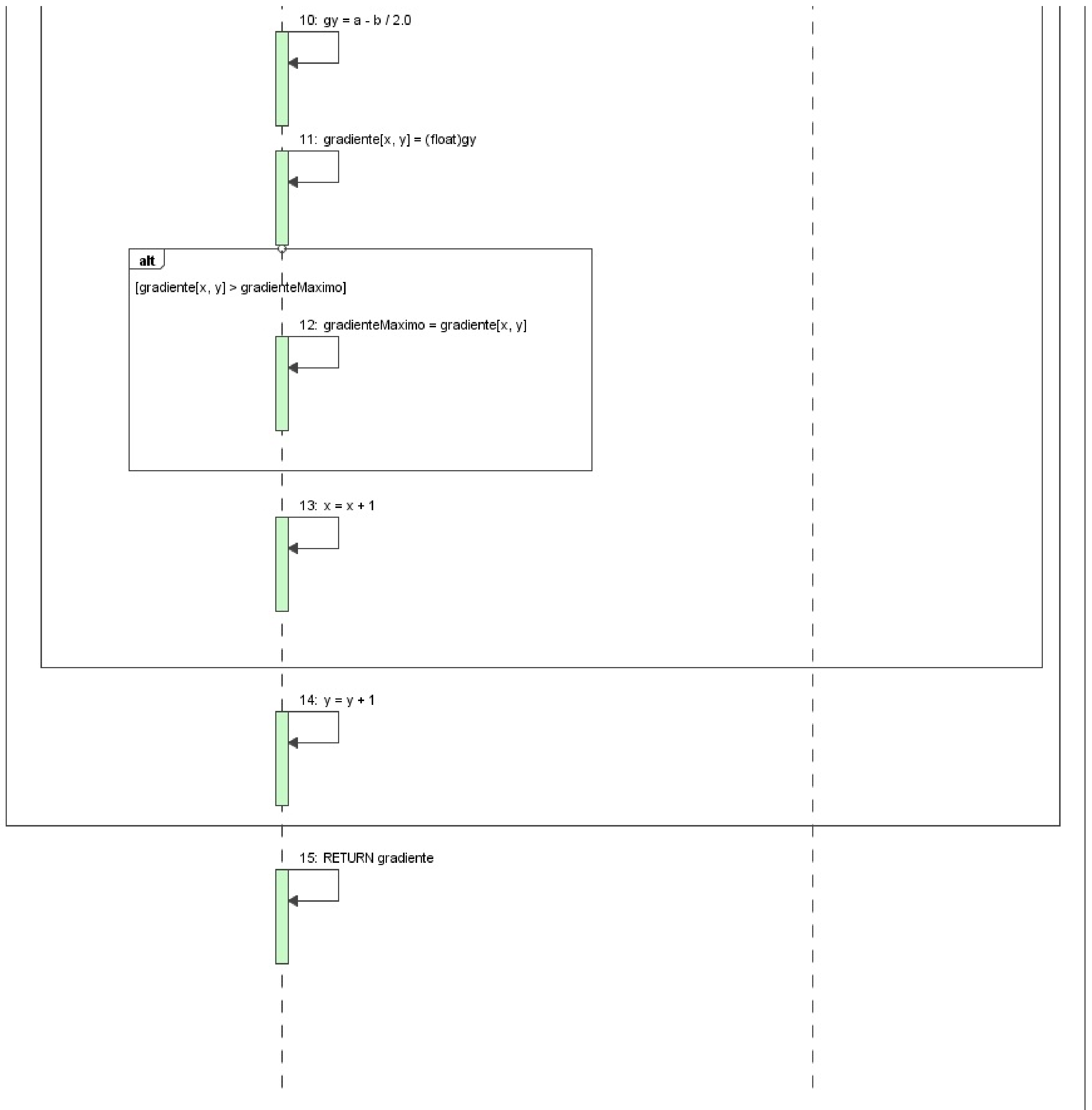

Figura A.68: Clase Gradiente. Segunda parte de la función *gradienteYR*.

Apéndice B

Interfaz de programación de la aplicación

La aplicación consta de una interfaz de programación bien definida que sirve para que un programador pueda hacer uso de los diferentes algoritmos en algún otro proyecto relacionado.

B.1 Espacio de nombres ProyectoSI

Dentro de este espacio de nombres se sitúa la siguiente clase:

B.1.1 Clase ProyectoSI

La clase ProyectoSI contiene los métodos con los distintos algoritmos que incorpora nuestra aplicación:

- **AlgoritmoEstandar** (Bitmap imagenI, Bitmap imagenD, Configuracion config)
- **AlgoritmoLankton** (Bitmap imagenI, Bitmap imagenD, Configuracion config)
- **AlgoritmoLankton2** (Bitmap imagenI, Bitmap imagenD, Configuracion config)
- **MostrarAnaglifo** (Bitmap imagenI, Bitmap imagenD)
- **DeteccionObjetos** (Bitmap imagenI, Bitmap imagenD, Configuracion config)
- **DeteccionBitmapObjetos** (Bitmap imagenI, Bitmap imagenD, Configuracion config)

A continuación se describen los métodos de esta clase:

Bitmap AlgoritmoEstandar (Bitmap imagenI, Bitmap imagenD, Configuracion config)

Breve descripción
Método principal para ejecutar el algoritmo estándar de correspondencia.
Valor de retorno
Un Bitmap con el resultado del algoritmo aplicado al par de imágenes.
Parámetros
imagenI Imagen izquierda

imagenD Imagen derecha
config Parámetros de configuración del algoritmo

Bitmap AlgoritmoLankton (Bitmap imagenI, Bitmap imagenD, Configuracion config)

Breve descripción
Método principal para ejecutar el algoritmo de correspondencia de Shawn Lankton.
Valor de retorno
Un Bitmap con el resultado del algoritmo aplicado al par de imágenes.
Parámetros
imagenI Imagen izquierda
imagenD Imagen derecha
config Parámetros de configuración del algoritmo

Bitmap AlgoritmoLankton2 (Bitmap imagenI, Bitmap imagenD, Configuracion config)

Breve descripción
Método principal para ejecutar la segunda versión del algoritmo de correspondencia de Shawn Lankton.
Valor de retorno
Un Bitmap con el resultado del algoritmo aplicado al par de imágenes.
Parámetros
imagenI Imagen izquierda
imagenD Imagen derecha
config Parámetros de configuración del algoritmo

Bitmap Anaglifo (Bitmap imagenI, Bitmap imagenD)

Breve descripción
Método principal para mostrar un anaglifo a partir de dos imágenes.
Valor de retorno
Un Bitmap con el resultado del algoritmo aplicado al par de imágenes.
Parámetros
imagenI Imagen izquierda
imagenD Imagen derecha

String DeteccionObjetos (Bitmap imagenI, Bitmap imagenD, Configuracion config)

Breve descripción
Método principal para obtener un XML en el cual se describa el objeto que se detecte en la escena pasada como parámetro.
Valor de retorno
XML con el contenido de la identificación del objeto.
Parámetros
imagenI Imagen izquierda
imagenD Imagen derecha
config Parámetros de configuración del algoritmo

ContenedorBitmapDisparidad DeteccionBitmapObjetos (Bitmap imagenI, Bitmap imagenD, Configuracion config)

Breve descripción

Método principal para obtener un Bitmap en el cual se marque el objeto detectado dado un par de imágenes obtenidas mediante cámaras. Ver también el método DeteccionObjetos.

Valor de retorno

Bitmap con el objeto identificado en color verde y disparidades media y máxima.

Parámetros

imagenI Imagen izquierda
imagenD Imagen derecha
config Parámetros de configuración del algoritmo

Bibliografía

[FS95] Y. Freund and R. Schapire. A desicion-theoretic generalization of on-line learning and an application to boosting. In *Computational Learning Theory*, pages 23–37. Springer, 1995.

[KSK06] A. Klaus, M. Sormann, and K. Karner. Segment-based stereo matching using belief propagation and a self-adapting dissimilarity measure. In *Pattern Recognition, 2006. ICPR 2006. 18th International Conference on*, volume 3, 2006.

[Lan07] Shawn Lankton. 3D Vision with Stereo Disparity, 2007.

[PDIC07] G. Pajares and J.M. De la Cruz. Visión por computador. Imágenes digitales y aplicaciones. *RA-MA*, 2007.

[PI05] R.S. Pressman and D. Ince. *Software engineering: a practitioner's approach*. McGraw-Hill New York, 2005.

[VJ04] P. Viola and M.J. Jones. Robust real-time face detection. *International Journal of Computer Vision*, 57(2):137–154, 2004.