Proyecto 2

IC3002 - Análisis de Algoritmos Profe.: Yuen Law Wan I Semestre 2020

Joseph Tenorio Pereira
2019064588

Jose Pablo Muñoz Montero
2019061904

***Abstract*—The following documentation revolves around the creation and analysis of a Monte Carlo path tracing algorithm for two dimensional images. This algorithm is based on starting with a black canvas and casting light rays out of every pixel in the canvas. The path of these rays will be followed, and by combining its reference color with the color of every surface it bounces on, the distance it travels, and the color of the light source reached, the correct color value will be assigned to each pixel. Since this is a Monte Carlo algorithm, the accuracy of the render is considered a "bet", since it varies with every try and improves by increasing the resources used. Nevertheless, several methods of optimization were applied, such as dynamic programing. To begin, a brief introduction on path tracing will be provided. Next, the thought process behind the algorithm will the thoroughly explained. Additionally, several rendered results will be presented to demonstrate the accuracy of the algorithm with varying parameters. Last, conclusions will be drawn.**

I. Introducción

II. Trabajo Relacionado

III. Métodos

A continuación se dará una explicación detallada del algoritmo utilizado para iluminar la imagen. Para comenzar, la técnica elegida fue recorrer cada pixel de la imagen e ir asignándole a estos su color calculado. Para el cálculo del color de cada pixel individual el algoritmo se dividirá en tres secciones para facilitar su compresión. La primera siendo un Ray Tracer que se encarga de calcular la iluminación directa para cada pixel. La segunda un Monte Carlo Path Tracer para calcular la iluminación indirecta y color bleeding para cada pixel. Y la tercera una serie de funciones geométricas para calcular el camino de los rayos de luz utilizados en las secciones anteriores. Para finalizar, se discutirá la optimización implementada.

Antes de comenzar es importante aclarar las estructuras de los objetos utilizados en el programa. Primero, los colores todos se interpretan como su valor RGB en una lista de tres elementos. Segundo, las figuras geométricas en la imagen (en este caso paredes) se crean a partir de un objeto línea que consiste de dos puntos (x, y), el programa solo implementa líneas rectas ya sean horizontales, verticales o diagonales. Tercero, las fuentes de luz pueden ser un único punto o una línea. Los rayos de luz se representan mediante un objeto rayo que consiste de un punto (x, y) del cual parte y un segundo punto que indica su dirección infinita, este punto se calcula a partir de un ángulo de dirección. Además, el vector es normalizado para simplificar los cálculos. Por último, la imagen de referencia y la imagen a pintar fueron traducidas a matrices de Numpy que se componen por listas con los valores RGB de cada pixel.

La primera sección del algoritmo encargada del cálculo de la iluminación directa comienza con el color en negro. Siguiente, se hace un ciclo para cada fuente de luz que se esté utilizando. Dentro de este ciclo se crea una línea recta del pixel actual a la fuente de luz actual y se le calcula la distancia mediante el teorema de Pitágoras. Luego, se debe revisar si existe alguna pared que interseque dicha línea. Esto se verifica mediante un ciclo para todas las paredes creadas, dentro del cual se busca un punto de intersección utilizando los determinantes ref1. Cabe resaltar que no se toman en cuenta las paredes transparentes. Si existiera algún punto de intersección el ciclo se interrumpe y el color del pixel se mantiene en negro. Si no existe ningún punto de intersección se procede a calcular la intensidad de la luz directa con la distancia calculada y la ley de la inversa del cuadrado ref2. Además, se obtiene el color de la imagen de referencia para el pixel actual. El color final es la multiplicación del color de referencia, el color de la fuente de luz y la intensidad calculada. De esta forma los colores se van sumando para cada fuente de luz, y al terminar el ciclo se promedia entre la cantidad de luces, así queda el pixel con su color de iluminación directa asignado.

A continuación, es necesario calcular si el pixel actual recibe luz indirectamente a través de rayos que rebotan en las paredes establecidas y llegan a él. Estos pueden venir de las mismas fuentes que lo iluminan directamente, o inclusive de alguna que no lo ilumina directamente. Adicionalmente, al rebotar, estos rayos llevan con ellos un poco del color de la pared en que rebotaron y lo pasan al pixel de destino como parte del efecto color bleeding. Esta sección del algoritmo es un Monte Carlo y se basa en lanzar rayos en direcciones aleatorias desde el pixel actual y seguir su camino para verificar si proveen iluminación indirecta. Por ser Monte Carlo debe contar con un parámetro de entrada que se refiere a la cantidad de recursos que se van a utilizar, en este caso son la cantidad de rayos lanzados por pixel. La función comienza con un ciclo para la cantidad de rayos elegida, dentro de este se calcula un ángulo aleatorio de 0 a 2 * π radianes (un ciclo completo). Luego, se crea un rayo con dicho ángulo y se llama una función recursiva con ese rayo para seguir su camino. Es importante mencionar que la función recursiva utiliza otro parámetro de entrada que se refiere a la cantidad de rebotes realizados, con tal de poder especificar un máximo y anular el rayo o

direccionarlo intencionalmente a la fuente de luz.

La función recursiva comienza preguntando por su condición de parada, es decir, si la cantidad máxima de rebotes no ha sido superada. Como no es el caso, se procede a revisar si el rayo interseca alguna pared en su camino. Esto se verifica con un ciclo para todas las paredes existentes y la misma fórmula de determinantes utilizada anteriormente ref1 la cual además retorna la distancia a la intersección. Sin embargo, en este caso si es necesario completar el ciclo ya que se debe obtener la intersección más cercana, la que retorna la menor distancia. Si no existe ninguna intersección la función recursiva retorna el color negro, ya que el rayo no aporto ningún color. Si existiera una intersección se calcula la intensidad de la luz que proviene de ese rebote con la distancia y la ley de la inversa del cuadrado ref2. Además se obtiene el color de la imagen de referencia para el pixel del cual partió el rayo. Por último, la función retorna el color bleeding producido por el rayo lanzado: la multiplicación de la intensidad, el color de referencia y el color entrante de donde ocurrió el rebote.

Antes de poder retornar se debe calcular el color entrante recursivamente. Para esto se crea un nuevo rayo a partir de donde ocurrió el rebote, el cual puede tener tres comportamientos diferentes. Si el rebote ocurrió en una superficie especular se obtiene un rayo con una dirección basada en el ángulo correcto de reflexión. Si el rebote ocurrió en una superficie no especular se obtiene un rayo con una dirección totalmente aleatoria, basada únicamente en no atravesar el segmento donde reboto. Por último, si el rebote ocurrió en una superficie no especular pero el siguiente rayo es el último de acuerdo al máximo, se obtiene un rayo direccionado específicamente a alguna fuente de luz accesible para no desperdiciarlo. Luego, se llama a la función recursivamente con el nuevo rayo y la cantidad de rebotes aumentada en uno, hasta llegar al último rayo permitido o hasta que el rayo no tenga ninguna intersección.

Al llegar al último rayo según el máximo se debe revisar si el rayo interseca con una fuente de luz, ya que antes estas estaban siendo ignoradas, y luego se debe revisar si no existe ninguna pared de por medio. Para esto se utilizan los mismos cálculos mencionados anteriormente. Si el rayo no cumple con ambas condiciones la función retorna negro y recursivamente continua retornando negro hasta llegar al pixel original. Si el rayo si cumple, se obtiene la intensidad con respecto a la luz, el color de referencia y se retorna la multiplicación de estos con el color de la luz, siendo esto el color de donde ocurrió el último rebote antes de llegar a la luz. Recursivamente se van acumulando los colores de los rebotes hasta llegar al pixel original y salir de la función recursiva retornando el color final producido por el color bleeding. Esta función también acumula la distancia total recorrida por el rayo y el color de la fuente de luz a la que llego para retornarlos.

Una vez de vuelta en la función original, si el color retornado por la función recursiva es diferente de negro, este se le suma al color del pixel (el cual en este momento tiene nada más su valor de iluminación directa) y se inicia un conteo de rayos válidos, es decir, rayos que llegaron a una fuente de luz. Además, se utiliza la distancia total recorrida para calcular la intensidad de la iluminación indirecta y junto con el color de referencia y el color de la fuente de luz que se intersecó se obtiene otro color para sumar al pixel y otro rayo valido. Una vez terminado el ciclo para la cantidad de rayos deseada, el color final del pixel se calcula promediándolo de nuevo entre la cantidad de rayos válidos y se le asigna a la imagen.

En cuanto a las funciones encargadas de crear los vectores que representan a rayos que rebotaron, estas se dividen en rebote especular, rebote dirigido y rebote aleatorio. Previo a explicar sus diferencias, cabe resaltar que los tres procedimientos crean un vector cuyo origen se ubica en el punto de intersección del rayo anterior con la superficie en cuestión. Debido a lo anterior, el grueso de los cálculos realizados tiene por objetivo determinar el ángulo a asignar al vector creado. Para evitar malentendidos, de aquí en adelante se entiende como ángulo de los vectores (o de los rayos que estos representan) a la cantidad de radianes existentes entre el límite del primer y cuarto cuadrante del plano cartesiano (mitad positiva del eje X) y la línea de rayo, esta última siendo aquella línea resultante de extender el rayo rebotado hasta el infinito en ambas direcciones. Además, se recuerda que dichos ángulos incrementan en el sentido de las agujas del reloj. También cabe resaltar que el plano cartesiano mencionado anteriormente tiene su origen en el centro de la imagen.

La primera función de rebote diseñada fue la de rebote aleatorio, la cual se emplea en aquellos rebotes que ni son el último rebote del rayo original ni se producen en superficies especulares. En este caso, el ángulo del nuevo rayo debe ser aleatoriamente escogido entre dos rangos de ángulos, cada uno con uno con un ancho π radianes y cada uno representando uno de los lados de la superficie, es decir, el punto medio de cada rango corresponde al valor del ángulo formado por la mitad positiva del eje X y los respectivos rayos (geométricos, no de luz) perpendiculares a la superficie y con origen en el punto de rebote. En el caso de los segmentos verticales, los dos rangos posibles a usar corresponden a $]\pi/2,3\pi/2[$ para el lado izquierdo y $]3\pi/2,2\pi[\cup$

Finalizando, es evidente que el algoritmo cuenta con una gran cantidad de ciclos y por ende puede llegar a tener un tiempo de ejecución muy largo, como se evidenciará en los resultados. No obstante, sí fue posible encontrar optimizaciones para mejorar dicha ejecución. Primero, una optimización sencilla es el hecho de que al lanzar el último rayo de un rebote no especular este se direcciona intencionalmente hacia alguna fuente de luz. De esta forma es mucho menos probable que el rayo se desperdicie, a menos que haya una pared de por medio. Con esta optimización fue posible reducir fuertemente la cantidad de rayos necesitados para producir una imagen de buena calidad, ya que la gran mayoría de rayos terminaban siendo válidos.

Segundo, se implementó un cierto grado de programación dinámica en el algoritmo, para evitar realizar cálculos que ya se habían realizado anteriormente. Por ejemplo, cuando el ultimo rebote de un rayo interseca una fuente de luz, se calcula el color del pixel donde ocurrió el rebote para utilizarlo en el

color bleeding. Pero, este valor es a su vez el color de la iluminación directa que le provee esa fuente de luz a dicho pixel. Por ende, resulta beneficioso guardar este cálculo para no tener que volverlo a realizar cuando se vaya a calcular la iluminación directa para ese pixel. De la misma manera se debe guardar el color calculado de iluminación directa para cada pixel, para no volverlo a calcular cuando un rayo rebote en un pixel ya iluminado. Indirectamente de esta manera también se evita calcular rebotes finales que ya hayan ocurrido antes. La estructura elegida para almacenar estos colores es una matriz de tres dimensiones, igual a la matriz utilizada para guardar las imágenes, pero ahora para cada pixel (x, y) se guardan múltiples colores: los valores de iluminación directa respecto a cada fuente de luz. Así, antes de realizar cualquier cálculo de iluminación directa se debe preguntar si ya existe un valor para el pixel actual respecto a la fuente de luz actual. Esta optimización logró reducir el tiempo de ejecución del algoritmo porque los pixeles en áreas potenciales de rebotes ya no repiten cálculos.

IV. Análisis de resultados

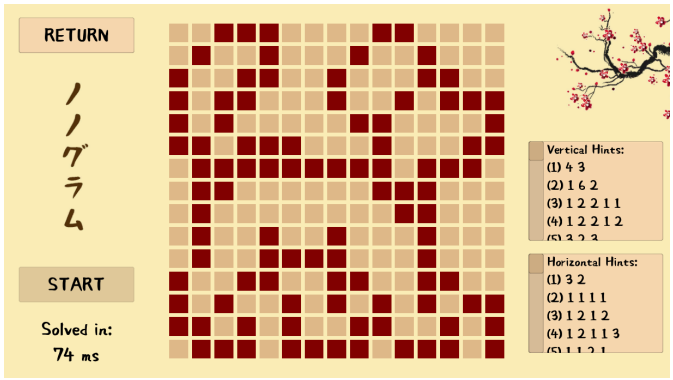

Fig. 1. Tiempo de ejecución y cuadrícula del principal nonograma de prueba en *Unity*.

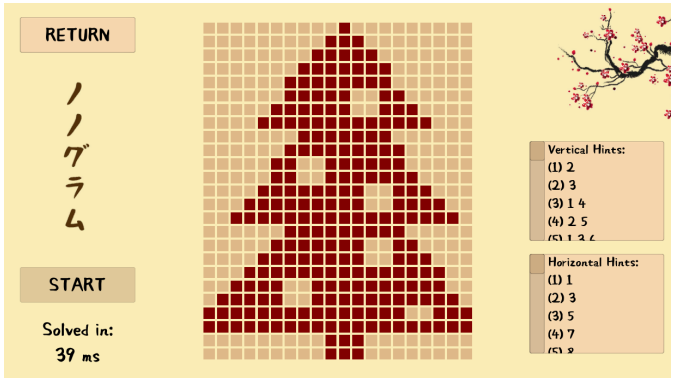

Fig. 2. Tiempo de ejecución y cuadrícula de un nonograma de prueba en *Unity*.

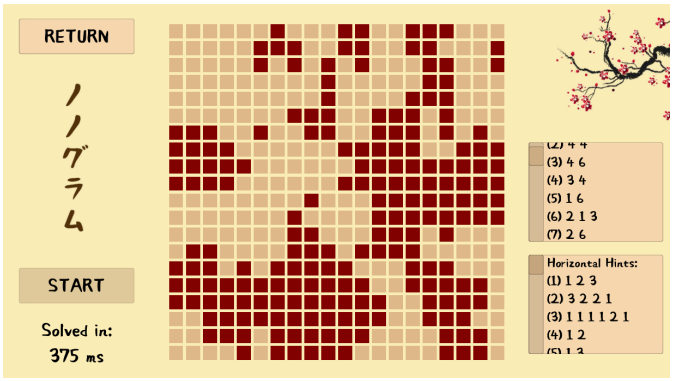

Fig. 3. Tiempo de ejecución y cuadrícula de un nonograma de prueba en *Unity*.

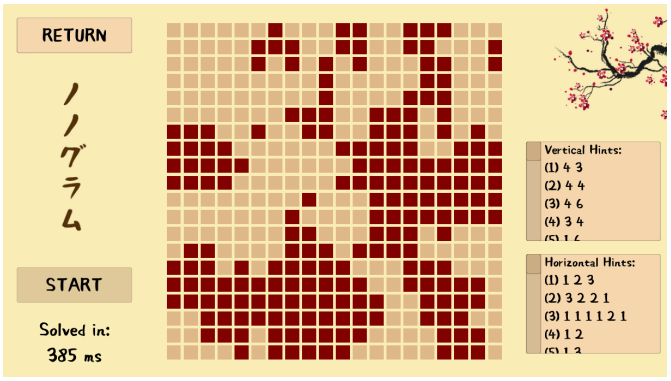

Fig. 4. Tiempo de ejecución y cuadrícula textitUnity.

V. Conclusión

VI. Acceso al proyecto

El repositorio de GitHub en el cual se ubica el proyecto de *Unity* creado puede ser accesado mediante el siguiente link: https://github.com/TilapiaBoi/AnalisisPRY2

Bibliografía

[1] J. Lázaro, *Backtracking*, Departamento de Ciencias de la Computación, Universidad de Alcalá, n.d. Accessed on: Mar. 18, 2020. [Online]. Available: ftp://www.cc.uah.es/pub/Alumnos/G_Ing_Informatica/Algoritmia_y_Complejidad/anteriores/Apuntes/08_Backtracking.pdf

[2] "Nonogram", n.d. Accessed on: Mar. 18, 2020. [Online]. Available: https://www.definitions.net/definition/Nonogram

[3] C. Wu , D. Sun, L. Chen, K. Chen, C. Kuo, H. Kang and H. Lin, "An Efficient Approach to Solving Nonograms", *IEEE Transactions On Computational Intelligence And AI In Games*, vol. 5. no. 3, Sept. 2013, pp. 251-264. Accessed on: Mar. 18, 2020. [Online]. Available: http://perpustakaan.unitomo.ac.id/repository/AnEfficientApproachtoSolvingNonograms.pdf

[4] Tech With Tim, *Python Sudoku Solver Tutorial with Backtracking p.2*, Apr. 4, 2019. Accessed on: Mar. 18, 2020. [Video file]. Available: https://www.youtube.com/watch?v=lK4N8E6uNr4&t=844s

[5] M. Richards, *Backtracking Algorithms in MCPL using Bit Patterns and Recursion*, Computer Laboratory, University of Cambridge, Feb. 23, 2009. Accessed on: Mar. 18, 2020. [Online]. Available: https://www.cl.cam.ac.uk/~mr10/backtrk.pdf