Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

Tarea 2

IIC2133 - Estructuras de datos y algoritmos

Primer semestre, 2016

Entrega: Lunes 9 de Mayo

Objetivos

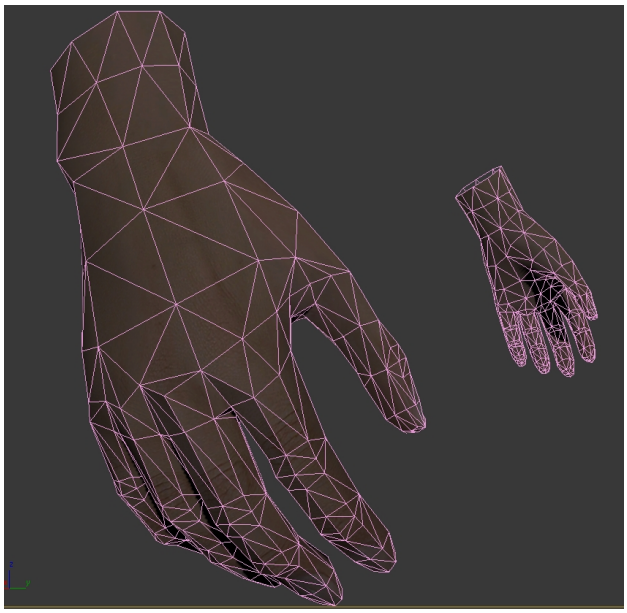
- Utilizar árboles para resolver un problema de optimización
- Aplicar la técnica algorítmica de Dividir para Conquistar en un contexto geométrico
- Investigar sobre el estado del arte en estructuras de datos

Introducción: Ray Tracing

Para poder hacer esta tarea necesitas saber qué es el Ray Tracing. No para implementarlo (no es esa la tarea) pero sí para poder optimizarlo usando una estructura de datos.

Ray tracing es una técnica de *rendering*. Rendering significa simplemente *convertir una escena en 3D a una imagen*. Ray tracing nos permite obtener imagenes muy realistas, es más: todas las películas 3D (incluyendo las de Pixar y Disney) se hacen a partir de raytracing.

El funcionamiento del Ray Tracing es el siguiente: existe una escena en 3D, descrita por su geometría. En general se trabaja con primitivas matemáticas y para los objetos complejos se los descompone en **triángulos**, los cuales permiten construir cualquier figura. Para efectos de esta tarea se trabaja únicamente con triángulos. Junto con la geometría, la escena tiene una cámara, luces y distintos materiales que especifican como dibujar cada objeto.

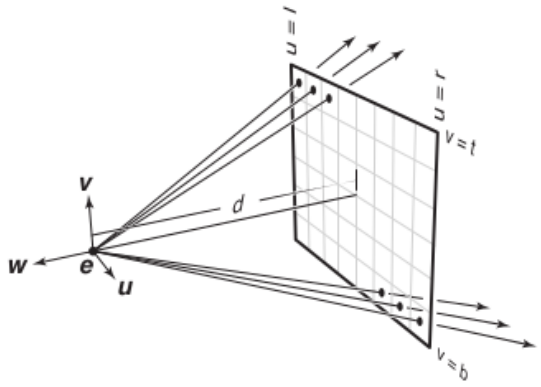


Modelo triangular de una mano

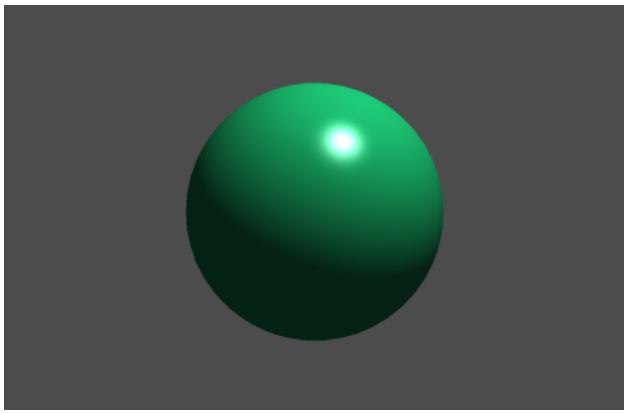
El método general para generar una imagen, ya sea en una cámara o en nuestros ojos es como sigue: la luz con la que vemos el mundo llega a nuestros ojos y a nuestras cámaras en línea recta. Comienza en las fuentes de luz, rebota en los objetos y eventualmente llega a destino.

Ray tracing simula los rayos de luz, de donde viene su nombre. Sin embargo, simular *todos* los rayos de luz es poco eficiente, pues muy pocos rayos llegarán finalmente al agujero de nuestra cámara.

Por esto es que el algoritmo asume que le llega luz a cada píxel. Para saber de qué color es esa luz simula el camino inverso de los rayos: desde la cámara, cruzando cada píxel, sale un rayo. Cuando un rayo toca un objeto, se revisa si el objeto está iluminado (por algunas o todas las fuentes de luz de la escena) y se obtiene un color a mostrar en la pantalla.



Rayos saliendo de la cámara



Esfera con una luz arriba. La esfera se observa *parcialmente iluminada* por la luz (sólo algunos de sus triángulos están expuestos)

No solo eso, sino que además de los rayos que salen de la cámara (llamados rayos principales) existen rayos secundarios para simular otros comportamientos de la luz. Por ejemplo

- Sombras: Desde cada punto de intersección se lanza un rayo hacia cada la luz. Si choca con algún objeto en el camino entonces a ese punto no le está llegando esa luz.
- Reflejos: En caso de que un objeto sea reflectivo, se lanza un rayo adicional de acuerdo al ángulo de incidencia para computar el reflejo sobre la superficie del objeto.
- Refracción: En caso de que un objeto sea refractante, se lanza un rayo adicional de acuerdo al indice de refracción y al ángulo de incidencia para computar los reflejos internos y distorsiones de un objeto.

Estos rayos tienen la misma precedencia que los rayos principales, y utilizan exactamente el mismo algoritmo para encontrar la intersección.

Además de eso, para suavizar los bordes de los objetos se necesita usar *antialiasing*, el cual en su modo más elemental requiere lanzar más de un rayo por píxel, ponderando los colores obtenidos por cada uno. Si bien esto mejora la calidad de la imagen obtenida, aumenta el tiempo de ejecución en un factor cuadrático, por lo que debe ser usado con cuidado.

Problema: demasiadas intersecciones

Para saber con qué objeto choca el rayo de luz, se utiliza un algoritmo que a partir de la descripción geométrica del rayo (origen y dirección) y de la del triángulo candidato (posición de los 3 vértices), determina si el rayo intersecta al triángulo y obtiene la distancia de intersección.

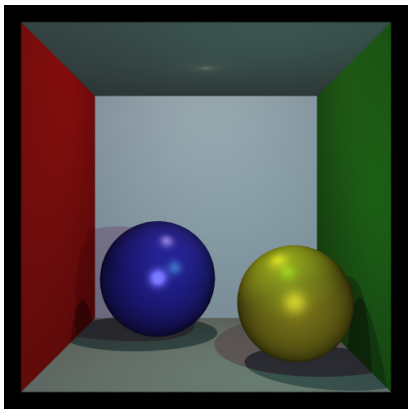
¿Qué triángulo es el que ve el rayo? El que intersecta con él a la menor distancia posible, pues esto significa que los otros triángulos que también intersectan al rayo están más lejos, escondidos detrás.

Raytracing se suele implementar de tal forma que para cada píxel, se intenta intersectar su rayo con cada objeto de la escena. Dentro del rayo se guarda una referencia al triángulo más cercano con el que ha intersectado. Luego de probar con todos los objetos, la referencia que tiene el rayo es la del objeto que **efectivamente ve**.

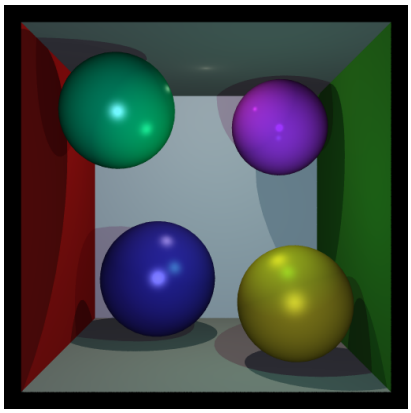
Esto nos lleva a un problema. La función `intersect(ray, triangle)` es muy costosa, mucho más que cualquier operación anterior o posterior. Calculemos la complejidad del algoritmo, en términos de la cantidad de veces que llamamos a la función:

- Sea p el número de píxeles, r el número de rayos por píxel. La cantidad de rayos principales será $m = p \cdot r$
- Existen n triángulos en la escena.
- Si por cada píxel hacemos n llamadas a `intersect`:
  - $\Rightarrow$ `Ray tracing` es $\Theta(m \cdot n)$.

En la práctica, esto hace si tenemos una escena con 2500 triángulos como la *Cornell Box* y le agregamos 2 esferas (cada una con aproximadamente 1200), el tiempo de rendereo se duplica. Esto es sin considerar los rayos secundarios.

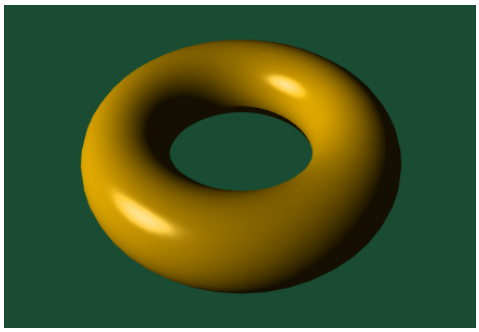


Cornell Box, toma t segundos



Con 2 esferas más, toma $2t$ segundos

¿Es posible usar menos veces `intersect`? Pensemos en la siguiente escena:



Toro

Es una escena sencilla, pero por cada píxel – incluso aquellos que claramente no verán al toro – estamos probando con cada triángulo por si acaso lo intersectamos y resulta ser el más cercano a la cámara en la trayectoria del rayo.

Hay muchas formas de resolver esto. Esa es vuestra tarea, que se describe a continuación.

Tarea: Organización del espacio

Tu tarea es optimizar el algoritmo de ray tracing original. Para ello, se ha preparado una librería que implementa todo el algoritmo. La librería carga escenas, muestra una ventana, implementa geometría (rayos, triángulos y la función `intersect`, entre otros). En definitiva, la librería ya permite formar imágenes. Sin embargo, para determinar el triángulo con el que choca cada rayo se hace por fuerza bruta, llamando a `intersect` para cada triángulo de la escena.

Se sabe que la complejidad óptima de un ray tracer es $O(m \cdot log(n))$. Tu objetivo es que el ray tracer llegue a esta eficiencia. Debes crear una estructura de datos que ordene espacialmente los triángulos de la escena, de forma tal que para saber con qué triángulo choca un rayo sea posible probar en primer lugar con los triángulos más cercanos a éste y descartar sin usar `intersect` a los triángulos que no están dentro de su trayectoria.

Detalles de la implementación

Descarga la carpeta de la Tarea 2 del repositorio *Syllabus*. Dentro de `src` encontrarás 3 carpetas: RAYTRACER, MODULES y SOLUTION. **Todo el código atingente a tu problema** está en la carpeta SOLUTION. En solution están los siguientes archivos:

- **geometry.h**: encapsula los componentes y operaciones de la geometría del problema. Rayos e intersecciones con triángulos.
- **scene.h**: encapsula las propiedades de una escena. Triángulos y puntos de la escena.
- **vector.h**: struct Vector, de 3 coordenadas. Operaciones vectoriales.
- **manager.h**: maneja la intersección con rayos.
- **manager.c**: código detrás de **manager.h**. Por el momento, encuentra el triángulo con que intersecta un rayo por el método de fuerza bruta.

De todos los archivos de la librería, **sólo tienes permitido** modificar **manager.h** y **manager.c**. Sí puedes crear nuevos archivos y agregarlos a la carpeta según te convenga.

Te recomendamos fuertemente que leas los comentarios de los headers presentes en SOLUTION, para que tengas bien claro lo que hace cada uno.

Adicionalmente en su propia carpeta viene el *structure_tester.c* el cual podrás modificar a tu gusto para debugear los leaks de tu código, dado que GTK+ tiene incompatibilidades con valgrind.
El programa compilado se llama `raytracer` y se usa de la siguiente manera:

```
./raytracer escena factor -f
```

Donde `escena` es la ruta al archivo de escena a renderear.

`factor` es el factor de *antialiasing*. Indica que deben tomarse factor^2 muestras por píxel

`-f` de *flat* es un parámetro opcional que indica que no se deben suavizar los triángulos.

Entrega

Deberás entregar en tu repositorio de Github una carpeta de nombre **Tarea02**. Y dentro de esta carpeta, una carpeta **Programa** y una carpeta **Informe**. Dentro de *Programa* deberás tener un *Makefile* y una carpeta **src** con todo el código de tu tarea. Ésta se compilará con el comando `make` dentro del mismo directorio, y tu programa deberá generarse en un archivo de nombre `raytracer`.

En la carpeta *Informe* deberás tener un archivo de nombre *Informe.pdf*.

Se recogerá el estado de la rama `master` de tu repositorio, 1 minuto pasadas las 24 horas del día de entrega. Recuerda dejar ahí la versión final de tu tarea.

Evaluación

Se otorgará un 70 % de la nota a que tu programa renderee correctamente las escenas de prueba. El 30 % de esta nota será asignado según el rendimiento de tu solución comparado con tus compañeros. El otro 30 % de la nota de la tarea vendrá del informe.

Durante las semanas de la tarea se subirán escenas de prueba con las que podrás probar el rendimiento de tu solución. Estos están separados en 4 categorías de complejidad creciente.

1. Easy: La escena contiene un solo objeto y una sola luz.
2. Normal: La escena contiene múltiples objetos, luces y materiales reflectantes.
3. Hard: En adición a lo anterior, la escena contiene objetos con una alta cantidad de triángulos.
4. Lunatic: Todo lo anterior, llevado al límite de la demencia.

Tu código se evaluará rendereando 3 escenas de cada categoría, por 0.5 puntos c/u. Se comparará el output de tu programa con el generado por el algoritmo de fuerza bruta, y se asignará puntaje según que tan igual es.

Cada escena tiene un tiempo límite acorde a una métrica que será especificada más adelante (depende de tu computador y de las características particulares de la escena). De no terminar el render en ese plazo se corta el programa y se pasa a la siguiente escena. Este tiempo incluye tanto la construccion de tu estructura como el rendereo en si

Informe

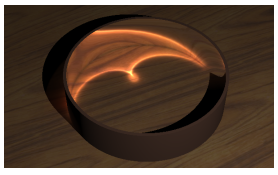
En tu informe deberás describir tu estructura, el criterio de construcción, por qué la elegiste y analizar su complejidad. También deberás hacer testing de su rendimiento, comparando cómo cambia si creas la estructura con distintos parámetros (si es que es parametrizable) y comparando su rendimiento con el approach de fuerza bruta que viene ya implementado.

Los detalles del informe serán publicados más adelante.

Bonus

Se otorgará una cierta cantidad de bonus a la nota de tu programa o de tu informe en caso de lograr las siguientes metas opcionales:

1. **Sin hacer agua**: (+10 % al *Programa*) este bonus será otorgado por que tu solución tenga 0 memory leaks al probarla con valgrind. Debe decir `All heap blocks were freed -- no leaks are possible`
2. **Magia negra**: (+30 % al *Programa*) este bonus se trata de implementar adicionalmente la estructura secundaria (un KD-Tree de puntos) para llevar a cabo el algoritmo de *Photon Mapping* el cual permite recrear *caustics*. Si quieres implementar esto, contacta a tus ayudantes, con gusto te guiarán.



Caustics por reflexión

Caustics por refracción

3. **Ortografía perfecta**: (+5 % al *Informe*) por tener 0 faltas de ortografía, obtendrás un 5 % de bonus en la nota de tu informe.
4. **Buen uso del espacio y del formato**: (+5 % al *Informe*) a juicio del corrector, si tu informe está bien presentado y usa el espacio y formato a favor de entregar la información, obtendrás un 5 % de bonus en tu informe.