The Raytracer

Fecha de entrega: 7 de diciembre, 2020

1 Temas

- Trazado de rayos (raytracing).
- Transformaciones inversas y transformaciones de rayo.
- Intersección de un rayo con esferas, cubos y mallas de polígonos planos convexos
- Modelo de iluminación de Phong

2 Enunciado

Un ray tracer contiene los componentes necesarios para:

1. enviar rayos a partir de la posición del observador hacia cada pixel,
2. intersecar los rayos con los objetos de la escena,
3. enviar rayos de sombra a partir de cada puntos de intersección a cada fuente de luz, y
4. iluminar los rayos, sumando las contribuciones de cada fuente de luz visible, y asignar un color al pixel.

Esta tarea[1] consiste en programar un **raytracer** con estas funciones. El raytracer deberá poder manejar, como primitivas básicas, esferas, cubos y mallas de polígonos planos convexos. La entrada de tu programa será la descripción de una escena compuesta por algunas de las primitivas antes mencionadas, ya sea organizadas jerarquicamente o no. Las escenas serán descritas con algún lenguaje de scripting (por ejemplo LUA (https://www.lua.org) con la ayuda de archivos con modelos en algún formato 3D (puedes leerlos con librerías como http://www.assimp.org) o del parser de archivos *.obj que acompaña el código de esta tarea. Te recomiendo empezar con escenas pequeñas para verificar tu progreso, ya que el proceso de rendering para escenas grandes tomará mucho tiempo de computo.

El raytracer deberá implementar las siguientes funcionalidades:

- Transformaciónes de cuerpos rígidos (para acomodar tus objetos en la escena)
- Las primitivas esfera, cubo, y polígono (suponerlos convexos y planos).
- Volumenes englobantes (esferas o cubos) para el tipo malla polígonal.
- Fuentes de luz puntual, con modelo de iluminación de Phong (ambiente + difuso + especular).
- Salida para una imágen resultante en formato PNG o JPEG.

Puedes usar normales **por cara** para el cubo y los tipos de malla poligonal para implementar los modelos de *shading*. **No es necesario** implementar las normales **por vértice**. Nota que el *shading* de Phong (interpolación de normales) no es necesario implementarlo ya que no se proporcionan las normales por vértice.

Deberás generar una escena de prueba (en el código adjunto es `sample.lua`) que demuestre: todos los tipos de primitivas requeridos, las fuentes de luz puntuales y al menos una superficie *brillante* (usando el modelo

[1] Esta tarea está inspirada de la tarea 4 del curso *Introduction to Computer Graphics* (CS488) de la Universidad de Waterloo.

de reflexión especular de Phong). El script debe escribir la imágen resultante en un archivo (por ejemplo, `sample.png` o `sample.jpeg`) y salir. Puedes ver scripts de ejemplo escritos en LUA en `data` para tomar ideas. También puedes ver las imágenes que corresponden a estos scripts de prueba en la página del curso de Waterloo (http://www.student.cs.uwaterloo.ca/~cs488/gallery-A4.html).

En esta tarea debes implementar solamente los rayos primarios y de sombra. *No es necesario* implementar los rayos secundarios recursivos, que son necesarios para implementar las reflexiones y transparencia. Nota técnica: la atenuación aplica solamente a los rayos de sombra y no a los rayos primarios.

Fondo de la imagen

Para empezar, debes definir un fondo de imagen (*background)* que no interfiera con los objetos de la escena. Esto significa que los colores no deben ser muy brillantes, o los patrones muy variados. Puedes intentar hacer efectos de un atardecer. Un fondo de color constante no es aceptable. Puedes parametrizar el fondo ya sea por posición del pixel o por la dirección del rayo. Esta última parametrización puede usarse, por ejemplo, para hacer un fondo de cielo con azul oscuro en el zenith, blanco en el horizonte y café abajo del horizonte, o implementar un cielo estrellado para una escena en el espacio. Usa tu imaginación. Este fondo deberá estár en todas las imágenes que hagas.

Escenas jerárquicas

La manera más sencilla de implementar un *ray tracer* jerárquico (para escenas con objetos modelados con una jerarquía) es transformar cada rayo del sistema coordenado de vista (VCS) al sistema coordenado del modelo (MCS) de cada modelo. El rayo transformado es intersecado con los modelos en una posición canónica, de tal manera de tomar ventaja de la forma especial de las ecuaciones. Para buscar la escena completa, para cada rayo se puede hacer un recorrido recursivo en un árbol, generando matrices de transformación mientras se avanza. Ten en cuenta que las fuentes de luz deben especificarse en el sistema coordenado del mundo (WCS), por lo que tendrás que transformar la intersección de MCS a WCS para hacer el *shading*.

Para calcular la matriz usada en la transformación de los rayos, deberás extender las llamadas del modelo para mantener la transformada *inversa* en cada nodo del modelo. Luego tendrás que componer estas transformadas en orden reverso mientras desciendes en el árbol para encontrar las transformaciones necesarias VCS a MCS para cada primitiva. Las intersecciones con los objetos se pueden hacer más eficientemente (y de manera más simple) usando la forma canónica para un objeto en MCS.

3 Desarrollo sugerido

Tienes la libertad de desarrollar tu código como quieras. La parte obligatoria de la tarea será desarrollar un *raytracer* no-jerárquico (Objetivos 1-6) y volúmenes englobantes (Objetivo 10). Una vez funcionando esta parte del código puedes agregar la parte jerárquica (Objetivos 8,9, transformaciones afines, una jerarquía general) para obtener crédito extra.

Finalmente, necesitarás hacer una escena original (Objetivo 7). Aunque puedes escribir tu escena al principio del proyecto se te recomienda esperar hasta saber si tu programa podrá tener modelos jerárquicos que te permitirán hacer una escena más interesante.

En esta tarea puedes tener salida en la consola, por ejemplo, puedes querer desplegar los parámetros de rendering y tener alguna indicación del progreso de tu raytracer (10%, 20%, etc.). De preferencia no despliegues toda la jerarquía de árbol de tu escena si quieres implementar el raytracer jerárquico.

Algunos consejos

Esta tarea representa mucho trabajo. Aunque los algoritmos parecen simples los detalles son importantes para hacerle funcionar. A continuación se describen algunos puntos ayuda:

- Abundan los problemas numéricos. Hay que estar particularmente atentos a lo siguiente:
  - Trata de minimizar el número de veces que normalizas vectores y normales. Cada vez que normalizas puedes introducir un pequeño error que más tarde puede derivar en problemas mayores.
  - La intersección del rayo con el objeto puede estar solo dentro del objeto. Cuando se arrojen rayos secundarios (como los rayos de sombras), el primero objeto que toca el rayo será el objeto mismo. Para evitar este problema, deshecha todas las intersecciones que ocurren muy cerca del origen del rayo.
  - Usar verificaciones de un *epsilon* en las rutinas de intersección, particularmente en la rutina de intersección entre el rayo y un polígono.
- En un raytracer jerárquico, en el recorrido *hacia abajo* debes transformar el punto y el vector que forman el rayo con la *transformación inversa*. En el recorrido *hacia arriba* debes transformar el punto de intersección con la transformación y (suponiendo que representes la normal como un vector columna) debes transformar la normal con la *transpuesta de la transformación inversa*. Es posible que esta transformación de la normal resulte en un valor diferente a cero de la cuarta coordenada, en cuyo caso deberás poner este valor en 0 (o, escribir un producto especial de matriz-vector que ignore la cuarta coordenada y utilice la transpuesta).

Para acelerar los cálculos en el recorrido hacia arriba puedes precalcular y almacenar todas las transformaciones VCS-MCS.

4 Interfase

Si decides utilizar el lenguaje de scripting LUA puedes utilizar la especificación siguiente de las primitivas. Tienes la opción de usar otro lenguaje para la creación de escenas mientras la funcionalidad sea equivalente.

`gr.nh_box(`*name*`,`(x,y,z)`,`r`)` – Regresa una caja no-jerárquica con el nombre *name*. La caja debe estar alineada con los ejes de su sistema coordenado de modelo (MCS), con una esquina en (x, y, z) y la esquina diagonalmente opuesta en $(x + r, y + r, z + r)$.

`gr.nh_sphere(`*name*`,`(x,y,z)`,`r`)` – Crea una esfera no-jerárquica con el nombre *name* y de radio r centrada en (x, y, z).

`gr.cube(`*name*`)` – Regresa una caja jerárquica con el nombre *name*. La caja debe estar alineada con los ejes de su MCS, con una esquina en $(0, 0, 0)$ y la esquina diagonalmente opuesta en $(1, 1, 1)$.

`gr.mesh(`*name*`,`$\{\{v_{1_x}, v_{1_y}, v_{1_z}\}, \ldots, \{v_{n_x}, v_{n_y}, v_{n_z}\}\}, \{\{p_{11}, p_{12}, \ldots, p_{1m}\}, \ldots, \{p_{n1}, p_{n2}, \ldots, p_{nm}\}\}$`)` – Crea una malla poligonal llamada *name* con los vértices y caras listados. La primera lista son las coordenadas de los vértices y la segunda es una lista de polígonos. Cada vértice está dado como una tripleta (x, y, z) y cada polígono es una lista de índices enteros de la lísta de vértices. Los vértices se indexan empezando a contar de 0. Los polígonos se suponen convexos y planos y también pueden tener un número arbitrario de vértices.

`gr.light(`$\{x, y, z\}, \{r, g, b\}, \{c_0, c_1, c_2\}$`)` – Crea una fuente de luz puntual en (x, y, z) de intensidad (r, g, b). Los parámetros de atenuación c_0, c_1, c_2 especifican la atenuación para una fuente de luz particular de acuerdo a la fórmula $1/(c_0 + c_1 r + c_2 r^2)$

`gr.render`($node, filename, w, h, eye, view, up, fov, ambient, lights$) – Produce una imágen de $w \times h$ pixels con el raytracer y la almacena en el archivo $filename$. La cámara se coloca en la posición eye, viendo hacia la dirección $view$ con el vector up indicando la dirección hacia arriba (todas estas cantidades son vectores de 3 elementos). Se utilizará un volumen de vista (*field-of-view*) de fov grados. El componente de ambiente de la luz debe tener una intensidad de $ambient$ (también un vector). Todas las luces utilizadas estarán listadas en $lights$.

Las funciones `gr.nh_box` y `gr.nh_sphere` te permitirán implementar una versión no-jerárquica de tu raytracer, ya que puedes colocar las esferas y cajas en posiciones arbitrarias de la escena. De esta manera puedes probar los aspectos de tu código tal como el shading y las sombras sin necesidad de que el raytracer jerárquico esté funcionando.

5 Código de apoyo (opcional)

Se anexa un código base que puedes utilizar (no es obligatorio) para programar tu raytracer. Este código fue creado en la U. de Waterloo como base para un raytracer en el curso de Gráficas CS488/688. Podrás encontrar los archivos en directorio `src/`:

`a4.cpp` – Contiene la función `render()` que necesitas implementar.

`image.cpp` – Una clase que almacena imágenes rectangulares en formato `PNG`. Contiene funciones para almacenar y cargar este tipo de imágenes.

`light.cpp` – Una clase simple que implementa fuentes de luz.

`mesh.cpp` – Una clase simple que implementa mallas poligonales.

`polyroots.cpp` – Solucionador robusto de raíces polinomiales. Puedes utilizarle en tus funciones de intersección del raytracer.

Además se proporcionan un conjunto de scripts de ejemplo en el directorio `data/`. Algunos de estos requiren que los ejecutes en ese directorio ya que dependen de los archivos que se encuentran alli. Se proporciona un lector simple del formato `OBJ` de Alias/Wavefront para mallas (`readobj.lua`) así como un objeto `OBJ` de ejemplo (`cow.obj`).

Para esta tarea no es necesario implementar una interfaz de usuario. Tu programa podrá ejecutarse desde la línea de comando.

En esta tarea se utliza la función `run_lua` que se encargará de leer la escena y llamar a la función `render()` (en `a4.cpp`) cuando sea necesario. Una vez que esta función se ejecute el programa terminará.

6 Entregables

Ejecutable

Código compilable con un ejecutable `rt` que tomará un archivo de escena como argumento.

Imágenes ejemplo

Deberás generar archivos de imagen como salida en esta tarea. Si decides utilizar el código adjunto podrás generarlas utilizando el script `nonhier.lua` que está en el directorio `data`. Si implementas también el raytracer jerárquico con el código adjunto deberás también probar los scripts:

- `nonhier.lua` – una escena no-jerárquica.
- `macho-cows.lua` – una escena jerárquica con instancias.
- `simple-cows.lua` – una versión simplificada de `macho-cows.lua`.

Si decides utilizar otro lenguaje para describir tus escenas el contenido de cada escena deberá ser equivalente.

Los archivos de imagen deben estar en formato `PNG` o `JPEG`. Los scripts `macho-cows.lua` y `simple-cows.lua` probarán completamente la funcionalidad de tu raytracer (incluyendo el ray tracer jerárquico) excepto por tu función adicional. Si no quieres utilizar estos puedes hacer tus escenas equivalentes.

Además, para mostrar tu implementación de los volumenes englobantes, deberás realizar un rendering especial de ya sea `nonhier.lua` o `macho-cows.lua` donde dibujarás estos volúmenes en lugar de las mallas poligonales. Esta imagen será almacenada en `nonhier-bb.png` o `macho-cows-bb.png`. Los archivos de imagen estarán en el directorio `data/`.

Escena de ejemplo

Debes generar un script de ejemplo llamado `sample.lua` y hacer un render de este en la imagen `sample.png`. El archivo y la imagen estarán en el directorio `data/`. No es necesario hacer un rendering especial de este script para mostrar los volumenes englobantes de las mallas poligonales.

Documentación adicional

Tu archivo `README` deberá incluir la descripción de la funcionalidad adicional de tu raytracer. Si utilizas modelos bajados de internet da crédito al lugar donde los obtuviste.

Deberás entregar al menos un archivo de imagen: `sample.png`. Esta imagen debe tener una resolución de al menos 500×500. Puedes entregar más imágenes pero no olvides mencionarlo en el archivo `README`. Si no funciona tu raytracer jerárquico entrega una imagen generada sin éste y menciona que no funcionan las transformaciones jerárquicas en el archivo `README`.

Objetivos:

Fecha de entrega: lunes 7 de diciembre

Nombre: _____

1. Los objetos son visibles en la imagen. Esto implica que puedes generar rayos primarios que intersecan con esferas y puedes generar imágenes `PNG` como archivo de salida.
2. Se generan adecuadamente imágenes con cubos y mallas poligonales.
3. Los objetos se ordenan adecuadamente de atrás hacia adelante.
4. Hay una función que genera un fondo de la escena sin que oscurezca la vista de ninguno de los objetos en la escena. Este fondo se encuentra en todas las imágenes generadas.
5. El modelo de iluminación de difusión y especular (Phong) funciona correctamente.
6. Las sombras funcionan correctamente.
7. Se proporciona un script que define y dibuja (render) una escena original .
8. Las transformaciones jerárquicas funcionan correctamente. Las esferas y los cubos se pueden transformar con transformaciones afines.
9. Los volúmenes englobantes (esferas o cajas) se han implementado para objetos poligonales como se muestra en las imágenes con rendering especial.

Recuerda que si no pudiste hacer funcionar alguno de los objetivos anteriores puedes escribir en el archivo README por qué no funciona y tus ideas para solucionarlo. Si algún objetivo no está hecho y no hay explicación sobre este no será tomado en cuenta en las tareas.

Recuerda también que el código y la documentación que entregues es personal y deberás citar tus fuentes.