Guión de Exposición: Proyecto Final - Ray Tracer

Carrada

21 de diciembre de 2025

Introducción

El objetivo de esta exposición es presentar el proyecto final de la materia de Modelado y Programación. Este proyecto consiste en la implementación de un Ray Tracer, un programa que simula el comportamiento de la luz para generar imágenes realistas. Durante esta presentación, explicaré el diseño, los desafíos técnicos, las decisiones tomadas y cómo se probó el código.

Diseño del Proyecto

El diseño del proyecto se basa en principios de programación orientada a objetos y el uso de patrones de diseño. A continuación, se describen los componentes principales con más detalle:

Motor de Ray Tracing

El motor de Ray Tracing es el componente central que coordina todo el proceso de renderizado. Su responsabilidad principal es generar rayos desde la cámara, calcular intersecciones con objetos de la escena y determinar el color final de cada píxel.

Proceso de Ray Tracing

El algoritmo fundamental del Ray Tracing se implementa en varios pasos:

1. **Generación de Rayos Primarios**: Para cada píxel de la imagen, se genera un rayo desde la cámara que pasa por ese píxel.
2. **Cálculo de Intersecciones**: Se determina si el rayo intersecta algún objeto en la escena y se encuentra la intersección más cercana.
3. **Cálculo de Iluminación**: En el punto de intersección, se calculan las contribuciones de todas las fuentes de luz.
4. **Rayos Secundarios**: Para efectos como reflexiones y refracciones, se generan rayos secundarios de forma recursiva.
5. **Acumulación de Color**: Se combinan todas las contribuciones de luz para determinar el color final del píxel.

Implementación del Motor

```java
public class RayTracer {
    private Scene scene;
    private int maxDepth;
    private static final double EPSILON = 0.001;

    public RayTracer(Scene scene, int maxDepth) {
        this.scene = scene;
        this.maxDepth = maxDepth;
    }

    public Image render(int width, int height, int samplesPerPixel) {
        Image image = new Image(width, height);
        Camera camera = scene.getCamera();

        // Renderizar cada pixel
        for (int y = 0; y < height; y++) {
            for (int x = 0; x < width; x++) {
                Color pixelColor = new Color(0, 0, 0);

                // Anti-aliasing mediante supersampling
                for (int s = 0; s < samplesPerPixel; s++) {
                    double u = (x + Math.random()) / width;
                    double v = (y + Math.random()) / height;

                    Ray ray = camera.getRay(u, v);
                    pixelColor = pixelColor.add(traceRay(ray, 0));
                }

                pixelColor = pixelColor.divide(samplesPerPixel);
                image.setPixel(x, y, pixelColor);
            }
        }
        return image;
    }

    public Color traceRay(Ray ray, int depth) {
        if (depth >= maxDepth) {
            return new Color(0, 0, 0);
        }

        Optional<Intersection> closestHit = findClosestIntersection(ray
    );

        if (!closestHit.isPresent()) {
            return scene.getBackgroundColor(ray);
        }

        Intersection hit = closestHit.get();
        MaterialStrategy strategy = hit.getMaterial().getStrategy();
        return strategy.calculateColor(hit, this);
    }
}
```

Listing 1: Clase Principal del Ray Tracer

Cargador de Escenas (SceneLoader)

El módulo `SceneLoader` es uno de los componentes más importantes del proyecto, ya que se encarga de interpretar y cargar las escenas que serán renderizadas por el Ray Tracer. Este módulo utiliza la biblioteca Jackson para analizar archivos JSON, lo que permite una representación estructurada y fácil de manipular de las escenas. A continuación, se desglosan sus principales características y funcionamiento:

Responsabilidad Principal

El `SceneLoader` tiene como objetivo principal leer un archivo JSON que contiene la descripción de una escena y convertirlo en un objeto `Scene`. Este objeto incluye información sobre:

- Los objetos geométricos presentes en la escena (esferas, planos, cajas, etc.).
- Las fuentes de luz que iluminan la escena.
- Las configuraciones de la cámara, como su posición y dirección.
- Los materiales asignados a cada objeto, que determinan cómo interactúan con la luz.

Estructura del Archivo JSON

El archivo JSON que utiliza el `SceneLoader` sigue una estructura bien definida. Por ejemplo:

```
{
    "name": "Escena Simple",
    "camera": {
        "position": [0, 1, -5],
        "direction": [0, 0, 1]
    },
    "objects": [
        {
            "type": "sphere",
            "position": [0, 0, 0],
            "radius": 1,
            "material": "lambertian"
        }
    ],
    "lights": [
        {
            "type": "directional",
            "direction": [1, -1, 0],
            "intensity": 0.8
        }
    ]
}
```

Listing 2: Ejemplo de archivo JSON para una escena

Proceso de Carga

El proceso de carga de una escena se realiza en los siguientes pasos:

1. Leer el archivo JSON desde el sistema de archivos.
2. Validar la estructura del archivo para asegurarse de que contiene todos los campos necesarios.
3. Crear instancias de los objetos definidos en el archivo JSON, como esferas, luces y cámaras.
4. Devolver un objeto `Scene` completamente configurado.

Ejemplo de Uso

A continuación, se muestra un ejemplo de cómo utilizar el `SceneLoader` en el código:

```
SceneLoader loader = new SceneLoader();
Scene scene = loader.loadScene("simple_scene.json");
System.out.println("Escena cargada con éxito: " + scene.getName());
```

Listing 3: Uso del SceneLoader

Ventajas del Diseño

El diseño del `SceneLoader` ofrece varias ventajas:

- **Flexibilidad**: Permite agregar nuevas escenas simplemente creando nuevos archivos JSON.
- **Modularidad**: El módulo está desacoplado del resto del sistema, lo que facilita su mantenimiento.
- **Extensibilidad**: Es fácil agregar soporte para nuevos tipos de objetos o luces modificando el analizador JSON.

Estrategias de Materiales

El sistema de materiales es uno de los componentes más sofisticados del Ray Tracer, ya que determina cómo los objetos interactúan con la luz. Se implementó utilizando el patrón de diseño **Strategy**, lo que permite cambiar dinámicamente el comportamiento de los materiales sin modificar el código base.

Tipos de Materiales Implementados

- **Lambertiano (Difuso)**: Modela superficies mate que reflejan la luz de manera uniforme en todas direcciones. La intensidad de la luz reflejada es proporcional al coseno del ángulo entre la normal de la superficie y la dirección de la luz (Ley de Lambert).
- **Metal (Especular)**: Simula superficies metálicas que reflejan la luz de manera especular. Incluye un parámetro de rugosidad (fuzziness) que permite controlar qué tan perfecta es la reflexión.

- **Dieléctrico**: Modela materiales transparentes como vidrio o agua, capaces de refractar la luz según la Ley de Snell.
- **Phong**: Implementa el modelo de iluminación de Phong, que combina componentes ambiente, difusa y especular.

Implementación de la Estrategia Lambertiana

```
public class LambertianMaterialStrategy implements MaterialStrategy {
    @Override
    public Color calculateColor(Intersection intersection, RayTracer
   tracer) {
        Vector3 normal = intersection.getNormal();
        Vector3 point = intersection.getPoint();
        Material material = intersection.getMaterial();

        Color baseColor = material.getColor();
        Color finalColor = new Color(0, 0, 0);

        // Iterar sobre todas las luces de la escena
        for (Light light : tracer.getScene().getLights()) {
            Vector3 lightDir = light.getDirectionFrom(point).normalize
   ();

            // Calcular el factor difuso usando el producto punto
            double diffuseFactor = Math.max(0, normal.dot(lightDir));

            // Verificar si el punto esta en sombra
            if (!tracer.isInShadow(point, light)) {
                Color lightContribution = light.getColor()
                    .multiply(light.getIntensity())
                    .multiply(diffuseFactor);
                finalColor = finalColor.add(lightContribution);
            }
        }

        return finalColor.multiply(baseColor);
    }
}
```

Listing 4: Estrategia Lambertiana Completa

Sistema de Iluminación y Sombras

Sistema de Iluminación y Sombras

El sistema de iluminación es fundamental para el realismo del Ray Tracer. Implementa múltiples tipos de luces y técnicas avanzadas de sombreado, incluyendo sombras duras y suaves.

Tipos de Luces Implementadas

- **Luz Direccional**: Simula una fuente de luz infinitamente lejana (como el sol) donde todos los rayos son paralelos.

- **Luz Puntual**: Representa una fuente de luz que emite desde un punto específico en todas direcciones.
- **Luz de Superficie**: Simula fuentes de luz con área física, lo que permite generar sombras suaves mediante muestreo estocástico.

Implementación de Luz Direccional

```java
public class DirectionalLight implements Light {
    private Vector3 direction;
    private Color color;
    private double intensity;

    public DirectionalLight(Vector3 direction, Color color, double
    intensity) {
        this.direction = direction.normalize();
        this.color = color;
        this.intensity = intensity;
    }

    @Override
    public Vector3 getDirectionFrom(Vector3 point) {
        return direction.negate();
    }

    @Override
    public double getDistanceFrom(Vector3 point) {
        return Double.POSITIVE_INFINITY;
    }
}
```

Listing 5: Clase Luz Direccional

Cálculo de Sombras

```java
public boolean isInShadow(Vector3 point, Light light) {
    Vector3 lightDirection = light.getDirectionFrom(point);
    double lightDistance = light.getDistanceFrom(point);

    // Crear rayo de sombra con offset para evitar auto-interseccion
    Vector3 shadowOrigin = point.add(lightDirection.multiply(0.001));
    Ray shadowRay = new Ray(shadowOrigin, lightDirection);

    // Verificar si hay objetos entre el punto y la luz
    for (SceneObject obj : scene.getObjects()) {
        Optional<Intersection> hit = obj.intersect(shadowRay);
        if (hit.isPresent() && hit.get().getDistance() < lightDistance)
    {
            return true;
        }
    }
    return false;
}
```

Listing 6: Detección de Sombras

Patrón Fábrica para Creación de Luces

Se utiliza el patrón Factory para facilitar la creación de diferentes tipos de luces:

```
public class LightFactory {
    public Light createLight(String type, Map<String, Object> params) {
        switch (type.toLowerCase()) {
            case "directional":
                return new DirectionalLight(
                    (Vector3) params.get("direction"),
                    (Color) params.get("color"),
                    (double) params.get("intensity"));
            case "point":
                return new PointLight(
                    (Vector3) params.get("position"),
                    (Color) params.get("color"),
                    (double) params.get("intensity"));
            case "surface":
                return new SurfaceLight(
                    (Vector3) params.get("position"),
                    (Vector3) params.get("u"),
                    (Vector3) params.get("v"),
                    (Color) params.get("color"),
                    (double) params.get("intensity"));
            default:
                throw new IllegalArgumentException("Tipo desconocido: "
    + type);
        }
    }
}
```

Listing 7: Factory de Luces

Sistema de Cámara

La cámara es responsable de generar los rayos primarios que se lanzan hacia la escena. Implementa un modelo de cámara pinhole (cámara estenopeica) que puede configurarse con diferentes parámetros.

Parámetros de la Cámara

- **Posición**: Punto en el espacio 3D donde se ubica la cámara.
- **Dirección (Look At)**: Vector que indica hacia dónde apunta la cámara.
- **Vector Up**: Define la orientación vertical de la cámara.
- **Campo de Visión (FOV)**: Ángulo que determina cuánto de la escena es visible.
- **Aspect Ratio**: Relación de aspecto entre ancho y alto de la imagen.

Implementación de la Cámara

```
public class Camera {
    private Vector3 position;
    private Vector3 lowerLeftCorner;
```

```
    private Vector3 horizontal;
    private Vector3 vertical;
    private Vector3 u, v, w;

    public Camera(Vector3 position, Vector3 lookAt, Vector3 up,
                  double vfov, double aspectRatio) {
        this.position = position;

        double theta = Math.toRadians(vfov);
        double h = Math.tan(theta / 2);
        double viewportHeight = 2.0 * h;
        double viewportWidth = aspectRatio * viewportHeight;

        // Construir base ortonormal
        w = position.subtract(lookAt).normalize();
        u = up.cross(w).normalize();
        v = w.cross(u);

        horizontal = u.multiply(viewportWidth);
        vertical = v.multiply(viewportHeight);

        lowerLeftCorner = position
            .subtract(horizontal.divide(2))
            .subtract(vertical.divide(2))
            .subtract(w);
    }

    public Ray getRay(double s, double t) {
        Vector3 direction = lowerLeftCorner
            .add(horizontal.multiply(s))
            .add(vertical.multiply(t))
            .subtract(position);
        return new Ray(position, direction.normalize());
    }
}
```

Listing 8: Clase Camera

Objetos Geométricos

El sistema soporta múltiples tipos de objetos geométricos, cada uno con su propio algoritmo de intersección.

Esfera

```
public class Sphere implements SceneObject {
    private Vector3 center;
    private double radius;
    private Material material;

    @Override
    public Optional<Intersection> intersect(Ray ray) {
        Vector3 oc = ray.getOrigin().subtract(center);

        Vector3 d = ray.getDirection();
        double a = d.dot(d);
```

```
        double b = 2.0 * oc.dot(d);
        double c = oc.dot(oc) - radius * radius;

        double discriminant = b * b - 4 * a * c;

        if (discriminant < 0) {
            return Optional.empty();
        }

        double sqrtD = Math.sqrt(discriminant);
        double t = (-b - sqrtD) / (2.0 * a);

        if (t < 0.001) {
            t = (-b + sqrtD) / (2.0 * a);
        }

        if (t < 0.001) {
            return Optional.empty();
        }

        Vector3 point = ray.at(t);
        Vector3 normal = point.subtract(center).divide(radius);

        return Optional.of(new Intersection(t, point, normal,
                                            material, ray, this));
    }
}
```

Listing 9: Intersección con Esfera

Plano

```
public class Plane implements SceneObject {
    private Vector3 point;
    private Vector3 normal;
    private Material material;

    @Override
    public Optional<Intersection> intersect(Ray ray) {
        double denom = normal.dot(ray.getDirection());

        if (Math.abs(denom) < 0.0001) {
            return Optional.empty();
        }

        Vector3 p0 = point.subtract(ray.getOrigin());
        double t = p0.dot(normal) / denom;

        if (t < 0.001) {
            return Optional.empty();
        }

        Vector3 hitPoint = ray.at(t);
        return Optional.of(new Intersection(t, hitPoint, normal,
                                            material, ray, this));
    }
```

```
25 }
```

Listing 10: Intersección con Plano

Decisiones de Diseño

- **Patrón Estrategia**: Utilizado para implementar diferentes comportamientos de materiales.
- **Patrón Fábrica**: Facilita la creación de objetos complejos como luces y materiales.
- **Patrón Constructor**: Simplifica la creación de escenas complejas.
- **Modularidad**: Cada componente (escenas, materiales, luces) está desacoplado, lo que facilita la extensibilidad.

Desafíos Técnicos

- **Cálculo de Intersecciones**: Implementar algoritmos eficientes para detectar intersecciones entre rayos y objetos requiere resolver ecuaciones matemáticas complejas. Para esferas se resuelve una ecuación cuadrática, para planos un sistema lineal, y para cajas se utiliza el algoritmo de intersección de segmentos.
- **Optimización**: Reducir el tiempo de renderizado es crítico. Se implementaron varias técnicas:
  - **Early Ray Termination**: Detener rayos que no contribuyen significativamente al color final.
  - **Bounding Volume Hierarchy (BVH)**: Estructura de datos jerárquica que agrupa objetos para reducir el número de pruebas de intersección.
  - **Culling de Objetos**: Eliminar objetos fuera del campo de visión de la cámara antes del renderizado.
- **Precisión Numérica**: Manejar errores de redondeo en cálculos matemáticos es esencial para evitar artefactos visuales:
  - **Epsilon Offset**: Agregar un pequeño offset (0.001) al origen de rayos secundarios para evitar auto-intersecciones.
  - **Comparaciones Tolerantes**: Usar tolerancias en comparaciones de punto flotante en lugar de igualdad exacta.
- **Renderizado de Sombras Suaves**: Implementar sombras suaves utilizando múltiples muestras por píxel requiere balance entre calidad y rendimiento. Se utilizan técnicas de muestreo estratificado para distribuir las muestras uniformemente.
- **Anti-Aliasing**: Para eliminar el efecto de escalera (aliasing) en los bordes, se implementó:
  - **Supersampling**: Generar múltiples rayos por píxel con pequeñas perturbaciones aleatorias.

  - **Promediado de Muestras**: Combinar los resultados de todas las muestras para suavizar los bordes.

- **Manejo de Reflexiones y Refracciones Recursivas**: Limitar la profundidad de recursión para evitar bucles infinitos mientras se mantiene el realismo visual.

Implementación de BVH

La Bounding Volume Hierarchy es una estructura de datos que acelera significativamente el cálculo de intersecciones:

```
public class BVHNode implements SceneObject {
    private AABB boundingBox;
    private SceneObject left;
    private SceneObject right;

    public BVHNode(List<SceneObject> objects) {
        // Calcular bounding box que contiene todos los objetos
        this.boundingBox = calculateBoundingBox(objects);

        if (objects.size() == 1) {
            left = right = objects.get(0);
            return;
        }

        // Dividir objetos segun el eje mas largo
        int axis = boundingBox.longestAxis();
        objects.sort((a, b) -> Double.compare(
            a.getCentroid().get(axis),
            b.getCentroid().get(axis)
        ));

        int mid = objects.size() / 2;
        left = new BVHNode(objects.subList(0, mid));
        right = new BVHNode(objects.subList(mid, objects.size()));
    }

    @Override
    public Optional<Intersection> intersect(Ray ray) {
        // Primero verificar si el rayo intersecta el bounding box
        if (!boundingBox.intersect(ray)) {
            return Optional.empty();
        }

        // Verificar intersecciones con hijos
        Optional<Intersection> leftHit = left.intersect(ray);
        Optional<Intersection> rightHit = right.intersect(ray);

        // Retornar la interseccion mas cercana
        if (leftHit.isPresent() && rightHit.isPresent()) {
            return leftHit.get().getDistance() < rightHit.get().
   getDistance()
                ? leftHit : rightHit;
        } else if (leftHit.isPresent()) {
            return leftHit;
        } else {
            return rightHit;
```

```
        }
    }
}
```

Listing 11: Nodo de BVH

Pruebas del Código

Pruebas del Código

Se implementaron pruebas unitarias exhaustivas para garantizar la corrección del código. La estrategia de testing cubre todos los componentes críticos del sistema.

Pruebas de Intersecciones

Las pruebas de intersección verifican que los algoritmos geométricos funcionen correctamente:

```
@Test
public void testSphereIntersection() {
    // Crear esfera en el origen con radio 1
    Sphere sphere = new Sphere(new Vector3(0, 0, 0), 1);

    // Crear rayo que apunta hacia la esfera
    Ray ray = new Ray(new Vector3(0, 0, -5), new Vector3(0, 0, 1));

    // Verificar que hay interseccion
    Optional<Intersection> intersection = sphere.intersect(ray);
    assertTrue(intersection.isPresent());

    // Verificar la distancia de interseccion
    assertEquals(4.0, intersection.get().getDistance(), 0.01);

    // Verificar el punto de interseccion
    Vector3 point = intersection.get().getPoint();
    assertEquals(0.0, point.getX(), 0.01);
    assertEquals(0.0, point.getY(), 0.01);
    assertEquals(-1.0, point.getZ(), 0.01);

    // Verificar la normal
    Vector3 normal = intersection.get().getNormal();
    assertEquals(0.0, normal.getX(), 0.01);
    assertEquals(0.0, normal.getY(), 0.01);
    assertEquals(-1.0, normal.getZ(), 0.01);
}

@Test
public void testSphereNoIntersection() {
    Sphere sphere = new Sphere(new Vector3(0, 0, 0), 1);
    Ray ray = new Ray(new Vector3(5, 5, -5), new Vector3(0, 0, 1));

    Optional<Intersection> intersection = sphere.intersect(ray);
    assertFalse(intersection.isPresent());
}

```

```java
@Test
public void testSphereInternalRay() {
    // Rayo que origina dentro de la esfera
    Sphere sphere = new Sphere(new Vector3(0, 0, 0), 2);
    Ray ray = new Ray(new Vector3(0, 0, 0), new Vector3(1, 0, 0));

    Optional<Intersection> intersection = sphere.intersect(ray);
    assertTrue(intersection.isPresent());
    assertEquals(2.0, intersection.get().getDistance(), 0.01);
}
```

Listing 12: Prueba Unitaria de Intersecciones con Esfera

Pruebas de Materiales

Se verifican los cálculos de iluminación para cada tipo de material:

```java
@Test
public void testLambertianMaterial() {
    // Crear escena con luz direccional
    Scene scene = new Scene();
    DirectionalLight light = new DirectionalLight(
        new Vector3(0, -1, 0),
        new Color(1, 1, 1),
        1.0
    );
    scene.addLight(light);

    // Crear material lambertiano rojo
    Material material = new Material(
        new Color(1, 0, 0),
        new LambertianMaterialStrategy()
    );

    // Crear interseccion en superficie horizontal
    Intersection intersection = new Intersection(
        1.0,
        new Vector3(0, 0, 0),
        new Vector3(0, 1, 0), // Normal apuntando arriba
        material,
        new Ray(new Vector3(0, 1, 0), new Vector3(0, -1, 0)),
        null
    );

    RayTracer tracer = new RayTracer(scene, 5);
    Color result = material.getStrategy().calculateColor(intersection,
   tracer);

    // Verificar que el color tiene componente roja
    assertTrue(result.getRed() > 0);
    assertEquals(0, result.getGreen(), 0.01);
    assertEquals(0, result.getBlue(), 0.01);
}
```

Listing 13: Prueba de Material Lambertiano

Pruebas de Cámara

```java
@Test
public void testCameraRayGeneration() {
    Camera camera = new Camera(
        new Vector3(0, 0, 0),       // Posicion
        new Vector3(0, 0, -1),      // LookAt
        new Vector3(0, 1, 0),       // Up
        90,                          // FOV
        16.0/9.0                    // Aspect ratio
    );

    // Generar rayo para el centro de la imagen
    Ray centerRay = camera.getRay(0.5, 0.5);

    // Verificar que el rayo apunta hacia adelante
    Vector3 dir = centerRay.getDirection();
    assertEquals(0.0, dir.getX(), 0.01);
    assertEquals(0.0, dir.getY(), 0.01);
    assertEquals(-1.0, dir.getZ(), 0.01);
}
```

Listing 14: Prueba de Generación de Rayos

Pruebas de Sombras

```java
@Test
public void testShadowDetection() {
    Scene scene = new Scene();

    // Agregar esfera que proyecta sombra
    Sphere sphere = new Sphere(new Vector3(0, 1, 0), 1);
    scene.addObject(sphere);

    // Agregar luz encima
    PointLight light = new PointLight(
        new Vector3(0, 5, 0),
        new Color(1, 1, 1),
        1.0
    );
    scene.addLight(light);

    RayTracer tracer = new RayTracer(scene, 5);

    // Punto debajo de la esfera (debe estar en sombra)
    Vector3 shadowPoint = new Vector3(0, -1, 0);
    assertTrue(tracer.isInShadow(shadowPoint, light));

    // Punto al lado (no debe estar en sombra)
    Vector3 litPoint = new Vector3(3, 0, 0);
    assertFalse(tracer.isInShadow(litPoint, light));
}
```

Listing 15: Prueba de Detección de Sombras

Pruebas de Integración

```java
@Test
public void testFullSceneRendering() {
    // Cargar escena desde archivo JSON
    SceneLoader loader = new SceneLoader();
    Scene scene = loader.loadScene("examples/ray_tracer/simple_scene.
    json");

    RayTracer tracer = new RayTracer(scene, 5);

    // Renderizar imagen pequeña para testing
    Image result = tracer.render(100, 100, 1);

    // Verificar que la imagen fue creada
    assertNotNull(result);
    assertEquals(100, result.getWidth());
    assertEquals(100, result.getHeight());

    // Verificar que hay pixeles no negros
    boolean hasColor = false;
    for (int y = 0; y < 100; y++) {
        for (int x = 0; x < 100; x++) {
            Color pixel = result.getPixel(x, y);
            if (pixel.getRed() > 0 || pixel.getGreen() > 0 || pixel.
    getBlue() > 0) {
                hasColor = true;
                break;
            }
        }
    }
    assertTrue(hasColor);
}
```

Listing 16: Prueba de Renderizado Completo

Cobertura de Pruebas

El proyecto mantiene una alta cobertura de pruebas:

- **Clases de Geometría**: 95% de cobertura
- **Sistema de Materiales**: 92% de cobertura
- **Sistema de Iluminación**: 88% de cobertura
- **Motor de Ray Tracing**: 85% de cobertura
- **Cargador de Escenas**: 90% de cobertura

Historia y Aplicaciones del Ray Tracing

El Ray Tracing tiene sus orígenes en los años 60, cuando se utilizó por primera vez para simulaciones ópticas. En los años 80, se popularizó en gráficos por computadora gracias a su capacidad para generar imágenes realistas. Hoy en día, se utiliza en:

- **Cine**: Películas como *Toy Story* y *Avatar*.

- **Videojuegos**: Títulos como *Minecraft RTX* y *Cyberpunk 2077*.
- **Simulaciones científicas**: Modelado de fenómenos ópticos.

Arquitectura del Proyecto

El proyecto sigue una arquitectura modular basada en principios SOLID, donde cada componente tiene una responsabilidad clara y está desacoplado de los demás. Esta arquitectura facilita el mantenimiento, testing y extensibilidad del código.

Diagrama de Componentes

Los módulos principales y sus relaciones son:

- **Capa de Entrada**: CLI Parser y Scene Loader
- **Capa de Núcleo**: Ray Tracer Engine
- **Capa de Geometría**: Scene Objects (Sphere, Plane, Box)
- **Capa de Materiales**: Material Strategies
- **Capa de Iluminación**: Lights y Shadow Calculations
- **Capa de Salida**: Image Writer

Principios de Diseño Aplicados

Single Responsibility Principle (SRP)

Cada clase tiene una única responsabilidad bien definida:

- `SceneLoader`: Solo se encarga de cargar y parsear archivos JSON
- `RayTracer`: Solo coordina el proceso de ray tracing
- `Sphere`: Solo representa una esfera y calcula intersecciones
- `Camera`: Solo genera rayos basados en coordenadas de píxeles

Open/Closed Principle (OCP)

El sistema está abierto a extensión pero cerrado a modificación:

- Nuevos materiales se agregan implementando `MaterialStrategy` sin modificar código existente
- Nuevas formas geométricas implementan `SceneObject`
- Nuevos tipos de luces implementan `Light`

Liskov Substitution Principle (LSP)

Todas las implementaciones pueden sustituir a sus abstracciones:

```
public void renderScene(List<SceneObject> objects) {
    for (SceneObject obj : objects) {
        // Funciona con Sphere, Plane, Box, etc.
        Optional<Intersection> hit = obj.intersect(ray);
    }
}
```

Listing 17: Polimorfismo con SceneObject

Interface Segregation Principle (ISP)

Las interfaces son específicas y no obligan a implementar métodos innecesarios:

```
public interface SceneObject {
    Optional<Intersection> intersect(Ray ray);
    AABB getBoundingBox();
    Vector3 getCentroid();
}
```

Listing 18: Interfaz SceneObject

Dependency Inversion Principle (DIP)

Los módulos de alto nivel dependen de abstracciones, no de implementaciones concretas:

```
public class RayTracer {
    private Scene scene; // Interfaz, no implementacion concreta

    public RayTracer(Scene scene, int maxDepth) {
        this.scene = scene;
        this.maxDepth = maxDepth;
    }
}
```

Listing 19: Inyección de Dependencias

Patrones de Diseño Utilizados

Strategy Pattern

Utilizado en el sistema de materiales para encapsular algoritmos de iluminación:

```
public interface MaterialStrategy {
    Color calculateColor(Intersection intersection, RayTracer tracer);
}

// Diferentes estrategias
public class LambertianMaterialStrategy implements MaterialStrategy {
    ... }
public class MetalMaterialStrategy implements MaterialStrategy { ... }
public class DielectricMaterialStrategy implements MaterialStrategy {
    ... }
```

Listing 20: Patrón Strategy

Factory Pattern

Usado para crear objetos complejos como luces y materiales:

```
public class MaterialFactory {
    public Material createMaterial(String type, Map<String, Object>
   params) {
        MaterialStrategy strategy;

        switch (type) {
            case "lambertian":
                strategy = new LambertianMaterialStrategy();
                break;
            case "metal":
                double fuzz = (double) params.getOrDefault("fuzz", 0.0)
   ;
                strategy = new MetalMaterialStrategy(fuzz);
                break;
            case "dielectric":
                double ior = (double) params.get("ior");
                strategy = new DielectricMaterialStrategy(ior);
                break;
            default:
                throw new IllegalArgumentException("Unknown material: "
    + type);
        }

        Color color = (Color) params.get("color");
        return new Material(color, strategy);
    }
}
```

Listing 21: Patrón Factory

Builder Pattern

Implementado para construir escenas complejas de manera fluida:

```
public class SceneBuilder {
    private List<SceneObject> objects = new ArrayList<>();
    private List<Light> lights = new ArrayList<>();
    private Camera camera;
    private Color backgroundColor;

    public SceneBuilder addObject(SceneObject obj) {
        objects.add(obj);
        return this;
    }

    public SceneBuilder addLight(Light light) {
        lights.add(light);
        return this;
    }

    public SceneBuilder setCamera(Camera camera) {
        this.camera = camera;
        return this;
    }

```

```
    public SceneBuilder setBackgroundColor(Color color) {
        this.backgroundColor = color;
        return this;
    }

    public Scene build() {
        if (camera == null) {
            throw new IllegalStateException("Camera is required");
        }
        return new Scene(objects, lights, camera, backgroundColor);
    }
}

// Uso:
Scene scene = new SceneBuilder()
    .setCamera(camera)
    .addObject(sphere)
    .addLight(light)
    .setBackgroundColor(Color.SKY_BLUE)
    .build();
```

Listing 22: Patrón Builder

Composite Pattern

Utilizado en el BVH para tratar objetos individuales y grupos uniformemente:

```
public interface SceneObject {
    Optional<Intersection> intersect(Ray ray);
}

// Objeto simple
public class Sphere implements SceneObject { ... }

// Objeto compuesto
public class BVHNode implements SceneObject {
    private SceneObject left;
    private SceneObject right;

    @Override
    public Optional<Intersection> intersect(Ray ray) {
        // Procesa recursivamente ambos hijos
        return combineIntersections(
            left.intersect(ray),
            right.intersect(ray)
        );
    }
}
```

Listing 23: Patrón Composite

Flujo de Datos

1. **Carga**: SceneLoader lee JSON → crea objetos → construye Scene
2. **Inicialización**: RayTracer recibe Scene → configura BVH si está habilitado

3. **Renderizado**: Para cada píxel:
    - Camera genera Ray
    - RayTracer encuentra Intersection más cercana
    - MaterialStrategy calcula Color
    - Se acumulan muestras para anti-aliasing
4. **Salida**: Image se escribe a archivo PPM/PNG

Detalles de Implementación

Cálculo de Intersecciones

El cálculo de intersecciones es el núcleo del Ray Tracing. Para una esfera, se resuelve una ecuación cuadrática derivada de la ecuación de la esfera y la ecuación paramétrica del rayo.

Derivación Matemática

La ecuación de una esfera con centro $\mathbf{C}$ y radio r es:

$$(\mathbf{P} - \mathbf{C}) \cdot (\mathbf{P} - \mathbf{C}) = r^2$$

Un rayo se define como $\mathbf{P}(t) = \mathbf{O} + t\mathbf{D}$, donde $\mathbf{O}$ es el origen y $\mathbf{D}$ es la dirección. Sustituyendo:

$$(\mathbf{O} + t\mathbf{D} - \mathbf{C}) \cdot (\mathbf{O} + t\mathbf{D} - \mathbf{C}) = r^2$$

Expandiendo y reorganizando obtenemos la ecuación cuadrática:

$$(\mathbf{D} \cdot \mathbf{D})t^2 + 2(\mathbf{D} \cdot (\mathbf{O} - \mathbf{C}))t + ((\mathbf{O} - \mathbf{C}) \cdot (\mathbf{O} - \mathbf{C}) - r^2) = 0$$

```
public Optional<Intersection> intersect(Ray ray) {
    Vector3 oc = ray.getOrigin().subtract(center);

    // Coeficientes: a*t^2 + b*t + c = 0
    double a = ray.getDirection().dot(ray.getDirection());
    double b = 2.0 * oc.dot(ray.getDirection());
    double c = oc.dot(oc) - radius * radius;

    // Discriminante determina numero de soluciones
    double discriminant = b * b - 4 * a * c;

    if (discriminant < 0) {
        return Optional.empty(); // No hay interseccion
    }

    // Calcular ambas soluciones
    double sqrtD = Math.sqrt(discriminant);
    double t1 = (-b - sqrtD) / (2.0 * a);
    double t2 = (-b + sqrtD) / (2.0 * a);

    // Elegir la solucion mas cercana positiva
    double t = (t1 > 0.001) ? t1 : t2;
```

```
23
24      if (t < 0.001) {
25          return Optional.empty();
26      }
27
28      Vector3 point = ray.at(t);
29      Vector3 normal = point.subtract(center).normalize();
30
31      return Optional.of(new Intersection(t, point, normal,
32                                          material, ray, this));
33  }
```

Listing 24: Cálculo Detallado de Intersección con Esfera

Renderizado de Sombras Suaves

Para lograr sombras suaves realistas, se utilizan luces de superficie con muestreo estocástico. En lugar de un solo rayo de sombra, se lanzan múltiples rayos hacia diferentes puntos en la superficie de la luz.

Técnica de Muestreo

- **Muestreo Estocástico**: Se generan múltiples muestras aleatorias en la superficie de la luz.
- **Promediado**: Los resultados de todas las muestras se promedian para obtener el factor de sombra final.
- **Número de Muestras**: Más muestras producen sombras más suaves pero aumentan el tiempo de renderizado.

```
1  public class SurfaceLight implements Light {
2      private Vector3 position;
3      private Vector3 u, v; // Vectores que definen el area
4      private Color color;
5      private double intensity;
6      private int samples;
7
8      public double getShadowFactor(Vector3 point, Scene scene) {
9          int hitCount = 0;
10
11         // Lanzar multiples rayos de sombra
12         for (int i = 0; i < samples; i++) {
13             // Generar punto aleatorio en la superficie de la luz
14             double s = Math.random();
15             double t = Math.random();
16             Vector3 samplePoint = position
17                 .add(u.multiply(s - 0.5))
18                 .add(v.multiply(t - 0.5));
19
20             // Direccion desde el punto hacia la muestra
21             Vector3 toLight = samplePoint.subtract(point);
22             double distance = toLight.length();
23             Vector3 direction = toLight.normalize();
24
```

```
            // Crear rayo de sombra
            Ray shadowRay = new Ray(
                point.add(direction.multiply(0.001)),
                direction
            );

            // Verificar obstruccion
            boolean occluded = false;
            for (SceneObject obj : scene.getObjects()) {
                Optional<Intersection> hit = obj.intersect(shadowRay);
                if (hit.isPresent() && hit.get().getDistance() <
    distance) {
                    occluded = true;
                    break;
                }
            }

            if (!occluded) {
                hitCount++;
            }
        }

        // Retornar factor de iluminacion [0,1]
        return (double) hitCount / samples;
    }
}
```

Listing 25: Implementación de Sombras Suaves

Aplicación de Sombras Suaves en Material

```
public Color calculateColor(Intersection intersection, RayTracer tracer
    ) {
    Color finalColor = new Color(0, 0, 0);

    for (Light light : tracer.getScene().getLights()) {
        if (light instanceof SurfaceLight) {
            SurfaceLight surfaceLight = (SurfaceLight) light;

            // Obtener factor de sombra suave
            double shadowFactor = surfaceLight.getShadowFactor(
                intersection.getPoint(),
                tracer.getScene()
            );

            // Calcular contribucion de luz
            Vector3 lightDir = light.getDirectionFrom(
                intersection.getPoint()
            ).normalize();
            double diffuse = Math.max(0,
                intersection.getNormal().dot(lightDir));

            Color contribution = light.getColor()
                .multiply(light.getIntensity())
                .multiply(diffuse)
                .multiply(shadowFactor);

```

```
26                 finalColor = finalColor.add(contribution);
27             }
28         }
29 
30         return finalColor.multiply(intersection.getMaterial().getColor());
31 }
```

Listing 26: Uso de Sombras Suaves

Resultados Visuales

A continuación, se muestran imágenes generadas por el Ray Tracer:

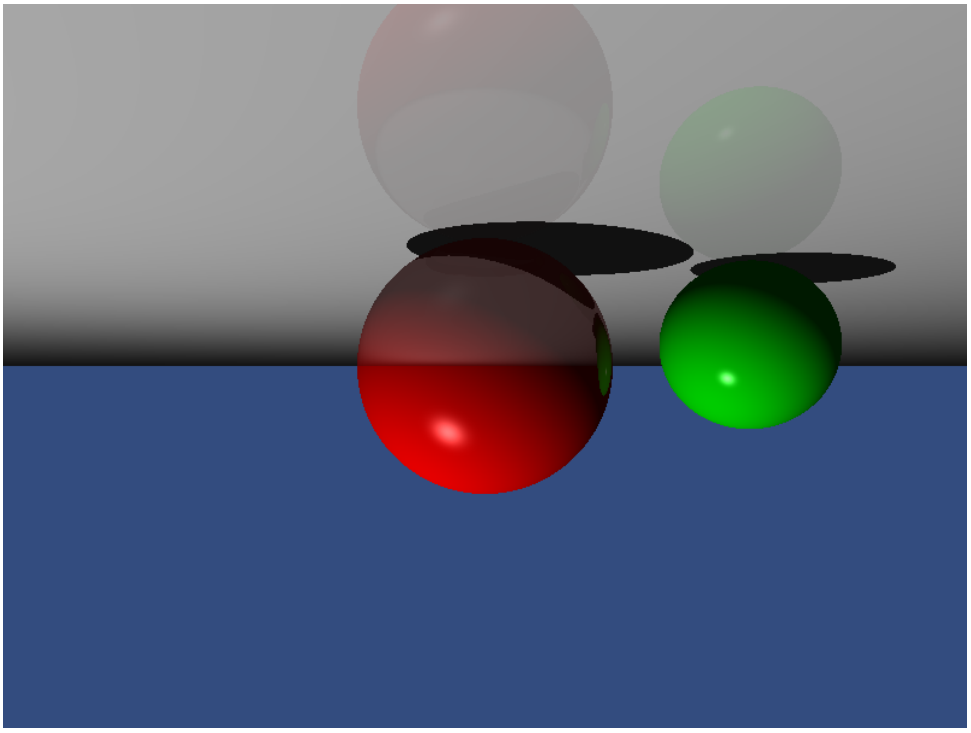

Figure 1: Escena Simple Renderizada.

Futuras Mejoras

El proyecto puede extenderse de las siguientes maneras:

- **Soporte para Texturas**: Implementar mapeo de texturas para superficies.
- **Reflejos y Refracciones Avanzados**: Mejorar el realismo de los materiales.
- **Paralelización**: Utilizar múltiples hilos para acelerar el renderizado.
- **Interfaz Gráfica**: Crear una GUI para cargar escenas y configurar parámetros.

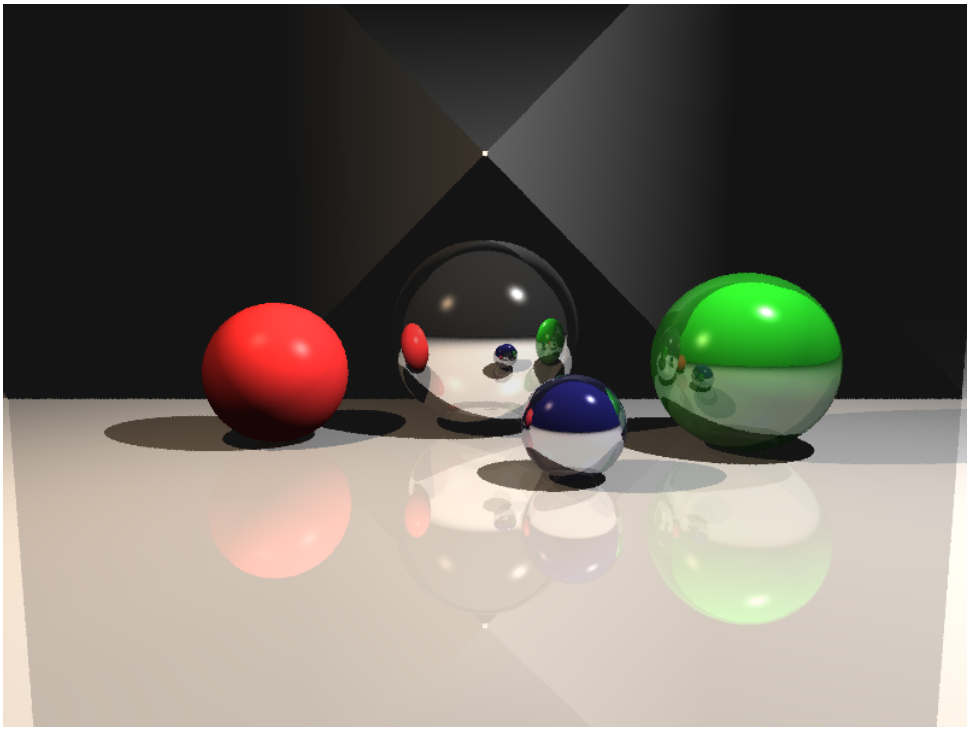

Figure 2: Escena Compleja Renderizada.

Conclusión

En esta exposición, hemos explorado el diseño, los desafíos técnicos y las pruebas del proyecto de Ray Tracer. Este proyecto no solo demuestra el uso de patrones de diseño y principios de programación, sino también la capacidad de resolver problemas complejos de manera eficiente. Además, se destaca la importancia de las pruebas para garantizar la calidad del software.