Actividad recuperable de evaluación: trazador de rayos básico (30 puntos sobre 90)

Plazo para entrega: hasta las 8:00h. del lunes 10 de diciembre de 2018

Procedimiento de entrega

- Fichero comprimido upnatray0.zip. Ese fichero debe contener la carpeta upnatray0 con los siguientes elementos:
  - Proyecto *NetBeans* con el nombre *UPNaTRay*; debe estar limpio de clases compiladas
  - Fichero *informe.pdf* que sirva como breve memoria del trabajo realizado. Debe explicar qué problemas se han encontrado en la realización de la actividad y qué solución se les ha dado. También debe incluir una estimación del número de horas dedicadas a completar la actividad. La extensión del informe no ha de ser superior a tres páginas.
- Sólo se podrá hacer una entrega.

Objetivo de aprendizaje

- Se quiere integrar en un único ejercicio buena parte de los contenidos comentados en las clases de teoría de modo que adquieran sentido para quien complete la actividad.
- El plagio y la copia directa desvirtúan ese objetivo por lo que conllevarán un suspenso irrevocable en la evaluación de la actividad.

Objetivos de realización

- Esta actividad consiste en la implementación de un trazador de rayos muy elemental que únicamente proporcione la posibilidad de usar objetos de geometría simple y de color uniforme en el modelado de las escenas.
- Como continuación de la actividad, posteriormente se ampliará el trazador para que sintetrice imágenes a partir de escenas iluminadas.

- Se sugiere hacer uso de *assert* para comprobar precondiciones, postcondiciones y, en general, cualquier condición (de entrada, intermedia, o de salida) que sea de cumplimiento obligado.

Clase *Image*

- Su definición se basa en la clase *BufferedImage* de *Java*.
- Debe hacer uso de un modelo de color que incluya canal α (transparencia) y que esté basado en el espacio de color *sRGB*.
- Ha de disponer de un constructor al que se le pasa la resolución espacial de la imagen *raster* a generar así como el color de fondo a emplear.
- Ha de disponer también del método

```
public synthesis (final Group3D scene, final Camera camera);
```

  que sirve para generar una imagen a partir de la vista de la escena obtenida desde la cámara aplicando la proyección asociada a ésta.
- Ese método realiza una iteración por los *pixels* que componen la imagen para obtener su color.
- Por cada *pixel* a colorear:
  - Genera un rayo usando la cámara y la proyección proporcionadas.
  - Calcula la intersección con el grupo de elementos de la escena.
  - Calcula el valor de color del *pixel* en base al resultado de la intersección y a la iluminación aplicada sobre la escena.
- Cuando el rayo enviado a través del *pixel* a colorear no intersecta con ningún objeto de la escena, sobre ese *pixel* se aplica el color establecido como color de fondo.

Clases propias del trazador de rayos

Clase *tracer.Ray*

- Sirve para definir objetos que actúen como rayos.
- Ha de proporcionar los constructores:

```
public Ray (final Pointd3D R, final Vector3D v);
public Ray (final Pointd3D R, final Point3D Q);
```

  El primero define un rayo que parte del punto R y sigue la dirección del vector v. El segundo define un rayo que parte del punto R y pasa por el punto Q.

- Ha de proporcionar el método

```
public Point3D pointAtParameter (final float t);
```

  que devuelve el punto del espacio que el rayo alcanza cuando el valor del parámetro t es el pasado como argumento.

- Ha de proporcionar los consultores públicos

```
public Point3D getStartingPoint ();
public Vector3D getDirection ();
```

Clase *tracer.Hit*

- Almacena la información relativa a la intersección con un objeto de la escena:
  - Valor del parámetro t
  - Punto en que ocurre la intersección
  - Normal a la superficie en ese punto
  - Referencia al objeto de la escena con el que se ha intersectado

- Con vistas a reducir sobrecostes de ejecución en el manejo de esta clase de objetos, la clase debe incluir el objeto público estático *Hit.NOHIT* que sirva para indicar que no hay intersección.
  - Esa circunstancia se puede modelar conceptualmente como una intersección a distancia infinita (valor `Float.POSITIVE_INFINITY` para el parámetro t).
- Ha de proporcionar el consultor público

```
public boolean hits ();
```

  para saber si el objeto contiene información de una intersección efectiva.
- Ha de proporcionar el método público

```
public boolean isCloser (final Hit hit);
```

  para saber si el objeto sobre el que se aplica guarda información de una intersección que ocurra más cerca que la intersección del objeto *hit* pasado como argumento.
- Además se deben ofrecer métodos consultores para obtener los valores de los diferentes atributos que forman un objeto de esta clase.

Clase abstracta *tracer.RayGenerator*

- Sirve para englobar las clases que encapsulan los diferentes métodos de generación de rayos.
- Dado que esos métodos son función de la proyección a aplicar, resulta necesario poder operar con el método adecuado en cada situación.
- Incluye un constructor protegido

```
protected RayGenerator (final Camera, final int W, final int H);
```

que requiere la cámara para la cual se han de generar los rayos y los valores que definen la resolución espacial de la imagen a generar.

- Para obtener un rayo establece el método público abstracto

```
public abstract Ray getRay (final int m, final int n);
```

  donde m y n son los índices de columna y de fila del *pixel* que se quiere colorear con ayuda del rayo generado.

- Se considera que las columnas de *pixels* de la imagen se indexan de izquierda a derecha, y las filas se indexan de abajo a arriba; en ambos casos, el indexado se hace desde 0.

Clases para primitivas de modelado geométrico

Clase pública abstracta *models.Object3D*

- Se trata de la clase abstracta que engloba a toda primitiva de modelado geométrico tridimensional.
- Por el momento, cada objeto de la clase *Object3D* ha de disponer de un atributo de color en forma de objeto de la clase *java.awt.Color*.
- Posteriormente un objeto podrá incluir como atributo un material que defina cómo responde la superficie del objeto a la radiación luminosa que recibe.
- Para poder interactuar con los rayos, la clase establece el método público abstracto

```
public abstract Hit intersectionWith (final Ray ray);
```

- El efecto de la aplicación del método debe ser el siguiente:
  - Si el rayo no intersecta con el objeto, se devuelve el objeto estático *Hit.NOHIT*.
  - En caso contrario, se devuelve un objeto de la clase *Hit* con la información del punto de intersección (valor del parámetro α, punto de intersección, normal en el punto de intersección, objeto con el que intersecta).

Subclase *models.Sphere*

- Es una extensión de la clase *Object3D* que como atributos adicionales incluye el punto en el que se localiza el centro de la esfera y su radio.

Subclase *models.Triangle*

- Es una extensión de la clase *Object3D* cuyas instancias se construyen a partir de sus tres vértices de clase *Point3D*.
    - Se espera que esos vértices sean proporcionados en sentido antihorario.

Subclase *models.Plane*

- Es un extensión de la clase *Object3D* que como atributos adicionales incluye:
    - Un objeto de la clase *Point3D* que corresponde a un punto contenido en el plano.
    - Un objeto de la clase *Vector3D* que establece la normal al plano.

Subclase *models.Cylinder*

- Es un extensión de la clase *Object3D* que como atributos adicionales incluye:
    - Un objeto de la clase *Point3D* que corresponde a un punto en el eje del cilindro.
    - Un objeto de la clase *Vector3D* que define la dirección del eje del cilindro.
    - Un escalar r que establece la longitud del radio.
    - Un escalar ℓ que establece la longitud respecto al eje dado.

Subclase *models.Capsule*

- Es un extensión de la clase *Object3D* que sirve para modelar cilindros con tapas semiesféricas.
- Sus instancias se construyen a partir de los mismos elementos que las instancias de la clase *Cylinder*.
- El algoritmo de intersección tiene que considerar que en cada extremo del eje del cilindro finito se sitúa el centro de una esfera de radio r.

Subclase *models.Box*

- Es un extensión de la clase *Object3D* que sirve para modelar ortoedros.
- Sus instancias se construyen a partir de los siguientes elementos:
  - Un objeto de clase *Point3D* que corresponde al centro geométrico del ortoedro.
  - Un objeto de clase *Vector3D* que define la dirección del eje horizontal del ortoedro.
  - Un objeto de clase *Vector3D* que sirve como auxiliar para definir la dirección del eje vertical del ortoedro.
  - Tres escalares positivos w, h y d que respectivamente definen las longitudes en anchura, altura y profundidad de los lados del ortoedro.

Subclase *models.TriangularMesh*

- Es una extensión de la clase *Object3D* para definir superficies mediante una malla de triángulos.
- Como atributos mantiene:
  - La colección de triángulos que forman la estructura reticular.
  - Un objeto de clase *BoundingBox* que sirva para eventualmente reducir el tiempo consumido por el método *intersectionWith()* cuando el rayo no intersecta con ninguno de los triángulos de la malla.

- Debe proporcionar los constructores

```
public TriangularMesh (final Map<Integer, Point3D> vertices,
                       final List<String> facetas,
                       final Color color);
public TriangularMesh (final Map<Integer, Point3D> vertices,
                       final List<String> facetas);
```

  donde:

  - *vertices* es una correspondencia que asocia un índice distinto a cada punto que forma la retícula.
  - *facetas* es una lista donde cada elemento es una tripleta de índices que definen uno de los triángulos de la retícula.
  - *color* es el color a aplicar a todos los triángulos; en el caso del constructor que no incluye ese argumento, se debe asignar un color aleatorio a cada triángulo de la malla.

Subclase *models.Group3D*

- Es una subclase especial de la clase *Object3D* que permite agrupar diferentes primitivas de modelado geométrico.
- Sirve para estructurar la construcción de la escena.
- La clase debe incluir el método público

```
void addObject (final Object3D obj);
```

  para incorporar objetos al grupo.

- El método *intersectionWith()* de esta subclase itera por los objetos almacenados en la estructura llamando a sus respectivos métodos de intersección.
- Devuelve la información correspondiente a la intersección más cercana a la cámara, o el objeto *Hit.NOHIT* en caso de que el rayo no intersecte con objeto alguno.

Subclase *models.AffineTransformation*

- Es una subclase especial de la clase *Object3D* que permite aplicar transformaciones de modelado a modelos definidos.
- Debe proporcionar suficientes constructores públicos para que resulte ágil la definición de combinaciones de los tres tipos básicos de transformaciones.

```
public AffineTransformation (
    final Vector3f s, // Factores de escala
    final Vector3f axis, // Dirección del eje de rotación
    final float theta, // Ángulo de rotación
    final Vector3f d, // Vector de traslación
    final Object3D object // Objeto sobre el que actuar
);

public AffineTransformation (
    final Vector3f s, // Factores de escala
    final Vector3f axis, // Dirección de eje de rotación
    final float theta, // Ángulo de rotación
    final Object3D object // Objeto sobre el que actuar
);

public AffineTransformation (
    final Vector3f axis, // Dirección de eje de rotación
    final float theta, // Ángulo de rotación
    final Vector3f d, // Vector de traslación
    final Object3D object // Objeto sobre el que actuar
);

public AffineTransformation (
    final Vector3f s, // Factores de escala
    final Vector3f d, // Vector de traslación
    final Object3D object // Objeto sobre el que actuar
);

public AffineTransformation (
    final Vector3f axis, // Dirección de eje de rotación
    final float theta, // Ángulo de rotación
    final Object3D object // Objeto sobre el que actuar
);
public AffineTransformation (
    final Vector3f s, // Factores de escala
    final Object3D object // Objeto sobre el que actuar
);
```

Subclase abstracta confinada *models.ProceduralObject3D*

- Es una extensión abstracta de la clase *Object3D*.
- Sirve de clase base sobre la que definir clases para modelos con algoritmos de intersección basados en *ray marching*.
  - Modelos con algoritmos de intersección procedimentales.
- Su método *rayMarching()* encapsula el esquema de marcha que en las clases de modelos procedimentales sirve como pieza clave para la definición del método *intersectionWith()*.
- Las extensiones de esta clase deben proporcionar implementaciones de los métodos:

```
public Hit intersectionWith (final ray ray);
protected float SDF (final Point3D P);
protected float distanceUpperBound (final Point3D P);
```

Subclase *models.Torus*

- Es una extensión de la clase *ProceduralObject3D* para definir superficies tóricas.
- Sus instancias se construyen a partir de los siguientes elementos:
  - Un elemento de clase *Point3D* que define el centro geométrico del toro.
  - Un elemento de clase *Vector3D* que junto con el anterior sirve para definir el eje de revolución del toro.
  - Un escalar R que define la distancia al eje de revolución a la que se encuentra el centro de la circunferencia que define la superficie tórica.
  - Un escalar r que define el radio de esa circunferencia.

Subclase *models.Ellipsoid*

- Es una extensión de la clase *ProceduralObject3D* para definir elipsoides.
- Sus instancias se construyen a partir de los siguientes elementos:
  - Un elemento de clase *Point3D* que define el centro geométrico del elipsoide.
  - Un objeto de clase *Vector3D* que define la dirección del eje horizontal del elipsoide.
  - Un objeto de clase *Vector3D* que sirve como auxiliar para definir la dirección del eje vertical del elipsoide.
  - Tres escalares R_x, R_y y R_z que los radios del elipsoide en cada eje de su sistema de referencia de modelado.

Clases relacionadas con vista y proyección

Clase *view.Camera*

- Esta clase sirve para mantener la información relativa a la cámara como punto de vista desde el que sintetizar la imagen.
- Los atributos de los objetos de esta clase son:
  - El punto de emplazamiento de la cámara.
  - Una matriz de la transformación que permite pasar de coordenadas de cámara a coordenadas de escena (inversa de la matrix de vista).
  - Un objeto de la clase *Projection* que define la óptica aplicada por la cámara para generar la imagen.
- El constructor de una cámara toma como argumentos los parámetros de visualización:
  - Punto V de emplazamiento de la cámara
  - Punto C al que se orienta la cámara (punto *lookAt*)
  - Vector vertical auxiliar *up*

- El punto C y el vector *up* son empleados en un cálculo de alineamiento, que junto con el punto V, es necesario para componer la inversa de la matriz de la transformación de vista. Para conocer mejor este proceso, léase el fichero *transformacion.vista.pdf*.

- Con el fin de poder cambiar la óptica de una cámara se ha de ofrecer el método público

```
public void setProjection (final Projection projection);
```

  que asocia una proyección concreta a la cámara sobre la que se aplica.

- Esa matriz inversa es empleada por los métodos públicos

```
public void toSceneCoordenates (final Point3D P);
public void toSceneCoordenates (final Vector3D v);
```

  para traducir coordenadas de cámara a coordenadas de escena.

- Con el fin de simplificar el proceso de trazado de rayos, esta clase incluye el método

```
public RayGenerator getRayGenerator (final int W, final int H) {
  return optics.getRayGenerator(this, W, H);
}
```

  donde *optics* es el nombre del atributo empleado para referenciar a la proyección asociada a la cámara.

- Deben proporcionarse además los consultores públicos

```
public Point3D getPosition ();
public Vector3D getLook ();
public Projection getProjection ();
```

  que devuelvan información propia de la cámara sobre la que se aplican.

Clase abstracta *view.Projection*

- Engloba los diferentes tipos de proyecciones que se pueden elegir para generar una imagen.
- Sus atributos son:
  - Anchura w de la ventana de proyección
  - Altura h de la ventana de proyección
- Ha de considerarse que el vector de vista de la cámara con la que se asocie es normal al plano de proyección.
- Vista desde la posición de la cámara, la ventana de proyección se presenta como muestra la figura 1.

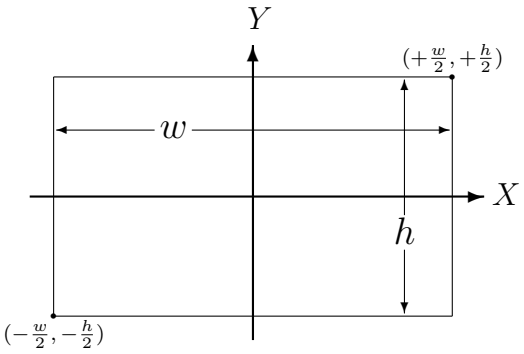

Figura 1: situación de la ventana de proyección en el plano. La cámara está situada sobre el eje Z, que es ortogonal al plano de proyección.

- Debe incluir el método púbico abstracto

```
abstract RayGenerator getRayGenerator (final Camera camera,
                                       final int W, final int H);
```

  donde W y H respectivamente son la anchura y la altura en *pixels* de la imagen a sintetizar.
- Deben proporcionarse además los consultores públicos

```
public float getWidth ();
public float getHeight ();
```

que devuelvan información propia de la proyección sobre la que se aplican.

Subclase *view.Orthographic*

- Corresponde a una proyección paralela que proyecta en la dirección del vector del vista de la cámara con la que está asociada.
- Dispone de un constructor al que se deben proporcionar la altura h de la vantana de proyección y la relación de aspecto entre altura y anchura (cociente entre altura h y anchura w de la ventana).
- La definición del método *getRayGenerator* debe ser la siguiente:

```
RayGenerator getRayGenerator (final Camera camera,
                              final int W, final int H) {
  return new OrthographicRayGenerator(camera, W, H);
}
```

donde la clase *OrthographicRayGenerator* es una clase estática interna en la que queda implementada la forma concreta que deben tener los rayos cuando se sintetiza la imagen aplicando una proyección ortogonal.

  - Es extensión de la clase *RayGenerator*.

- En particular, el método *getRay()* de la clase *OrthographicRayGenerator* devuelve un rayo que parte del punto del plano de proyección cuyas coordenadas de cámara son:

$$\begin{aligned} x &= \left(m + \frac{1}{2}\right) \cdot \frac{w}{W} - \frac{w}{2} \\ y &= \left(n + \frac{1}{2}\right) \cdot \frac{h}{H} - \frac{h}{2} \\ z &= 0 \end{aligned}$$

y cuya dirección es paralela a la dirección de vista (véase figura 2).

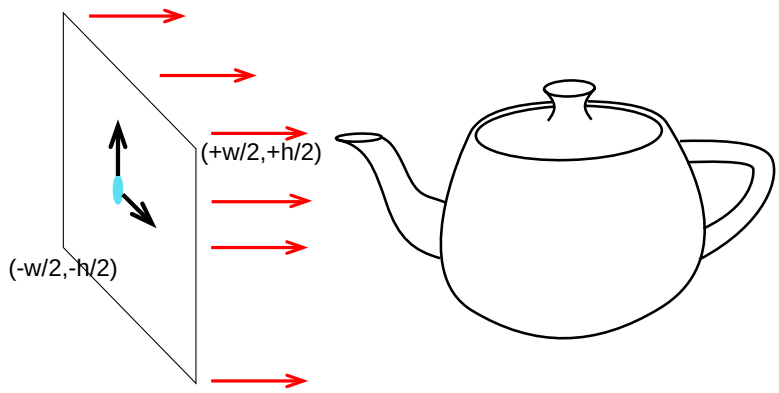

Figura 2: rayos enviados en dirección paralela a la dirección de vista.

Subclase *view.Perspective*

- Corresponde a una proyección en perspectiva cuyo centro de proyección es el emplazamiento de la cámara con la que está asociada la proyección.
- Dispone de un constructor al que se deben proporcionar dos valores:
  - Ángulo *fov* de apertura vertical del campo visual (en grados)
  - Relación de aspecto a (ratio entre anchura y altura de la ventana)

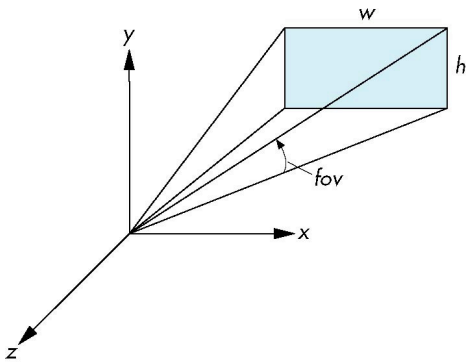

Figura 3: especificación de la ventana de proyección asociada a una proyección en perspectiva respecto del sistema de referencia de la cámara.

- A partir de esos tres valores se calculan las dimensiones de la ventana de proyección (véase figura 3) aplicando las siguientes fórmulas:

$$\begin{aligned} h &= 2 \cdot \tan\left(\frac{fov}{2}\right) \\ w &= a \cdot h \end{aligned}$$

- La definición del método *getRayGenerator* debe ser la siguiente:

```
RayGenerator getRayGenerator (final Camera camera,
                              final int W, final int H) {
  return new PerspectiveRayGenerator(camera, W, H);
}
```

  donde la clase *PerspectiveRayGenerator* es una clase estática interna en la que queda implementada la forma concreta que deben tener los rayos cuando para sintetizar la imagen se aplica una proyección en perspectiva.

  - Es extensión de la clase *RayGenerator*.

- En concreto, el método *getRay()* de la clase *PerspectiveRayGenerator* devuelve un rayo (véase figura 4) que parte del punto de emplazamiento de la cámara y pasa por el punto de la ventana de proyección cuyas coordenadas de cámara son:

$$\begin{aligned} x &= \left(m + \frac{1}{2}\right) \cdot \frac{w}{W} - \frac{w}{2} \\ y &= \left(n + \frac{1}{2}\right) \cdot \frac{h}{H} - \frac{h}{2} \\ z &= -1 \end{aligned}$$

Subclase *view.Angular*

- Corresponde a una proyección de tipo *fish-eye* en la que la escena se proyecta sobre un casquete esférico que sobresale del plano de proyección.

- Ese casquete queda definido a partir del punto de emplazamiento de la cámara, de la dirección de vista y de una apertura ω.

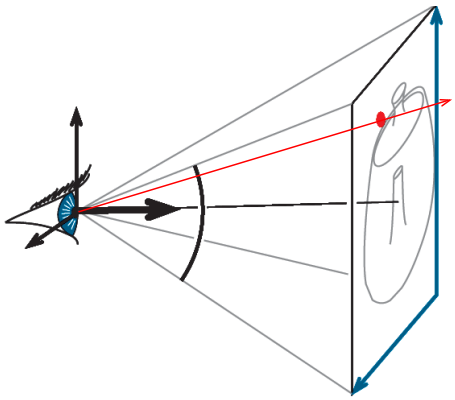

Figura 4: rayo que parte del punto de emplazamiento de la cámara y pasa por el punto de la ventana de proyección coloreado en rojo.

    - Se debe considerar que el plano de proyección está a distancia 1 de la cámara.

- Dispone de un constructor al que se debe proporcionar la apertura ω del campo de visión angular respecto al eje de visión.