Apéndice A - Active Rendering

En este anexo se describe el framework utilizado para brindar aceleración gráfica a las aplicaciones desarrolladas en esta tesis. Este framework, llamado active rendering, fue desarrollado por Andrew Davison en [38], en donde se proporciona el código para su implementación. Este código fue modificado con permiso del autor para adaptarlo nuestras necesidades.

1. Descripción del framework active rendering

Una de las características de la versión 1.1.0 de JOGL es que permite acceder de manera directa a la superficie de dibujo y al contexto de OpenGL. De esta manera, una aplicación en Java puede extender la clase Canvas del paquete AWT incluyendo un thread, al que llamaremos rendering thread, que se encargue de desarrollar las tareas relacionadas con la graficación utilizando el API de OpenGL. La estructura de una aplicación de este tipo se ilustra en la figura A1.

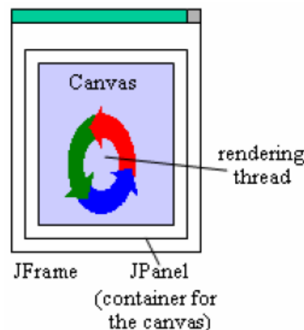

Figura A1: Estructura de una aplicación en Java utilizando active rendering [38].

Las tareas desarrolladas por el rendering thread pueden resumirse en el siguiente pseudo-código:

```
1.  make the context current for this thread
2.  initialize rendering
3.  while isRunning
4.              update aplication
5.              render scene
6.              put the scene onto the canvas
7.              sleep a while
8.              maybe do updates without rendering them
9.              gather statistics
10. discard the rendering context
11. exit
```

Figura A2: Pseudo-Código para el framework active rendering.

La principal ventaja de este framework es que nos permite controlar de manera directa la ejecución de la aplicación, ya que se puede agregar código para suspender su actualización. Esto es de gran utilidad cuando la aplicación no se encuentra en primer plano, por ejemplo, cuando la ventana que la contiene esta minimizada o no es la ventana activa. Además, otra de las ventajas de este framework es que permite tener acceso al código de medición del tiempo de ejecución, lo que permite controlar el número de frames por segundo (FPS) al que trabaja el rendering thread para separarlo del número de actualizaciones por segundo de la aplicación (UPS). Los detalles de implementación de este framework se describen a continuación.

2. Implementación del Framework Active Rendering

La implementación del framework active rendering se encuentra en la clase Renderer. Esta clase proporciona la funcionalidad básica para aplicaciones que requieren aceleración gráfica y se encarga de controlar la interacción con el usuario a través del manejo de eventos del teclado y del mouse.

En términos generales, un objeto de la clase Renderer es un Canvas con un thread, que se encarga de dibujar en él, a una frecuencia establecida con el fin de aproximarse a un número de frames por segundo indicado al inicio de la ejecución. Los principales métodos y relaciones de la clase Renderer se muestran en la figura A3.

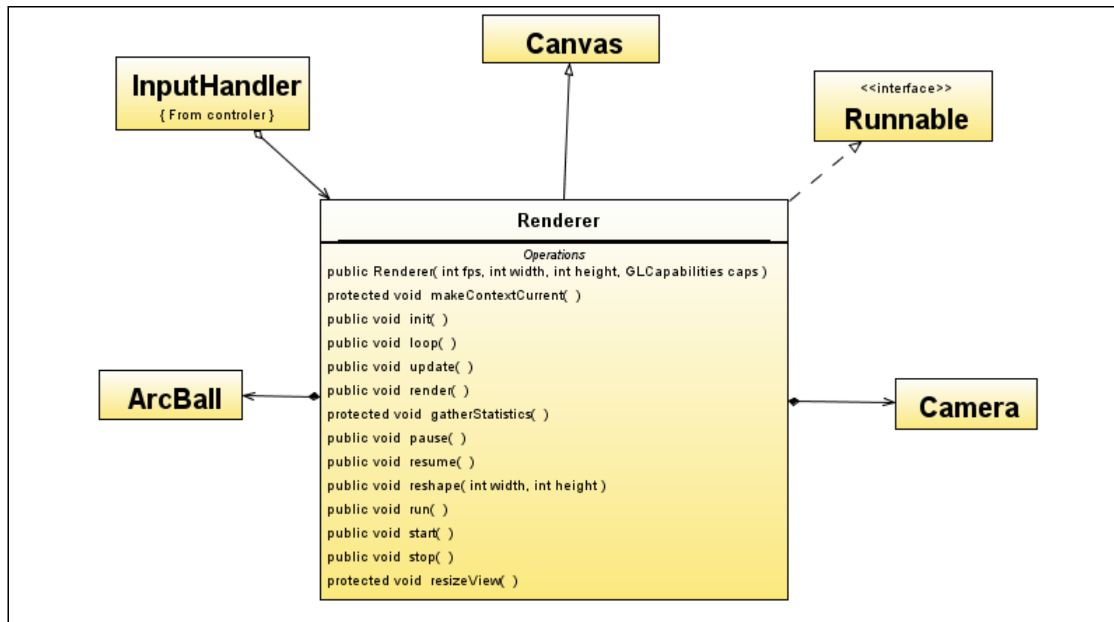

Figura A3: Diagrama de la clase Renderer con sus principales métodos y relaciones.

Para crear objetos de esta clase, se proporciona un único constructor, el cual se encarga de inicializar el Canvas con una configuración adecuada, además de crear una superficie para graficar y un contexto de acuerdo a los parámetros que recibe, entre otras tareas.

```
/*** OpenGL related***/
protected GLDrawable drawable;    //rendering surface
protected GLContext context;      //rendering context (holds OpenGL state info)
protected GL gl;                  // OpenGL API interface
protected GLU glu;                // OpenGL Graphics Utility Library interface
protected GLUT glut;              // OpenGL Utility Toolkit interface

public Renderer(int fps, int width, int height, GLCapabilities caps){
   super(Renderer.getGraphicsConfiguration(caps));
   this.period = (long)1000000000.0/fps;    //in nanoseconds
   this.renderAreaWidth = width;
   this.renderAreaHeight = height;

   cam = new Camera(new Vector3f(0, 5, 50), new Vector3f(0, 5, 0), new Vector3f(0, 1, 0));
   arcBall = new ArcBall(renderAreaWidth, renderAreaHeight);

   //get a rendering surface and a context for this canvas
   drawable = GLDrawableFactory.getFactory().getGLDrawable(this, caps, null);
   context = drawable.createContext(null);
```

```
        /*** Statistics initialization ***/
        fpsStore = new double[NUM_FPS];
        upsStore = new double[NUM_FPS];
        for(int i = 0; i < NUM_FPS; i++){
            fpsStore[i] = 0.0;
            upsStore[i] = 0.0;
        }
}
```

El parámetro fps indica el número de veces que deseamos actualizar lo que vemos en pantalla por segundo. En base a este parámetro se calcula el periodo de tiempo, en nanosegundos, requerido para alcanzar dicha tasa, y éste se utiliza para poner a dormir al thread de animación entre cada actualización. El resto de los parámetros indican las dimensiones del área de dibujo (width y height) y el conjunto de capacidades que debe soportar nuestro contexto de OpenGL, que se especifican en un objeto de la clase GLCapabilities. En el constructor de esta clase se crean objetos de la clase Camera y ArcBall. La clase Camera se utiliza para cambiar el punto desde el que se observan los objetos que se grafican en pantalla, mientras que la clase ArcBall se utiliza para cambiar la orientación de dichos objetos. Esta es una de las principales modificaciones hechas al código desarrollado por Andrew Davison. En la versión original, el código para manipular la cámara se encuentra dentro de la implementación del framework active rendering. En cambio, en nuestra versión la manipulación de la cámara se separa y se pone en una clase aparte, pues nuestra intención es que la clase Renderer sirva de esqueleto para el desarrollo de cualquier aplicación gráfica. Además, debido a que no todas las aplicaciones requieren de la interacción del usuario a través de la manipulación de una cámara, esta separación fue de gran utilidad.

Otra modificación de gran utilidad fue la adición de un objeto de la clase ArcBall a la implementación del framework active rendering, de manera que si una aplicación requiere que el usuario pueda aplicar rotaciones a un objeto utilizando el mouse, sólo tiene que agregar una línea de código. Esto se describe más adelante.

Ejecución del rendering thread

La clase Renderer implementa el método run() de la interfaz Runnable. Este método constituye una parte fundamental para esta clase, debido a que a partir de su ejecución, inicia el ciclo de animación de nuestra aplicación. El pseudo-código presentado al inicio de este apéndice se implementa en este método.

```
public void run(){
    init();
    loop();
    context.destroy();
}
```

Al iniciar su ejecución, el rendering thread invoca al método init() de la clase Renderer. Este método se encarga de inicializar el contexto de OpenGL, y es el lugar indicado para habilitar el buffer de profundidad, crear luces, cargar texturas, display lists, entre otras tareas.

```
public void init(){
    makeContextCurrent();

    gl = context.getGL();
    glu = new GLU();
    glut = new GLUT();

    resizeView();

    gl.glShadeModel(GL.GL_SMOOTH);
    gl.glClearColor(0.17f, 0.65f, 0.92f, 0.0f); //blue
    gl.glLightfv(GL.GL_LIGHT0, GL.GL_POSITION, light_position, 0);

    gl.glEnable(GL.GL_LIGHTING);
    gl.glEnable(GL.GL_LIGHT0);
    gl.glEnable(GL.GL_DEPTH_TEST);
    gl.glHint(GL.GL_PERSPECTIVE_CORRECTION_HINT, GL.GL_NICEST);

    context.release();
}
```

Además de las tareas mencionadas anteriormente, el método init() incluye una llamada al método resizeView(). Este método es llamado cada vez que la ventana de la aplicación se mueve o cambia de tamaño, incluyendo la primera vez que aparece en pantalla. Cuando esto ocurre, es necesario actualizar el tamaño y la perspectiva de nuestro Canvas, y esto se hace con el siguiente método.

```
protected void resizeView(){
    gl.glViewport(0, 0, renderAreaWidth, renderAreaHeight);
    gl.glMatrixMode(GL.GL_PROJECTION);
    gl.glLoadIdentity();
    glu.gluPerspective(45.0, (float)renderAreaWidth/(float)renderAreaHeight, 1, 500);
    gl.glMatrixMode(GL.GL_MODELVIEW);
    gl.glLoadIdentity();
    arcBall.setBounds(renderAreaWidth, renderAreaHeight);
}
```

Después de inicializar el contexto de OpenGL, el rendering thread ejecuta el método loop(). Este método representa la parte más importante del framework descrito por Andrew Davison, pues es aquí donde se lleva a cabo lo que el autor llama active rendering.

```
public void loop(){
    initStatistics();
    isRunning = true;
    while(isRunning){
        makeContextCurrent();
        update();
        render();
        drawable.swapBuffers();
        gatherStatistics();
        sleep_a_while();
        context.release();
        storeStatistics();
    }
    printStats();
}
```

El método loop() comienza con una llamada al método initStatistics(), en el que se inicializa un conjunto de variables utilizadas para recolectar estadísticas que miden el desempeño de nuestra aplicación. Posterior a esto, se asigna el valor true a la variable

isRunning, y se inicia un ciclo de actualización, despliegue en pantalla y recolección de estadísticas que dura mientras la aplicación este ejecutándose.

Dentro del método loop(), la parte más importante consiste en las llamadas a los métodos update() y render(). Estos métodos se encargan de actualizar el estado de nuestra aplicación y de desplegar en pantalla el resultado de dichas actualizaciones, respectivamente. Sin embargo, debido a que la clase Renderer tiene como propósito proporcionar la funcionalidad básica para desarrollar aplicaciones gráficas, su método update() no realiza tarea alguna, razón por la que debe sobrescribirse en aplicaciones futuras.

```
public void update(){
   /*
   if(isRunning && !isPaused) {
      //do some update work
   }
   */
}
```

En cuanto al método render(), éste tiene el propósito de desplegar en pantalla el resultado de las actualizaciones realizadas en el método update(). En la clase Renderer, este método únicamente limpiar el área de dibujo y establece la ubicación de la cámara y las rotaciones aplicadas por el usuario a través del mouse. Por esta razón, este método también debe sobrescribirse en aplicaciones posteriores.

```
public void render(){
   if(context.getCurrent() == null){
      System.err.println("CURRENT CONTEXT IS NULL");
      System.exit(1);
   }

   if(isResized){
      resizeView();
      isResized = false;
   }

   gl.glClear(GL.GL_COLOR_BUFFER_BIT | GL.GL_DEPTH_BUFFER_BIT);
   gl.glMatrixMode(GL.GL_MODELVIEW);
```

```
    gl.glLoadIdentity();

    //setup look at matrix
    glu.gluLookAt(cam.positionX(), cam.positionY(), cam.positionZ(),
            cam.lookAtX(),   cam.lookAtY(),   cam.lookAtZ(),
            cam.upX(),      cam.upY(),      cam.upZ());

    //setup arcball rotation
    gl.glMultMatrixf(matrix, 0);

    //render something

    gl.glFlush();
}
```

Después de la ejecución de los métodos update() y render(), se muestra la escena en pantalla y se recolectan algunas estadísticas. Esta parte es una de las más importantes en la clase Renderer, ya que es aquí donde se calcula el tiempo que demora la ejecución de los métodos update() y render(). Dependiendo de este tiempo, el rendering thread se pone a dormir por un periodo de tiempo que le permita alcanzar el número de frames por segundo (FPS) especificado durante la construcción del objeto de la clase Renderer.

Si la ejecución de estos métodos se demora demasiado, puede ser necesario realizar algunas actualizaciones sin mostrar en pantalla los cambios que se generen. El resultado que se obtiene de esto es una aplicación que corre a un número de FPS muy cercano al que se especificó, evitando desplegar en pantalla las actualizaciones que consumen más tiempo.

En la clase Renderer se distinguen dos índices: el número de frames por segundo (FPS) que mide al número de veces que se ejecuta el método render(), y el número de actualizaciones por segundo (UPS) que mide el número de veces que se ejecuta en método update(). Estos índices son diferentes, pues es posible que en algunos equipos más lentos se limite el número de FPS, sin embargo, el programa lleva a cabo actualizaciones adicionales sin mostrar en pantalla el resultado que generan, de manera que el número de UPS es muy cercano al número de FPS solicitado.

Esta separación nos permites crear animaciones de muy buena calidad, incluso en equipos con hardware incapaz de trabajar al número de FPS requerido.

Control sobre la ejecución del rendering thread

La clase Renderer proporciona métodos para controlar la ejecución del rendering thread. Estos métodos se describen a continuación.

Para iniciar la ejecución de una aplicación, es necesario invocar al método start() de la clase Renderer. Este método crea un thread que ejecuta el método run() definido en esta misma clase. La ejecución del thread inicia con el llamando a su método start().

```
public void start(){
    if(animator == null || !isRunning){
        animator = new Thread(this, "Rendering Thread");
        animator.start();
    }
}
```

Para detener la ejecución del rendering thread, se proporciona el método stop(). Este método únicamente asigna el valor false a la variable isRunning, lo que provocará que el thread termine la ejecución del método loop() y regrese al método run() para terminar la ejecución de la aplicación.

```
public void stop(){
    isRunning = false;
}
```

Para pausar y reanudar la actualización del estado de nuestra aplicación, se proporcionan los métodos pause() y resume(). Típicamente, estos métodos son invocados cuando la ventana que contiene nuestro Canvas deja de estar activa, por ejemplo, cuando se

le minimizada o cuando la ventana se encuentra total o parcialmente cubierta por otra ventana. Estos métodos suspenden y reinician las actualizaciones indicadas en el método update(). Más adelante se describe la forma en que se les invoca.

```
public void pause(){
   isPaused = true;
}

public void resume(){
   isPaused = false;
}
```

Las llamadas a los métodos stop(), pause() y resume() se realizan cuando se produce un evento generado por una ventana. Por esta razón, es necesario que la ventana de nuestra aplicación implemente los métodos de la interfaz WindowsListener de la forma que se describe a continuación.

Integración dentro de una ventana

Debido a que la clase Renderer se extiende de la clase Canvas AWT, los objetos de la clase Renderer pueden agregarse en diferentes contenedores, tanto del paquete AWT como del paquete Swing. De esta manera, suponiendo que deseamos agregar un objeto de la clase Renderer en un JFrame, la estructura de nuestra aplicación sería similar a la que se muestra en la figura a.1, al inicio de este apéndice.

Los objetos de la clase Renderer pueden agregarse de manera directa en contenedores como una ventana o un panel utilizando diferentes layouts. Sin embargo, lo más recomendable es agregar los objetos de esta clase en un contenedor, por ejemplo, un JPanel, y agregar este último a una ventana. De esta manera, delimitamos el área de dibujo, y al mismo tiempo, protegemos a otros componentes gráficos como botones y menús, de ser cubiertos por nuestro objeto de la clase Renderer, que no debemos olvidar, se trata de un Canvas. Este problema surge cuando mezclamos componentes ligeros (Swing) con componentes pesados (AWT) como nuestro Renderer.

Una vez que hemos agregado el Canvas a la ventana utilizando el layout de nuestra elección, es necesario implementar los métodos de la interfaz WindowListener de la siguiente forma:

```
/*** WindowListener Interface Methods ***/
public void windowActivated(WindowEvent e){
   renderer.resume();
}

public void windowDeactivated(WindowEvent e){
   renderer.pause();
}

public void windowDeiconified(WindowEvent e){
   renderer.resume();
}

public void windowIconified(WindowEvent e){
   renderer.pause();
}

public void windowClosing(WindowEvent e){
   renderer.stop();
}

public void windowClosed(WindowEvent e){
}

public void windowOpened(WindowEvent e){
}
```

La implementación de estos métodos nos permitirá suspender y reanudar la actualización de nuestro Renderer, dependiendo de si la ventana que lo contiene se encuentra o no activa. Por ejemplo, si nuestro Renderer muestra en pantalla algún tipo de animación, esta animación se ejecutará mientras la ventana que lo contiene esté activa. En caso contrario, la animación se suspende cuando la ventana pasa a un segundo plano o cuando es minimizada, y eventualmente se reanudará desde el momento en que se detuvo. La ejecución de una animación termina cuando la ventana que contiene a nuestro Renderer se cierra.

Por último, con el fin de que nuestra aplicación sea capaz de manejar cambios en el tamaño de la ventana, es necesario implementar el método *componentResized(ComponentEvent e)* de la interfaz ComponentListener, y agregar este listener al contenedor de nuestro Renderer de la siguiente forma:

```
public void componentResized(ComponentEvent e){
    Dimension dimension = e.getComponent().getSize();
     renderer.reshape(dimension.width, dimension.height);
}
```

Al hacer esto, el área de dibujo cambiará su dimensión cada vez que el contenedor de nuestro Renderer cambie de tamaño. Por esta razón, se recomienda que el Renderer se agregue en un Panel o JPanel, que a este contenedor se le agregue el ComponentListener indicado, y que sea el contenedor el que se agregue a la ventana. Así, el área de dibujo quedará limitada por la dimensión del panel, y no por la dimensión de la ventana, lo que nos permitirá agregar a esta última, otros componentes como botones y menús, además del Renderer.

Manejo de eventos del teclado y del Mouse

El manejo de eventos del teclado y del mouse permite a nuestras aplicaciones interactuar con el usuario. Para este propósito se creo la clase InputHandler, cuya función es recibir los eventos generados por el usuario y notificar al Renderer para que realice las acciones correspondientes en respuesta al evento generado. El diagrama de esta clase se muestra en la figura A4.

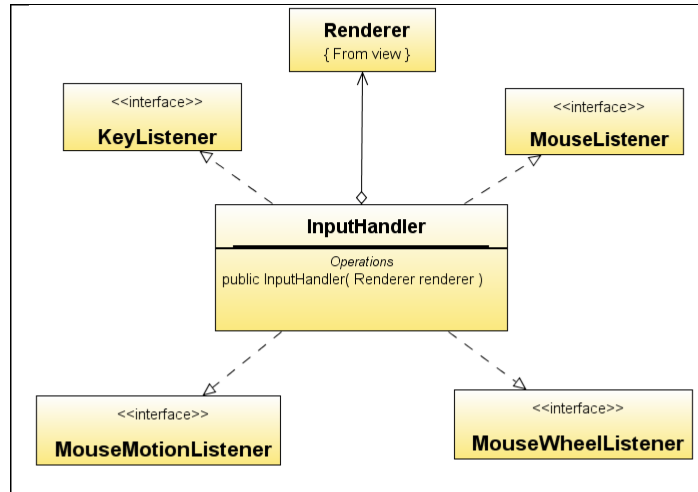

Figura A4: Diagrama de la clase InputHandler.

Es importante resaltar que el manejo de eventos se realizó de esta forma porque en Java los eventos que recibe una ventana son manejados por un thread especial llamado *event dispath thread*, y en la especificación de OpenGL se señala que sólo un thread puede acceder al contexto de OpenGL, que en el caso de la clase Renderer se trata del rendering thread. Por esta razón, cuando se recibe un evento, la clase InputHandler invoca al método correspondiente en la clase Renderer, con lo que se modifica el valor de una variable. De esta manera, cada vez que el rendering thread ejecuta el método render(), se obtiene el valor de la variable y los objetos se grafican de forma adecuada.

Los eventos que recibe la clase InputHandler provocan cambios en la posición de la cámara, o bien, un cambio en la orientación de los objetos que se grafican, y con esto, se modifica la forma en que el usuario ve lo que se muestra en pantalla. Estos cambios se generan a través de los métodos de las clases Camera y ArcBall, cuyo diagrama se muestra en la figura A5.

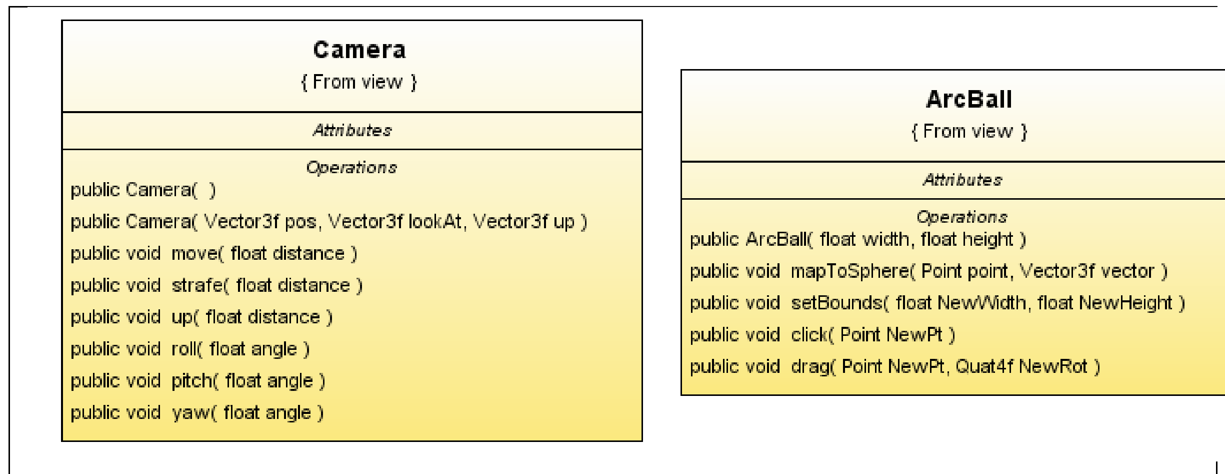

Figura A5: Diagrama de las clases Camera y ArcBall.

La implementación de estas clases se basó en los tutoriales que se describen en [39] y [40]. El código que aparece en estos tutoriales estaba en C++ y en el caso de la cámara, describía una implementación utilizando Direct3D, razón por la que fue portado a Java y se modificó para ser utilizado con OpenGL.

Los eventos que permiten la interacción del usuario con la clase Renderer se resumen en la tabla A1.

Teclas y movimientos del mouse	Acción que controla
↑	Mover la cámara al frente
↓	Mover la cámara hacia atrás
←	Girar la cámara a la izquierda
→	Girar la cámara a la derecha
CTRL + ↑	Mover la cámara hacia arriba
CTRL + ↓	Mover la cámara hacia abajo
CTRL + ←	Mover la cámara a la izquierda
CTRL + →	Mover la cámara a la derecha
X	Rotación positiva de la escena alrededor del eje X.
CTRL + X	Rotación negativa de la escena alrededor del eje X.

Y	Rotación positiva de la escena alrededor del eje Y.
CTRL + Y	Rotación negativa de la escena alrededor del eje Y.
Z	Rotación positiva de la escena alrededor del eje Z.
CTRL + Z	Rotación negativa de la escena alrededor del eje Z.
R	Regresar la cámara a su posición inicial.
Girar la rueda del mouse	Acercar/Alejar la escena (zoom in \ zoom out).
Arrastrar el puntero del mouse presionando el botón izquierdo.	Rotación libre de la escena alrededor de los ejes X, Y, Z.
Presionar el botón derecho del mouse.	Regresar la escena a su posición y orientación inicial.

Tabla A1. Control de eventos del teclado y mouse.

El hecho de que estos métodos se implementen en la clase Renderer nos permite heredar su funcionalidad a aplicaciones más específicas creadas a partir de esta clase, lo que favorece un desarrollo mucho más rápido y sencillo.

De esta manera, para que un objeto de la clase Renderer pueda responder a eventos del teclado o del mouse, es necesario crear un objeto de la clase InputHandler y agregarlo como listener para los diferentes tipos de eventos de la siguiente forma:

```
…
Renderer renderer = new Renderer(fps, width, height, caps);
InputHandler handler = new InputHandler(renderer);
renderer.addMouseMotionListener(handler);
renderer.addMouseWheelListener(handler);
renderer.addKeyListener(handler);
…
```