Universidad Autónoma de Querétaro
Facultad de Informática
Ingeniería en desarrollo de software

Virtualización del complejo arquitectónico de la facultad de Informática de la Universidad Autónoma de Querétaro

Tesis

Que como parte de los requisitos para obtener el grado de
Ingeniero en desarrollo de software

Presenta:
Alexei Horta Moiseevich

Dirigido por:
MSI. Fausto Abraham Jacques García

Nombre del Sinodal Presidente	_____ Firma
Nombre del Sinodal Secretario	_____ Firma
Nombre del Sinodal Vocal	_____ Firma
Nombre del Sinodal Suplente	_____ Firma
Nombre del Sinodal Suplente	_____ Firma

_____ _____

M.S.I.D. Juan Salvador Hernández Valerio
Director de la Facultad

Centro Universitario
Querétaro, Qro.
Fecha (será el mes y año de aprobación del Consejo Universitario)

RESUMEN

La simulación en tercera dimensión ha sido muy útil y popular, sus aplicaciones se han visto en campos como la automotriz, la aeronáutica, la robótica, etcétera. Esta investigación tiene como finalidad recrear la facultad de informática de la universidad autónoma de Querétaro y simular un recorrido virtual a través de sus edificios y exteriores, permitiendo al usuario conocer las instalaciones que conforman la facultad, así como curiosidades e información que se consideró pertinente exponer. El resultado se podrá visualizar como un programa de escritorio y como una aplicación para dispositivos móviles con sistema operativo Android. El proyecto se desarrolló en el entorno de desarrollo integrado Android Studio complementado con librearías del framework LibGDX. Este poderoso framework multiplataforma permite acceso directo a funcionalidades de bajo nivel como OpenGL. También provee facilidades para tareas comunes que facilitan el desarrollo al programador como cálculos trigonométricos y de algebra lineal. El presente trabajo se enfocará primordialmente a la codificación realizada con LibGDX en un entorno tridimensional, y se abordará de manera secundaria los temas de modelación en tercera dimensión y elaboración de texturas.

(**Palabras clave:** LibGDX, tridimensional, Querétaro)

SUMMARY

The third dimensional simulation has been very useful and popular, its applications have been seen in fields like automotive, aeronautics, robotics, etcetera. This research aims to recreate the computer science faculty of the Autonomous University of Querétaro and simulate a virtual tour through its buildings and exteriors, allowing the user to know the facilities that make up the faculty, as well as curiosities and information that was considered relevant to expose . The result can be viewed as a desktop program and as an application for mobile devices with Android operating system. The project was developed in the integrated development environment Android Studio complemented with libraries of the LibGDX framework. This powerful cross-platform framework allows direct access to low-level features such as OpenGL. It also provides facilities for common tasks that facilitate the development of the programmer as trigonometric calculations and linear algebra. The present work will focus primarily on the coding performed with LibGDX in a three-dimensional environment, and will address in a secondary way the themes of third-dimensional modeling and textures elaboration

(**Key words:** LibGDX, third-dimensional, Querétaro)

AGRADECIMIENTOS

Agradezco primordialmente a mis padres que me han apoyado bastante; ellos han dejado que tome mis decisiones, aunque tratando de aconsejarme y de guiarme a lo que consideraban era lo mejor para mí.

Al profesor Abraham Jacques García por el apoyo y asesoramiento que me brindó, así como la confianza que depositó en mí. Fue alentador y me inspiró a terminar un proyecto que fue todo un reto.

A karla Luna, quién me vio empezar y terminar esta investigación; su apoyo fue importante para este proyecto y para mi alma.

A mis compañeros de la facultad, particularmente a "El Nicocho" y a Darwin, los cuales hicieron de la universidad una experiencia peculiar.

A la Universidad Autónoma de Querétaro y a la Facultad de informática por el conocimiento adquirido y por la grata experiencia de ser un orgulloso alumno de dicha institución.

TABLA DE CONTENIDOS

INDICE DE FIGURAS

INDICE DE CÓDIGOS

INDICE DE TABLAS

1. INTRODUCCIÓN

A mediados de los noventa se popularizo de gran manera los juegos en tercera dimensión. Hoy en día, alrededor de todos los títulos AAA en consolas y *PC's* son videojuegos 3D. Y a medida en que la tecnología de dispositivos móviles evoluciona, más y más desarrolladores están optando por tecnologías móviles (Madhav, 2014). Las aplicaciones más recientes, que aprovechan al máximo las capacidades de un dispositivo móvil y los antecedentes que se expondrán a continuación, no solo han inspirado en la realización de esta investigación, sino también el aportar una aplicación con un entorno tridimensional exitoso.

1.1 ANTECEDENTES

Experiencias virtuales personalizadas lo han logrado múltiples organizaciones, centros de desarrollo o individuos cuyo interés es dar a conocer instalaciones de interés, un claro ejemplo de esto es el vaticano. Con modelos tridimensionales, les permite a los usuarios explorar el vaticano sin la necesidad de viajar. Lo atractivo de este tour es la modalidad de recorrer los pasillos del vaticano a través de diferentes periodos de la historia.

Esta experiencia es multiplataforma y funciona tanto en *Ipad* como en computadoras de escritorio por medio de un navegador web. Considerando la cantidad de personas católicas interesadas en este representativo complejo arquitectónico, resulta una idea llevada a cabo con gran efectividad.

Figura 1.1: Recorrido virtual en la página web del Vaticano

La figura 1.1 muestra imágenes tomadas directamente del recorrido virtual en el Vaticano desde un explorador web. Como se puede apreciar, el recorrido solo busca mostrar ciertos puntos de interés; la personalización del recorrido va acorde al propósito y ameniza la experiencia del usuario con música, sonidos y la posibilidad de moverte libremente por el espacio tridimensional, esto con la ayuda de la configuración de un par de botones con funciones básicas.

Este proyecto de tesis aspira a lograr un producto similar al ejemplo anteriormente presentado pero orientado al gremio académico que conforma la facultad de informática. Continuando con la comparación, un motor físico más sofisticado y un mundo virtual con menos limitaciones.

Dicho mundo virtual es ejemplificado en el proyecto desarrollado por *Shenyang Agriculture University*, la cual elaboró un tour virtual de su campus. El objetivo de dicho proyecto es el de proveer a aquellos que no tienen la posibilidad de recorrer el campus en persona, la opción de una vivida experiencia visual (Song, Zhang y Yang 2010) coincidiendo así con una de las justificaciones base del proyecto de tesis. La tecnología usada fue el software de realidad virtual *VR-Platform* y el software de diseño de modelos tridimensionales 3DSMax.

El desarrollo del proyecto del tour virtual del campus *Shenyang* está más enfocado al diseño de modelos y lo que implica esto, en donde la optimización de modelos, optimización del número de polígonos en los modelos y optimización del número de modelos, son técnicas empleadas para un mejor rendimiento del producto. Dicho sistema cumple su función, sin embargo está lejos del campus virtual esperado (Song et al., 2010) En la figura 3, la imagen del lado izquierdo se aprecia el desarrollo logrado por Shenyang Agriculture University.

Por otra parte, *Chinese University of Hong Kong* desarrolló su propio Campus virtual usando la tecnología *OpenSimulator*, el cual puede usarse para simular un entorno virtual similar a *Second Life* (Chen et al, 2010). Parte importante de dicho proyecto son las clases virtuales que los usuarios podrán tomar en un salón virtual. La herramienta propuesta, la manera de desarrollo y los alcances del proyecto difieren mucho del presente trabajo, sin embargo, se comparte los objetivos principales. En la figura 3, la imagen del lado derecho se aprecia el desarrollo logrado por *Chinese University of Hong Kong*.

El mundo virtual *Second Life* fue una parte importante en el trabajo hecho por la Universidad de *Hong Kong*; también se han convertido aulas virtuales con esta tecnología para escuelas y universidades de prestigio, ejemplos de ellos son *Harvard* y *New York University*. En un caso más particular la *University of Edinburgh* ha puesto un laboratorio de inteligencia artificial.

Otros institutos educacionales incluido *Princeton*, *Rice University*, *Pepperdine University*, *University of Derby* (UK) han implementado sus casas de estudio en *Second Life* para llevar a cabo sus clases y cursos de entrenamiento virtuales (Chen B. et al, 2010).

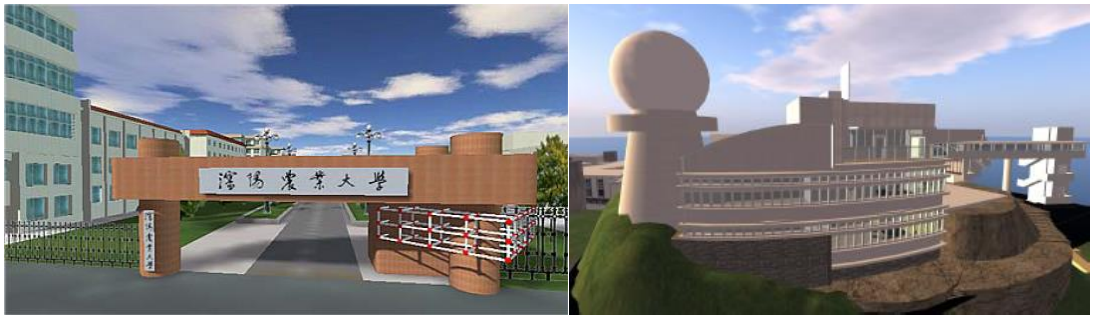


Figura 1.2: (Izquierda) Campus virtual de *Shenyang Agriculture University* y (derecha) el Campus virtual de *Chinese University of Hong Kong*

1.2. LIBGDX

LibGDX es un *framework* multiplataforma credo por Mario Zechner; el cual soporta *Windows*, *Linux*, *Mac OS X*, *Android*, *Blackberry*, IOS, y HTML5 como plataformas de destino (Reich, 2016). El lenguaje principal (pero no exclusivo) para el desarrollo de aplicación usando este *framework* es java. En vez de desplegar la aplicación de desarrollo en un dispositivo móvil o compilar a HTML5 para ejecutar el proyecto en un navegador web, *libGDX* permite trabajar en un ambiente de escritorio para agilizar el proceso de codificación (Reich, 2016).

"*LibGDX* es un *framework* y no un motor de juegos que usualmente cuenta con varias herramientas, tales como editor de niveles hecho y derecho, y un flujo de trabajo completamente predefinido. Esto pudiera sonar como una desventaja al principio, pero realmente resulta ser una ventaja que te permite libremente definir tu propio flujo de trabajo para cada proyecto." (Oehlke y Nair, 2015) De esta manera se diferencia *LibGDX* de otras tecnologías actualmente relevantes como lo es *Unity*, un motor de videojuegos multiplataforma. Aunque las funciones de alto nivel deberán ser suficiente para el desarrollo con este *framework*, *LibGDX* permite una interacción de bajo nivel dando la libertad de agregar llamadas personalizadas *OpenGL* (Oehlke y Nair, 2015).

Una tecnología importante perteneciente a las librerías de *LibGDX* es *OpenGL ES*, una API de programación de gráficos tridimensional, que está especialmente dirigido a

dispositivos móviles y embebidos. Mantenido por el grupo *Khronos*, una industria cuyos miembros incluye ATI, NVIDIA e Intel (Zechner y Green, 2012).

En la parte de gráficos de las librerías *LibGDX*, se ofrecen características como *render* a través de *OpenGL* ES 2.0 en todas las plataformas y ayuda de bajo nivel *OpenGL* como texturas, *meshes*, *shaders*, colecciones de vértices, etcétera. Las librerías de gráficos 3D ofrecen características como cámara de perspectiva, sistema de partículas, y librerías para *render* 3D con materiales y sistemas de luces (Oehlke y Nair, 2015).

Las librerías de matemáticas y física son extremadamente útiles, especialmente para trabajos como el presente, y cuentan con características como: clases de matriz, vector, *quaternion* y cajas delimitadoras. También existen otras librerías como las de audio, manejo de entrada y de utilidades.

Un *backend* es lo que permite a *LibGDX* tener acceso a las funcionalidades de plataforma correspondientes. Actualmente provee los siguientes *backends*: *Lightweight Java Game Library* (LWJGL), una librería de java para facilitar el desarrollo del juego en términos de tener acceso a los recursos del hardware en sistemas de escritorio; *WebGL*, este *backend* traslada código *Java* a *javaScript* y *SoundManager2*; *RoboVM* (IOS *backend*) y *Android* (Oehlke y Nair, 2015).

LibGDX provee una serie de módulos para poder tener acceso a varias partes del sistema, se pueden utilizar sin preocuparse por cuál es la plataforma de destino (The application framework, 2017). A continuación se describe la colección de módulos con la que cuenta *LibGDX*.

El módulo de aplicación provee acceso a la facilidad *logging*, a métodos para cerrar elegantemente la aplicación, persistir información, consulta a la versión API de *Android*, consulta al tipo de plataforma y consulta al uso de memoria. Por otra parte, el módulo de gráficos es usado para consultar el intervalo de tiempo entre el actual y el último *frame*, consultar tamaño de la pantalla y consultar el conteo de cuadros por segundo.

LibGDX también cuenta con el módulo de audio, el cual es usado para cargar sonidos en formato WAV, MP4, OGC y OGG (en caso de dispositivos IOS); el módulo de entrada,

encargado de la lectura de entrada de un teclado, mouse o *touch* (en caso de dispositivos IOS); el módulo de archivos, encargado del manejo de archivos externos e internos; y por último el módulo de red, encargado de consultas HTTP, creación de sockets y la funcionalidad de abrir una URI en un navegador web.

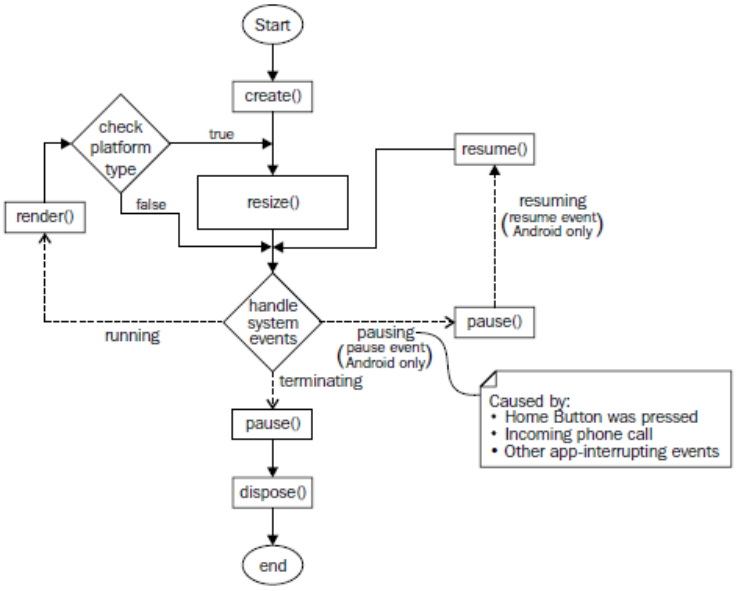


Figura 1.3: Ciclo de vida en *libGDX*.

Una aplicación *libGDX* cuenta con un ciclo de vida bien definido, será necesario que la clase que pretenda controlar este ciclo de vida tenga implementar la interfaz *ApplicationListener* (The life cycle, 2017).

El ciclo de vida cuenta con seis estados, el primero de ellos es *create*, el cual se llama al momento en que la aplicación es creada. El segundo método es *resize* y es llamado cada vez que la pantalla cambia su tamaño. El método *render* que por otro lado es invocado cada vez que el *renderizado* deba ser ejecutado, y es aquí debe residir la lógica de la aplicación. En un dispositivo móvil, cuando se presiona el botón Home o se recibe una llamada, se invoca al método pause, después de enfocar de nuevo la atención en la aplicación se invoca el método resume, y por último, cuando la aplicación es destruida se invoca al método *dispose*. El ciclo de vida con los métodos descritos se puede visualizar en la figura 1.3.

1.3. JUSTIFICACIÓN

Existen personas interesadas en estudiar una carrera relacionada a las tecnologías de la información, estas personas se encuentran investigando universidades que cuenten con este tipo de carreras, así como se encuentran investigando las materias en sus planes de estudio.

Muchos interesados en estudiar en la facultad de informática de la Universidad Autónoma de Querétaro viven en otros municipios, estados o inclusive otros países; distanciados de una visita rápida a la facultad para conocer sus instalaciones. El proyecto tratara de complementar esta búsqueda otorgándoles la facilidad de conocer las instalaciones de la facultad de informática por medio de un recorrido virtual que será de fácil acceso, sin importar la ubicación del usuario y sin costo alguno.

Debido a la aplicación, los interesados en estudiar una carrera a fin a las ciencias computacionales pueden motivarse y entusiasmarse en estudiar en la Facultad de Informática de la UAQ debido a que un alumno de dicha facultad realizó dicho proyecto.

Por último, se escoge el *framework LibGDX* por el contenido que se desea obtener en esta investigación, ya que *LibGDX* es una tecnología que obliga al desarrollador entender a plenitud su ciclo de vida y a conocer a detalle el proceso que conlleva la realización de una aplicación en tercera dimensión implementando un motor físico; a diferencia de motores de juego que cuenta con herramientas y flujos de trabajo predefinidos. También es importante mencionar que el uso de esta tecnología da como resultado un programa con una dinámica ingeniada por el desarrollador que puede ser presentado como un producto original. Dicho programa puede ser planteado con distintos propósitos, por ejemplo, si se deshará un recorrido en el centro universitario de la universidad autónoma de Querétaro, se podría utilizar el proyecto resultado de esta investigación, utilizando los modelos y texturas correspondientes al centro universitario pero sin hacer mayores modificaciones al código ya desarrollado.

1.4. OBJETIVO

El objetivo general de este trabajo de tesis es:

- Modelar tridimensionalmente el campus virtual de la Facultad de Informática a través del entorno de desarrollo integrando *Android Studio* complementado con librerías del

framework LibgGDX y planteado en el lenguaje de programación Java, para dar a conocer las instalaciones de la facultad por medio de un recorrido virtual de fácil acceso y sin costo alguno.

Los objetivos específicos de este trabajo de tesis son:

- Objetivo específico – Desarrollar objetos tridimensionales usando técnicas de modelación en tercera dimensión que asemejen a los inmuebles que conforman la facultad de informática de la Universidad Autónoma de Querétaro así como el mobiliario con el que cuenta.

- Objetivo específico – Crear e implementar texturas usando programas de diseño que asemejen las texturas de los inmuebles que conforman la facultad de informática de la Universidad Autónoma de Querétaro así como las texturas del mobiliario con el que cuenta.

- Objetivo específico – Utilizar los objetos tridimensionales que se crearon en un entorno de desarrollo integrado complementado con un *framework* que facilite manipular los objetos tridimensionales en un lenguaje de programación y bibliotecas que implementen un motor físico.

- Objetivo específico - Adaptar el programa a múltiples plataformas como lo son dispositivos móviles y computadoras personales de escritorio.

2. APLICACIÓN EN UN ENTORNO TRIDIMENSIONAL USANDO LIBGDX

Durante años, artistas han engañado al ojo humano haciéndolo percibir una imagen plana de dos dimensiones en una escena de tercera dimensión. Uno de los trucos usado es llamado proyección linear y funciona juntando líneas paralelas hacía un punto de fuga, para crear la ilusión de la perspectiva (Brothaler, 2013). En la Figura 2.1 se puede apreciar un ejemplo gráfico de esta definición, donde el camino se hace más pequeño a medida que se va alejando del visor; este tipo de trucos es necesarios para crear una escena de tercera dimensión.

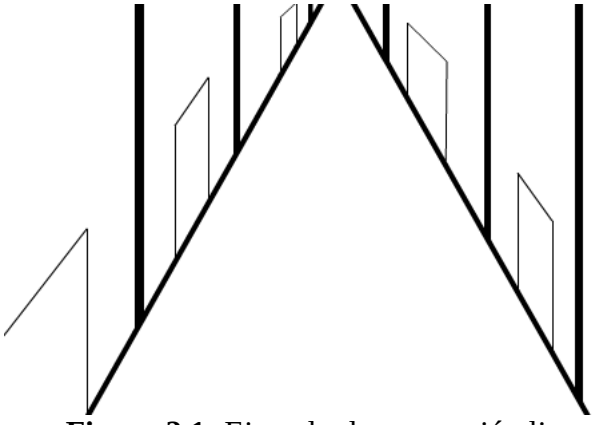


Figura 2.1: Ejemplo de proyección linear.

La variable *gl_Position* es un vector de cuatro componentes (Wright, 2011), con esta variable *OpenGL* comienza el proceso conocido como división de perspectiva, en el cual las coordenadas visibles "mentirán" en el rango de [-1, 1] para los componentes X, Y y Z, esto a pesar del tamaño o figura del área de *renderizado*. Para esto se utilizarán las coordenadas normalizadas del dispositivo que son coordenadas en los rangos [-1, 1] y son independientes del actual tamaño o figura de la pantalla (Brothaler, 2013).

Para crear la ilusión de tercera dimensión en la pantalla, *OpenGL* utiliza cada *gl_Position* y divide los componente X, Y y Z por el componente W. El componente W es usado para representar la distancia, lo cuál causa que objetos posicionados a una distancia lejana se muevan cerca del centro del área de *renderizado,* actuando como un punto de fuga (Brothaler, 2013). En la figura 2.2 se aprecia de manera gráfica lo anteriormente explicado, en donde cada punto en rojo representa un vértice cuyo componente W va incrementando de

uno en uno y puntos con un número mayor en su componente W se irán acercando más a la coordenada [0].

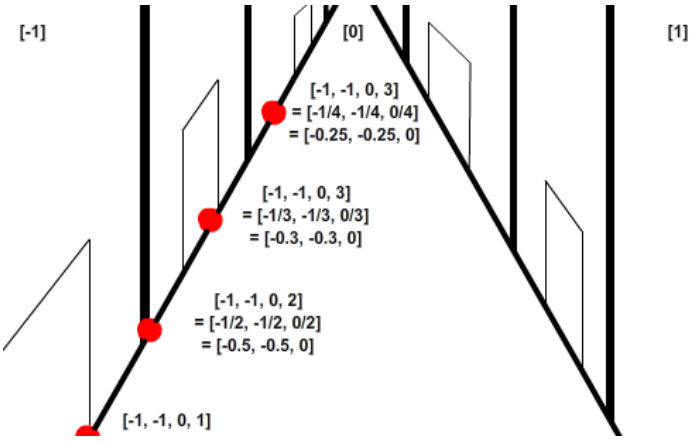


Figura 2.2: Ejemplo del proceso conocido como división de perspectiva

Por último, *OpenGL* necesita mapear los componentes X y Y de las coordenadas normalizadas del dispositivo a un área en la pantalla que el sistema operativo a destinado para su visualización conocido como *viewport*, estas coordenadas mapeadas son conocidas como coordenadas de ventana o *window coordinates* (Brothaler, 2013). El proceso anteriormente explicado puede visualizarse más cómodamente en la figura 2.3.

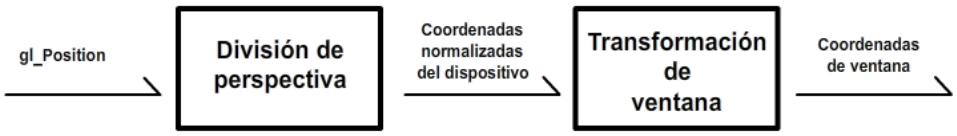


Figura 2.3: Proceso de transformación de coordenadas.

2.2. PROYECCIÓN DE PERSPECTIVA

Hay dos tipos de cámaras: ortogonal y de perspectiva. En un ambiente tridimensional se hace uso de la cámara de perspectiva (Oehlke y Nair, 2015). Las cámaras ortogonal y de perspectiva heredan de la clase base Camera, por lo que se heredan sus campos y métodos; entre los métodos heredados se encuentran *lookAt*, *rotate*, *transform* y *translate*.

Haciendo uso de una proyección ortogonal, no importa si un objeto está posicionado lejos o cerca de la cámara, siempre se tendrá el mismo tamaño en pantalla (Zechner y Green, 2012). Solamente la lejanía de un objeto se puede percibir al hacer uso de una proyección de perspectiva.

En la creación de una cámara con proyección de perspectiva se especifica el atributo *fieldOfViewY*, que se refiere al campo de visión de la cámara dado en grados; esto ayuda a especificar la cantidad de acercamiento que el desarrollador desea. Se recomienda pensar en una lente de cámara, donde 90^0 proporciona un amplio panorama y 30^0 un profundo acercamiento. Los demás parámetros especifican el ancho (*viewportWidth*) y alto (*viewportHeight*) de la ventana.

La clase Camera cuenta con dos atributos importantes, el atributo *near* indica la cantidad de acercamiento y la variable *far* indica la cantidad de alejamiento.

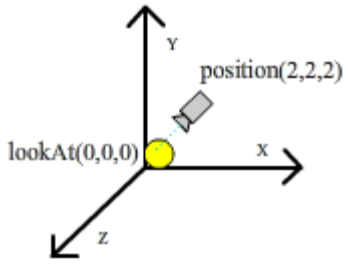


Figura 2.4: Cámara con proyección de perspectiva

Es importante asignar una posición y dirección a la cámara; en la figura 2.4 se puede apreciar el resultado obtenido respecto al código 2.1.

```
Public PerspectiveCamera cam;

cam = new PerspectiveCamera(67,Gdx.graphics.getWidth(),Gdx.
    ↪ graphics.getHeight());
cam.position.set(2, 2, 2);
cam.lookAt(0,0,0);
cam.near = 1f;
cam.far = 300f;
cam.update();;
```

Código 2.1: Configuración de una cámara de perspectiva

2.3. MODELOS PRIMITIVOS EN LIBGDX

Un modelo representa un recurso tridimensional y guarda una jerarquía de nodos, y un nodo es una combinación de geometría (*mesh*) y de material. Opcionalmente un modelo puede contener información de animación o *skinning*, estos conceptos no se abordarán en el presente trabajo.

Un modelo no está destinado a ser *renderizado*; si se desea generar la imagen del modelo, se utilizará la clase *ModelInstance*. La estructura de la clase *ModelInstance* es básicamente la misma que la estructura de la clase *Model*, con transformaciones de cambio de posición adicionales. También se permite la modificación de materiales y nodos sin destruir al modelo original. El modelo original es dueño de sus mallas y texturas; la instancia del modelo que le asigne el desarrollador comparte estos recursos (Oehlke y Nair, 2015).

Paquetes de software prominentes como *Blender*, *3DS Max*, *ZBrush* y *Wings 3D* proveen al usuario con grandes cantidades de funcionalidades para crear objetos 3D (Zechner y Green, 2012], en el presente trabajo se usará el software de desarrollo de objetos 3D *Modo* y se explicará a detalle el proceso de creación de un objeto. Pero para efectos de demostración y prueba, en el presente capítulo se explicará la creación de un simple programa que dibujará una colección de modelos primitivos usando únicamente las librerías de *LibGDX*.

2.4. AMBIENTES, MATERIALES Y LUCES

Las clases *Environment*, *Material* y *Lights* extienden de la clase *Attributes*, una clase usada para especificar valores uniformes (Material and environment, 2015).

El uso de la clase Material es para referir a las propiedades de un objeto que determina como interactuar con luces (Oehlke y Nair, 2015).

La clase *Environment* contiene valores uniformes especificando una locación, las luces son partes de esta clase (Material and environment, 2015). Existen diferentes tipos de luces: luz ambiental, luz direccional, luz de punto, y luz concentrada. En el presente trabajo se hará uso únicamente de la luz direccional, la cual actúa como un sol (Oehlke y Nair, 2015).

Una clase clave que obligadamente se incluirá en el método *render* es *ModelBatch*, usada para *renderizar* la instancia del modelo (Oehlke y Nair, 2015) recuperando, ordenando y *renderizando shaders* con ayuda de la interfaz *ShaderProvider*. Son conocidos como *shaders* las pequeñas subrutinas que le dicen a la unidad de procesador gráfico o GPU que dibujar (Brothaler, 2013). *ModelBatch* es un objeto relativamente pesado debido a los *shaders* que pueden crear; si existe la posibilidad, se recomienda rehusarlo y es por eso que típicamente se inicializa en el método *create* (Jastrzebski, 2016).

```
public Environment environment;
public ModelBatch modelBatch;

public void create () {
    modelBatch = new ModelBatch();
    environment = new Environment();
    environment.set(new ColorAttribute(ColorAttribute.
        ↪ AmbientLight, 0.4f, 0.4f, 0.4f, 1f));
    environment.add(new DirectionalLight().set(0.8f, 0.8f, 0.8f
        ↪ , -1f, -0.8f, -0.2f));

    //configuración de la cámara
}
```

Código 2.2: Ambiente tridimensional

En el código 2.2 se puede observar la integración del ambiente, configuración de luces e inicialización de una variable de tipo *ModelBatch*.

Por último se utilizará la clase *ModelInstance*, aquí se almacenarán las instancias de los modelos primitivos. La clase *ModelInstance* se define como una instancia de un modelo que permite especificar transformaciones globales y modificar materiales, como si se tuviera una copia de los materiales del modelo. Múltiples instancias pueden ser creadas del mismo modelo y todas compartiendo mallas y texturas, ya que el modelo es dueño de estas (Zechner, 2017).

Se presenta en el código 2.3 una función encargada de crear un modelo introduciendo las coordenadas en un espacio tridimensional y regresando un objeto *ModelInstance* diferente que se irá agregando a una colección de instancias del modelo (Material and environment, 2015).

```
public ModelInstance addSphere(float x, float y, float z)
{
    ModelInstance instance;
    ModelBuilder modelBuilder = new ModelBuilder();
    Model model = modelBuilder.createSphere(2, 2, 2, 20, 20,
          new Material(ColorAttribute.createDiffuse(Color.
              ↪ YELLOW)),
          VertexAttributes.Usage.Position | VertexAttributes.
              ↪ Usage.Normal);
    instance = new ModelInstance(model);
    instance.transform.trn(new Vector3(x, y, z));
    return instance;
}
```

Código 2.3: Función *addSphere*.

Dentro de la función *addSphere* se crea e inicializa la variable *modelBuilder* de tipo *ModelBuilder*, que será la clase que permitirá crear figuras básicas. En el presente programa, las figuras primitivas que se presentarán son: esfera, caja, cono, cilindro y cápsula.

ModelBuilder cuenta con una serie de métodos, cada uno correspondiente a una figura primitiva distinta; para la creación de una esfera se hace uso del método *createSphere* en el cual se tiene que especificar el ancho, alto y profundidad del objeto, así como las divisiones en las coordenadas U y V. También se asigna una instancia de *Material* con la clase *ColorAttribute* que permite pasar un color al *shader*. Se requiere especificar el tipo de atributo, como lo pueden ser *Diffuse*, *Specular*, *Ambient*, *Emissive*, *Reflection*, *AmbientLight* y *Fog*; se usará el tipo más simple para este programa de prueba, *Diffuse*. Por último, el atributo *Usage.Normal* define las normales para que la luz pueda ser aplicada (Bose, 2014).

```
public void create () {

    ...

  modelsInstance = new Array<ModelInstance>();
  modelsInstance.add(addSphere(-4,0,0));
  modelsInstance.add(addBox(0, 0, 0));
  modelsInstance.add(addCapsule(-8, 0, 0));
  modelsInstance.add(addCone(4, 0, 0));
  modelsInstance.add(addCylinder(8,0,0));
}
```

Código 2.4: Colección de instancias de modelo.

Se crea una serie de funciones con la misma rutina presentada en la función *addSphere* pero utilizando diferentes funciones que dan como resultado, figuras diferentes. En el código 2.4 se muestra la creación de diferentes figuras geométricas especificando coordenadas que se refieren a su posición dentro de un espacio tridimensional, para efectos de demostración se colocan en una hilera únicamente cambiando su posición en el eje X.

```
public void render () {
  Gdx.gl.glViewport(0, 0, Gdx.graphics.getWidth(), Gdx.
      ↪ graphics.getHeight());
  Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT | GL20.
      ↪ GL_DEPTH_BUFFER_BIT);

  modelBatch.begin(cam);
  for (ModelInstance instance : modelsInstance) {
     modelBatch.render(instance, environment);
  }
  modelBatch.end();
}
```

Código 2.5: Método *render*

De acuerdo al código 2.5, dentro del método *render*, las primeras líneas llaman a métodos de *OpenGL* de bajo nivel. Para lograr una interacción de bajo nivel, se hará uso de la interfaz *GL20* contenida en la clase *Gdx*; *GL20* contiene todos los métodos de *OpenGL ES 2.0*.

GlViewport es un método que especifica el tamaño de la superficie que tiene disponible para *renderizar* (Brothaler, 2013), como un rectángulo. Los primero dos parámetros a ingresar en el método *glViewport* son X y Y respectivamente, partiendo de la esquina inferior izquierda del rectángulo; los siguientes parámetros son el ancho y el alto, proporcionando las dimensiones del rectángulo. Se hace uso de la interfaz *Graphics*, la cual mantiene comunicación con el procesador gráfico y por lo tanto se puede hacer uso de métodos como *getDeltaTime*, *getDensity*, *getFramesPerSecond*, *getHeight*, *getWidth*, etcétera. Se utilizan los métodos *getHeight* y *getWidth* para asignarle el alto y el ancho al método *glViewport*.

GlClear es un método que limpia buffers por medio de operadores bit por bit o *Bitewise* de máscaras. Las tres máscaras las cuales se pueden utilizar en este método son: *GL_COLOR_BUFFER_BIT*, *GL_DEPTH_BUFFER_BIT* y *GL_STENCIL_BUFFER_BIT* (Khronos Group, s.f.).

GL_DEPTH_BUFFER_BIT es una máscara de suma importancia que limpiará el buffer de profundidad o *depth buffer*. Si no se contara con un *depth buffer*, ocurriría una serie de problemas al momento de *renderizar* correctamente figuras que están en diferentes

posiciones dentro de un plano tridimensional pero coinciden en la vista del usuario; particularmente no se dibujarían correctamente las partes en que una figura sobrepone otra.

El problema puede ejemplificarse en un cubo formado de triángulos cuyos lados poseen un color diferente cada uno. En la parte A dentro de la figura 2.5 puede apreciarse los lados de los cubos representados en un plano de dos dimensiones. En la parte B de la figura 2.5, se muestra el caso en que no se utilizara la máscara *GL_DEPTH_BUFFER_BIT*, se puede notar los lados que no deberían de visualizarse según la perspectiva de la imagen parecen ignorar la profundidad de las caras del objeto, en este caso, el lado inferior coloreado en grises aparece en el *renderizado* final de la imagen, lo mismo puede ocurrir con otros lados destinados a ocultarse; no se puede predecir la proyección final, sin embargo la figura B únicamente trata de explicar los problemas que puede ocasionar la carencia de la máscara *GL_DEPTH_BUFFER_BIT*. En la parte C se tiene el cubo proyectado de la manera correcta y utilizando la etiqueta *GL_DEPTH_BUFFER_BIT*, resolviendo de manera idónea la profundidad de los lados del cubo y ocultando aquellos que no deban visualizarse según la perspectiva del usuario.

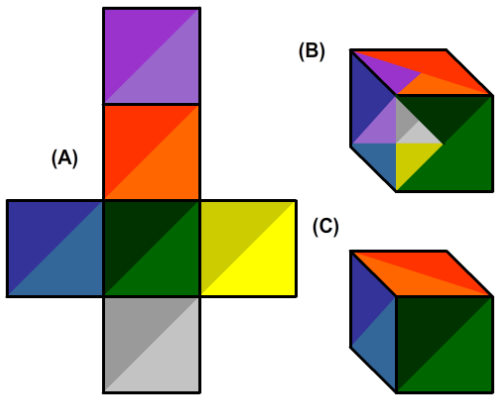


Figura 2.5: Aplicación de la máscara *GL_DEPTH_BUFFER_BIT*

Depth testing es una efectiva técnica para remover superficies escondidas. Cuando se dibuja un pixel, se le asigna un valor llamada valor Z que detona una distancia entre la perspectiva del usuario y el pixel. Después, cuando otro pixel necesita ser dibujado en esa locación de pantalla, el nuevo valor Z del pixel es comparado con el pixel anteriormente colocado ahí. Si el nuevo valor Z es más alto, significa que es más cercano a la vista del usuario y el anterior pixel es obscurecido por el nuevo. Esta maniobra es conseguida

internamente por el *depth buffer* con almacenamiento para valores de profundidad por cada pixel en la pantalla (Wright, 2011).

La clase *ModelBatch* maneja las llamadas *render* y es típicamente usado para *renderizar* instancias de modelos (Jastrzebski, 2016), y esto será el principal uso de esta clase. Dicha implementación ayuda a que el programador evite generar código para *renderizar* objetos y se concentre más en la parte lógica de la programación.

Render es un proceso el cual debe hacerse cada *frame*; se utiliza el método *begin* y *end* al principio y al final de *render* respectivamente, esta rutina dibujará el entorno tridimensional. Para hacer una llamada *render*, *LibGDX* se auxilia con la clase *Renderable*, la cual contiene todo lo necesario para efectuar la llamada.

Al invocar al método *render* de la clase *ModelBatch*, se provee el ambiente junto con la instancia de modelo; en este caso se cuenta con más de una instancia, por lo tanto se cicla de acuerdo a las instancias contenidas en la colección *modelsInstance*. En la figura 2.6 se muestra el programa en ejecución desplegando con éxito las figuras primitivas que se crearon.

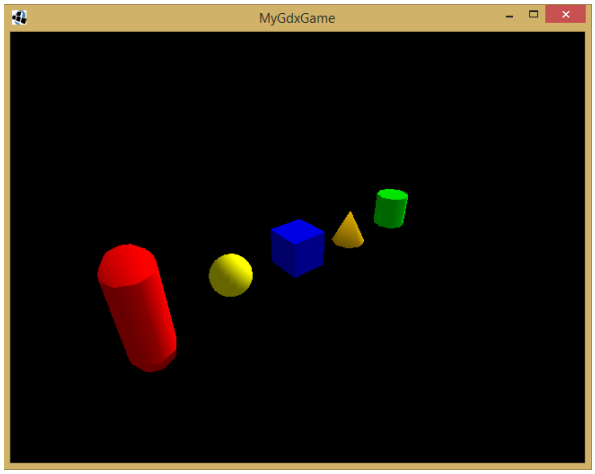


Figura 2.6: Figuras primitivas tridimensionales

3. MOTOR FÍSICO

"La física en juegos ha estado desde los primeros videojuegos programados. Fue primero visto en el movimiento de partículas: moscas, fuegos artificiales, balística, humo y explosiones" (Millington, 2007). Desde juegos primitivos se fue mejorando los motores físicos y fueron evolucionando, por ende es una parte muy importante en videojuegos y simuladores. Un motor físico es un pedazo de código que conoce varios conceptos generales de física pero no está programado para un escenario específico (Millington, 2007). Es importante mencionar que esto facilita la reusabilidad y hace de un motor físico, una tecnología portable entre videojuegos o simuladores con múltiples fines.

Existe el caso de *Damage Incorporated* publicado por *MacSoft*, que obtuvo la licencia para una tecnología creada por *Bungie Software* para sus productos *Marathon* y *Marathon 2*, entre 1994 y 1995 era considerada una tecnología sofisticada. Contaba con un simple pero eficiente modelo físico (Rouse, 2005), atributo que llevo a *MacSoft* a abrazar esta tecnología. Un desarrollador, siendo usuario de productos ya sean videojuegos o simuladores, puede encontrar en un producto, un motor físico interesante y poderoso, y si lo decide, puede conseguir las herramientas de desarrollo para implementar esta tecnología sin necesidad de darse a la laboriosa tarea de desarrollar uno propio que trate de igualar a la tecnología que fue de su interés.

En el videojuego *Half-Life* se contaba con la opción de mover cajas, aunque la física contaba con varias fallas y el movimiento de dichas cajas en momentos era confusa. Cuando salió la secuela de este videojuego, *Half-Life 2*, los desarrolladores crearon una física que reemplazo la pasada, esto abrió toda serie de nuevas oportunidades; las piezas de una caja de madera rota flotando en el agua, objetos que podían atorarse o usarse como "escudos movibles" y varios aditamentos más (Millington, 2007). En los casos de *Damage Incorporated* y de *Hal-Life*, se demuestra que las consideraciones del motor físico influyen radicalmente en el producto final. El resultado entre un producto estable o un producto errante puede decidirse en la implementación o desarrollo de un motor físico.

La velocidad puede llegar a ser un problema a la hora de implementar un motor físico robusto. Puede que en consolas modernas o computadoras personales potentes, el motor

físico no sea un problema en lo absoluto, pero puede ser significativo en dispositivos de mano como los celulares. Si un proyecto tiene la ambición de ser multiplataforma, es importante que el desarrollador sea selectivo en que motor físico se implementará al proyecto, en el caso de que solo se requiera simular un solo comportamiento físico en lugar de implementar un motor físico completo, se recomienda programar las entidades necesarias para simular esta pequeña parte, de esta manera el programa contará con un rendimiento más fluido.

A continuación se presentará la realización de un programa que simule el comportamiento de una esfera cayendo por la fuerza de la gravedad. Como principio, todo motor físico está basado en las leyes de newton que describen con gran precisión como un objeto (sin considerar su tamaño y forma) se comporta. En el programa se llamarán a estos objetos partículas, es importante aclarar que la lectura no se refiere a las partículas subatómicas como electrones o protones; en el programa, una partícula poseerá una posición más no una orientación, no importará a que dirección esté apuntando, lo único que interesará es la dirección en la que viajará (Millington, 2007). De esta manera se pretende crear un programa bastante sencillo que servirá para explicar un principio básico en los motores de física. A continuación un repaso a las dos primeras leyes de Newton.

Primera ley o ley de la inercia – Todo cuerpo persevera en su estado de reposo o movimiento uniforme y rectilíneo a no ser que sea obligado a cambiar su estado por fuerzas impresas sobre él.

Segunda ley o principio fundamental de la dinámica – El cambio de movimiento es directamente proporcional a la fuerza motriz impresa y ocurre según la línea recta a lo largo de la cual aquella fuerza se imprime.

3.1. DESARROLLO DE UN MOTOR FÍSICO

Se empezará creando una esfera de color amarillo.

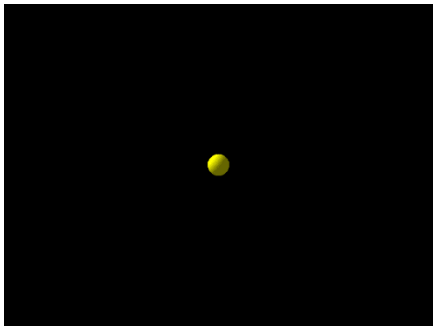


Figura 3.1: Esfera amarilla creada en *LibGDX*

En la Figura 3.1 se muestra una esfera, la cual se tomará como partícula. Algunos conceptos como tiempo, temperatura, masa y densidad se pueden describir con un número y una unidad, a este número se le conoce como cantidad escalar; otras cantidades están asociadas con una dirección y no pueden describirse con un solo número, por ejemplo, para describir plenamente una fuerza hay que indicar no sólo su intensidad, sino también en qué dirección tira y empuja, a esta serie de datos se le conocen como cantidad vectorial que tienen tanto una magnitud como una dirección en el espacio (Young et al, 2009). En el presente programa, la posición, velocidad y aceleración serán consideradas como vectores. Entre las clases que ofrece *Libgdx* se encuentra la clase *Vector3* que encapsula un vector en tercera dimensión, esta clase posee una serie de métodos que ejecutan operaciones vectoriales ya sean sumas, restas o multiplicaciones de vectores, entre otras de más complejidad. La mencionada clase *Vector3*, será el tipo de dato de las variables *position*, *velocity* y *acceleration*, estas variables conformarán la clase *Particle*.

```
public class Particle {
    Vector3 position;
    Vector3 velocity;
    Vector3 forceAccum = new Vector3(0f,0f,0f);
    Vector3 acceleration;
    float mass;
    float inverseMass;
    float friccion;

    public Particle(){}

    public void setMass(float mass){ this.mass = mass;
        ↪ inverseMass = (1.0f/mass); }
    public void setVelocity(Vector3 velocity){ this.velocity =
        ↪  velocity; }
    public void setAcceleration(Vector3 acceleration){ this.
        ↪ acceleration = acceleration; }
    public void setPosition(Vector3 position){ this.position =
        ↪  position; }
    public void setFriccion(float friccion) { this.friccion =
        ↪ friccion; }

    public void addForce(Vector3 force){ forceAccum.add(force)
        ↪ ; }

  public void clearAccum(){ forceAccum = new Vector3(0f,0f,0f)
      ↪ ; }

}
```

Código 3.1: Clase *Particle*

Al contar con un valor en la variable masa se tendrá una repercusión en el movimiento de la partícula actuando la fuerza de gravedad sobre esta. Se usará el principio de *D'Alembert's* que implica lo siguiente; si se tiene una colección de fuerzas actuando sobre un objeto, se puede reemplazar todas estas fuerzas con una sola fuerza (Millington, 2007).

Se declaró la variable *forceAccum* en la clase *Particle* siendo del tipo *Vector3*, esta variable servirá para la implementación del principio de *D'Alembert's* convirtiéndose en una variable acumulativa de la fuerza neta ejercita en la partícula.

El método *addForce* perteneciente a la clase *Particle* almacenará la sumatoria de las fuerzas que se agreguen en el método créate y el método *clearAccum* es usado para limpiar la variable *forceAccum*, este último método se llamará cada vez que la posición se refresque.

En el presente programa se agregará una fuerza parecida al de la gravedad de la tierra, el resultado deberá ser una esfera cayendo.

Será momento de aplicar la segunda ley de Newton en la cual el cambio de movimiento es directamente proporcional a la fuerza motriz impresa (Young et al., 2009); entonces, dependiendo de la fuerza impresa en la esfera se cambiará su velocidad y aceleración.

El cociente de la fuerza neta entre la aceleración constante es igual a la masa (Young et al., 2009). "La masa es una medida cuantitativa de la inercia... cuanto mayor sea su masa, más se resiste un cuerpo a ser acelerado" (Young et al., 2009).

(3.1)

a) $$m = \frac{f}{a}$$

b) $$f = ma$$

c) $$a = \frac{1}{m} f$$

"La velocidad total se representa gráficamente con el área total bajo la curva *ax*-*t* entre las líneas verticales *t1* y *t2*" (Young et al., 2009). En el límite donde los intervalos de *t* se hacen muy pequeños y numerosos, el valor de la aceleración media se acerca a la aceleración instantánea *ax* en el instante *t*. El área bajo la curva *ax*-*t* es la integral de *ax* en función de *t*. Por tanto se tiene la fórmula.

(3.2)
$$v_2 - v_1 = \int_{t_1}^{t_2} a_x dt$$

Y suponiendo que *t1* = 0, se tiene la siguiente fórmula.

(3.3)
$$v_x = v_0 + \int_0^t a_x dt$$

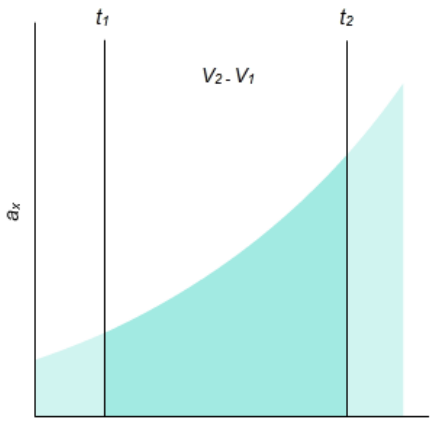


Figura 3.2: Gráfica *ax-t*

La Figura 3.2 es una gráfica *ax-t* en la cual se representa gráficamente el desplazamiento de la velocidad. Ahora supóngase que se desea conocer la posición y siguiendo una línea de pensamiento similar a la que se aplicó anteriormente, se tiene que "el cambio de posición x (es decir, el desplazamiento) es la integral en el tiempo de velocidad *vx*" (Young et al., 2009).

(3.4)
$$x_2 - x_1 = \int_{t1}^{t2} v_x dx$$

Y suponiendo que *t1* = 0, se tiene la siguiente fórmula.

(3.5)
$$x = x_0 + \int_0^t v_x dt$$

Dado a la fórmula 3.3 se puede calcular la velocidad conociendo la aceleración (la aceleración se definió en el inciso c de la fórmula 3.1), y también dado a la fórmula 3.5 se puede calcular la posición conociendo la velocidad, el resultado será regresado al llamar la función *positionUpdate* en un tipo de dato *Vector3*. Lo único que requirió la función para efectuar la serie de cálculos fue el lapso de tiempo *deltaTime*.

```
public Vector3 positionUpdate(float deltaTime){
        position.mulAdd(velocity, deltaTime);
        Vector3 resultAcc = acceleration;
        resultAcc.mulAdd(forceAccum, inverseMass);
        velocity.mulAdd(resultAcc, deltaTime);
        position.scl((float)Math.pow(friccion, deltaTime));
        clearAccum();
        return position;
    }
```

Código 3.2: Método *positionUpdate*

Para integrar la fricción, en teoría bastaría multiplicar la posición por la variable fricción; pero si el *frame rate* del programa incrementa, entonces habrá más actualizaciones y por tanto la fricción se ejercerá más. Para solucionar este detalle, la variable fricción se eleva a la *t* potencia. El Código 3.2 muestra la integración de la fricción al programa.

Los resultados finales del programa se pueden apreciar en la Figura 3.3. En estos resultados se percibe la caída de la esfera amarilla a una velocidad a la vista normal; tal es su comportamiento ya que la masa que se asignó tiene un valor de 2, si se aumentara la masa, la esfera se resistiría más en su caída, pero si se disminuyera la masa, la caída de la esfera sería más instantánea.

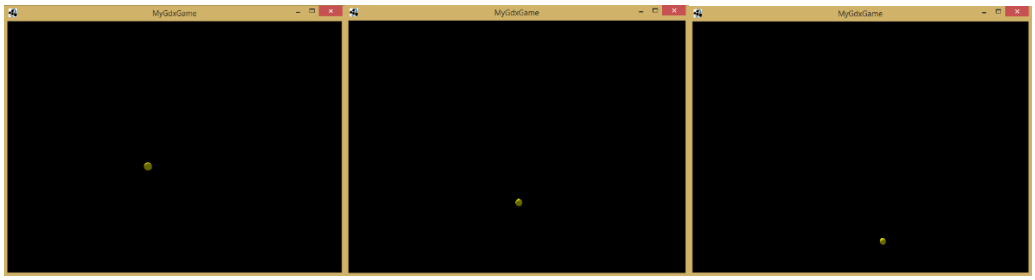


Figura 3.3: Programa en ejecución aplicando el principio de *D'Alembert's*.

Se desarrolló con éxito un simple programa donde se asignaron propiedades a una partícula, la cual manipuló la posición de una esfera amarilla y simuló de manera eficaz el comportamiento que ha de tener una bala con las propiedades que se asignaron.

En el programa se trabajó únicamente con la física de una partícula, es decir, importando su posición más no su orientación; pero en un motor robusto de física deben de considerarse los cuerpos rígidos, aquellos cuerpos que son más complejos, y al momento de codificar su movimiento rotacional resulta ser una labor más extenuante. No es alcance de

este trabajo seguir el desarrollo de un motor físico propio, ya que siendo basto e interesante, el contenido que se abordará en la presente investigación será variado y para los alcances del producto final se optó por utilizar la tecnología *Bullet*. Si bien se utilizará el motor físico *Bullet*, es importante entender como están desarrollados este tipo de tecnologías por medio del programa presentado implementando un primitivo motor físico de partículas.

3.2. FÍSICA BULLET

Bullet Physics es una librería escrita en C++ portable de código abierto enfocada a detección de colisiones, dinámicas de cuerpos rígidos y dinámicas de cuerpos suaves (Coumans, 2015). *LibGDX* cuenta con un envoltorio o *wrapper* para *Bullet* y la manera de integrarlo es únicamente seleccionando la extensión *Bullet* en el generador de proyecto *LibGDX*.

En el presente capítulo se explicará *Bullet* con un simple programa en el cual una pelota cae y colisiona con un rectángulo plano. Dicho programa será programado con objetos primitivos, aunque en capítulos posteriores se implementará *Bullet* a modelos complejos.

Para hacer uso de la librería *Bullet* es necesario cargarla a la memoria, por ello es necesario llamar a la función *Bullet.init* en el método *create* (Nair y Oehlke, 2015).

```
public void create() {
//Código
   Bullet.init();
}
```

Código 3.3: Inicialización de *Bullet*.

Contando con las librerías *Bullet* cargadas, se procede a agregar un mundo dinámico discreto, el cual provee una interfaz de alto nivel que maneja los objetos físicos y las restricciones, también implementa actualizaciones de todos los objetos en cada *frame* (Coumans, 2015). Para alimentar el constructor de la clase *btDynamicsWorld*, se necesitará inicializar un par de variables primero.

Bullet utiliza dos métodos *phase* o de fase, el primero encuentra objetos de colisión cercanos uno del otro, a este método se le conoce como *broad phase* o fase general y para dicha fase se utiliza la clase *btDbvtBroadphase*. El algoritmo del segundo método es más

especializado y preciso ya que ocurre después de la detección hecha por la primera fase, a este segundo método se le conoce como *near phase* o fase cercana y para esta fase se utiliza la clase *Dispatcher*, la cual es alimentada por la clase *btDefaultCollisionConfiguration* (Nair y Oehlke, 2015). Dicho manejo de fases es usado para optimizar la ejecución de la simulación, ya que si en un principio se usará un algoritmo preciso de colisión (como el algoritmo de *near phase*) en cada *frame*, y en caso de que existieran cientos de objetos en el programa, el CPU cargaría un enorme trabajo y el rendimiento decrecería (Dickinson, 2013).

El solucionador de restricciones es usado para adjuntar objetos entre sí y utiliza la clase *btSequentialImpulseConstraintSolver*. Por último, las variables inicializadas se colocan en el constructor de *btDiscreteDynamicsWorld* y se indica la gravedad que se usará en el programa. En el Código 3.4 se pueden apreciar dichas implementaciones.

```
private btDefaultCollisionConfiguration collisionConfiguration
    ↪ ;
private btCollisionDispatcher dispatcher;
private btDbvtBroadphase broadphase;
private btSequentialImpulseConstraintSolver solver;
private btDiscreteDynamicsWorld world;

public void create() {
//Código
collisionConfiguration = new btDefaultCollisionConfiguration()
    ↪ ;
dispatcher = new btCollisionDispatcher(collisionConfiguration)
    ↪ ;
broadphase = new btDbvtBroadphase();
solver = new btSequentialImpulseConstraintSolver();
world = new btDiscreteDynamicsWorld(dispatcher, broadphase,
    ↪ solver, collisionConfiguration);
world.setGravity(new Vector3(0, -9.8f, 1f));
}
```

Código 3.4: Configuración de la física *Bullet*.

El siguiente paso será crear las figuras de colisión o *CollisionShapes*, estas figuras permiten colisionar una gran variedad de diferentes objetos, no tienen una posición, masa, inercia, restitución, y otras propiedades por el estilo, ya que son figuras exclusivas para colisiones. Las figuras de colisión están adjuntos a objetos de colisión o cuerpos rígidos (Collision Shapes. s.f.); en la estructura de datos de detección de colisiones existen los objetos de colisión representados por la clase *btCollisionObject*, un objeto que posee una

transformación en el mundo y una figura de colisión (Coumans, 2015), también existen objetos fantasmas representado por la clase *btGhostObject*, el cual es un objeto de colisión especial. Un concepto importante es el mundo de colisión representado por la clase *btCollisionWorld*, donde se guardan los objetos de colisión.

Parecido a las figuras primitivas que se crearon en el capítulo anterior, las figuras de colision pueden ser primitivas, existe una gran variedad de figuras primitivas como lo son: esferas, cajas, cilindros, capsulas, conos y multi-esferas. También existen figuras más complejas compuestas de triángulos como la clase *btConvexHullShape*, definida por una nube de vértices, aunque la figura formada es la figura convexa más pequeña que encierra los vértices. Otra figura compleja es manejada por la clase *btBvhTriangleMeshShape*, un *mesh* estático que está comprendido de triángulos arbitrarios (Collision Shapes. s.f.). Si en algún escenario se tuvieran varios cuerpos rígidos con la misma figura, una manera de ahorrar memoria sería asignando la misma figura de colisión a cada uno de los cuerpos rígidos. En la primera línea del código 3.3 se crea la figura de colisión *btBoxShape*, creando una caja con un ancho y un alto, coincidiendo con las dimensiones el modelo de instancia *groundInstance.* Dicho código se encuentra en el método *create.*

"Los cuerpos rígidos son los bloques básicos de construcción de todas las simulaciones físicas. Son representados por la clase *btRigidBody*" (Rigid Bodies. s.f.), a los cuerpos rígidos se les asigna fuerzas, masa, inercia, velocidad y restricciones.

Existen tres tipos diferentes de cuerpos rígidos en *Bullet*: Cuerpos rígidos dinámicos, los cuales poseen una masa positiva y requieren de una clase *MotionState* que optimiza las calculaciones físicas usando *callbacks* cuando un cambio ocurre (Bose, 2014). Otro tipo de cuerpo rígido son los estáticos, cuya masa es cero y no pueden moverse, únicamente colisionar. Por último, existen los cuerpos rígidos cinemático cuya masa es cero y puede ser animada por el usuario, ideal para el personaje principal en el presente recorrido.

```
btCollisionShape groundshape = new btBoxShape(new Vector3
    ↪ (20,1/2f,20));
btRigidBody.btRigidBodyConstructionInfo bodyInfo = new
    ↪ btRigidBody.btRigidBodyConstructionInfo(0,null,
    ↪ groundshape,Vector3.Zero);
btRigidBody body = new btRigidBody(bodyInfo);

world.addRigidBody(body);
```

Código 3.5: Creación un cuerpo rígido con la figura de colisión de una caja.

Aparte de necesitar la figura de colisión para crear un cuerpo rígido, se necesitará el objeto *btRigidBodyConstructionInfo*, dicho objeto provee información para crear al cuerpo rígido; en el Código 3.5 se observa que la variable *bodyinfo* de tipo *btRigidBodyConstructionInfo* se inicializa con una masa de cero y sin objeto *MotionState* convirtiendo al cuerpo rígido en un cuerpo rígido estático. Contando con la variable *bodyinfo* preparada, se coloca en el constructor la clase *btRigidBody* de la variable *body*, la cual será el cuerpo rígido del suelo. Por último se agrega el cuerpo rígido creado al mundo dinámico con la función *addRigidBody*.

En *Bullet* el cambio de posición se maneja de manera muy sencilla con la clase *MotionState*. "El trabajo de los estados de movimiento es catalogar la actual posición y orientación del objeto" (Dickinson, 2013). Se usará la clase *btDefaultMotionState* para el manejo de los estados de movimientos y se tendrá que inicializar dicha clase con la matriz de transformación de la instancia de modelo *sphereInstance*. En el código 3.4 se muestra la inicialización de la clase *btDefaultMotionState* y enseguida se invoca al método *setWorldTransform*, que es llamado por *Bullet* cuando se ha transformado el objeto dinámico. Por último, se crea un cuerpo rígido para la figura esférica y a diferencia del suelo, se agrega el estado de movimiento *sphereMotionState* en el constructor de *btRigidBodyConstructionInfo*, también se especifica una masa de 1 haciendo del cuerpo rígido, un cuerpo rígido dinámico.

```
ModelInstance sphereInstance;
private btDefaultMotionState sphereMotionState;

public void create() {
    //Código
    sphereMotionState = new btDefaultMotionState(sphereInstance
        ↪ .transform);
    sphereMotionState.setWorldTransform(sphereInstance.
        ↪ transform);
    final btCollisionShape sphereShape = new btSphereShape(1f);
    bodyInfo = new btRigidBody.btRigidBodyConstructionInfo(1,
        ↪ sphereMotionState,sphereShape,new Vector3(1,1,1));
    body = new btRigidBody(bodyInfo);

    world.addRigidBody(body);
}
```

Código 3.6: Creación de un cuerpo rígido con la figura de colisión de una esfera.

En el método *render* se invocará al método *stepSimulation;* invocar este método es de suma importancia ya que da paso a la simulación. Es necesario que el desarrollador provea dos o tres parámetros para hacer uso de este método, el primer parámetro será el tiempo delta obtenido por la librería *graphics* de *LibGDX*. Para el segundo parámetro, se ingresa el número máximo de pasos de *Bullet*, este parámetro evitará que el rendimiento de la simulación decrezca limitando el número de pasos permitidos. El tercer parámetro será el tiempo de paso arreglado; *Bullet* hará las calculaciones logradas en el tiempo especificado en el tercer parámetro hasta alcanzar el tiempo delta especificado en el primer parámetro, se limitará a hacer los cálculos especificados por el segundo parámetro (Nair y Oehlke, 2015). En el Código 3.7 se muestra a la variable *world* invocar al método *stepSimulation* ingresando únicamente los primeros dos parámetros; si no se especifica el tercer parámetro, el tiempo de paso arreglado por default es 60 Hz.

```
public void render() {
    //Código
    world.stepSimulation(Gdx.graphics.getDeltaTime(), 5);
    //Código
}
```

Código 3.7: Implementación de *Bullet* en método *render*.

La ejecución del programa finalizado se aprecia en la Figura 3.4 en una secuencia de imágenes en donde la fuerza de gravedad del mundo es aplicada a una esfera roja obligándola

a caer al suelo, seguido de esto la esfera se mueve aún en el suelo llegando al límite de este y finalmente cayendo desapareciendo de la cámara.

En el presente programa se logró la implementación de un mundo discreto, el cual posee manejo de colisiones entre otras cosas. Si se hubiera continuado el desarrollo de un motor físico propio, llegar hasta este punto hubiera tomado varias líneas de código en cambio se sintetizo un resultado palpable en un programa corto y entendible.

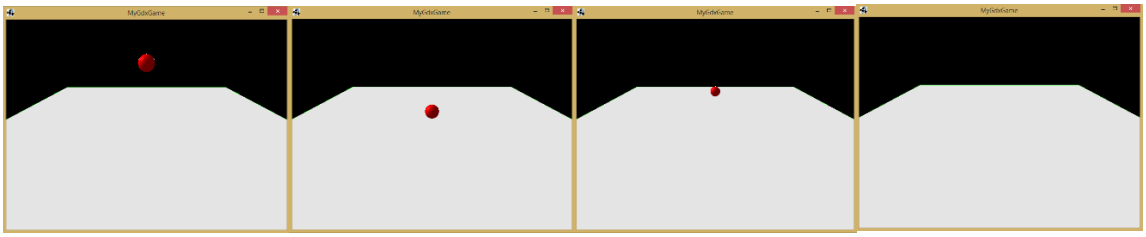


Figura 3.4: Esfera cayendo por la fuerza de la gravedad implementando *Bullet*.

4. METODOLOGÍA

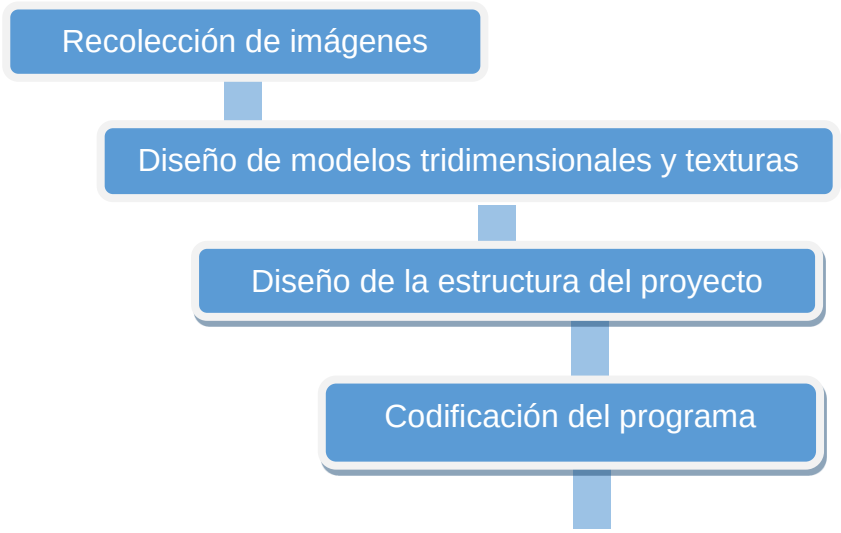
En la presente investigación se hace uso del concepto de triangulación de métodos, el cual plantea la utilización de más de un método.

Se utilizará un enfoque cuantitativo para plantear un problema de estudio concreto y antecedentes, por lo que llevaría a un análisis de la tecnología *LibGDX* y proyectos similares a la propuesta en la presente investigación. También se desarrolla una aplicación multiplataforma que compruebe la hipótesis expuesta en la introducción. Por último se llevaría a cabo una recolección de datos generados por el rendimiento de la aplicación en distintas plataformas.

Por otra parte, se utilizará un enfoque cualitativo para plantear la realización de una experiencia de un recorrido en un mundo tridimensional con física aplicada; la creación de modelos tridimensionales basada en una colección de imágenes de la facultad ayudará a contar con un resultado que las personas pueden identificar como instalaciones de una universidad.

Para la realización del presente proyecto, se optó por usar un par de características de una metodología de desarrollo de aplicaciones. Al observar la ruta metodológica, se puede apreciar que su flujo lineal de los procesos se asemeja a una metodología de cascada pero con distintos procesos.

A continuación se describirá a detalle, los métodos y pasos que llevaron a la realización de la aplicación orientada a dispositivos móviles y computadoras personales.



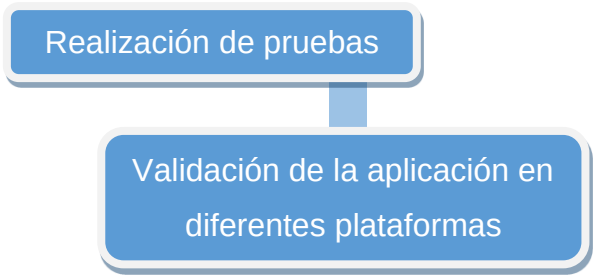


Figura 4.1: Ruta metodológica.

Recolección de imágenes: Se elaborará una colección de imágenes (como lo pueden ser fotografías y planos), que sirvan de referencia para la elaboración de modelos. Una vez contando con las referencias, se adecuará un proyecto en el programa *Modo* con las imágenes necesarias.

Diseño de modelos tridimensionales y texturas. Modelar el complejo arquitectónico pertinente basándose en las buenas técnicas de modelación, evitando deformaciones o apariencias poco fieles. Una vez concluido los modelos, se empezará a realizar mapas *UV's* en dichos modelos. Los mapas *UV's* se conocen así por sus coordenadas expresadas en U, V y W, siendo U horizontal, V vertical y W lo profundo. Los mapas *UV's* ayudan a definir como se alinea una textura con un objeto tridimensional (Murdock, 2014), esto ayudará para la implementación de texturas; específicamente se generan los mapas *UV's* para no evidenciar alguna deformación que puedan tener las texturas al momento de unir la textura con el modelo.

Contando con los modelos y sus respectivos mapas *UV´s*, es momento del desarrollo de texturas; estas se basarán en las referencias recabadas. La realización de texturas se llevará a cabo en un programa de diseño y se tratará de hacer esta parte, la parte más ágil del proyecto. En la realización de los modelos y texturas, se tuvo que considerar el nivel de detalle que contarán ciertas áreas. Parte del proceso es considerar la cantidad de vértices en los modelos, este proceso es fundamental porque los modelos que no sean soportados por los dispositivos más comunes, sólo ocasionarán problemas (McDermott, 2011). A estas alturas el proyecto contará con una estética agradable que se podrá mejorar y personalizar aún más con objetos que ofrece el *framework LibGDX*, esto se refiere a los objetos que conforman un ambiente.

Diseño de la estructura del proyecto. Se planteará la estructura de la aplicación considerando sus objetivos y aspectos más técnicos, como lo puede ser el funcionamiento

general de la tecnología *LibGDX* ya sea manejo de recursos, lógica de la aplicación, ciclo de vida de la aplicación, etcétera.

Codificación del programa. Se iniciará el desarrollo de código para cargar modelos y texturas a un proyecto *LibGDX*, posteriormente manipular estos objetos y asignarles una entidad con propiedades que fueron definidas en el esquema del programa.

Realización de pruebas. Se propone realizar pruebas de validación de la aplicación en su totalidad, se espera que la aplicación funcione correctamente usando un conjunto determinado de casos de prueba que reflejan el uso esperado de esta, posteriormente se realizan las pruebas de defectos, en los que los casos de prueba se diseñan para exponer los defectos (Sommerville, 2011). Por último se dará solución a los errores detectados.

Validación de la aplicación en diferentes plataformas. El entendimiento técnico detrás de crear recursos (modelos en tercera dimensión) es más profundo que solo modelar geometría de baja resolución. Cada plataforma o dispositivo tiene sus propias limitaciones y lo que puede correr bien en una plataforma no significa que corra bien en otra (McDermott, 2011) por lo que es importante en esta última etapa medir la fiabilidad del producto.

Otro tipo de pruebas que son de un gran interés en la presente investigación, son las pruebas de rendimiento, las cuales tienen que diseñarse para asegurar que la aplicación pueda procesar su carga esperada (Sommerville, 2011).

4.1. SUJETO EXPERIMENTAL

Se utilizaron los complejos arquitectónicos pertenecientes a la facultad de informática de la Universidad Autónoma de Querétaro como sujetos experimentales. Los edificios e inmuebles fueron guía para la realización de objetos tridimensionales.

4.2. SUJETOS DE PRUEBA

El programa de escritorio se probó principalmente en una computadora *Dell Inspiron 5737*, que cuenta con procesador *Intel Core I5*, con velocidad de CPU de 2.3 GHz, un sistema operativo de 64-bit y una memoria RAM de 8 GB

Por otra parte, la aplicación móvil se probó principalmente en un dispositivo *Galaxy Grand Prime*, que cuenta con un CPU *Quad-core* con velocidad de 1.2 GHz, un GPU *Adreno 306*, una memoria RAM de 1 GB y una memoria interna de 8 GB.

Por último, la aplicación móvil también se probó en un dispositivo *HTC ONE*, que cuenta con un procesador *Qualcomm Snapdragon* de cuatro núcleos, una velocidad de CPU de 1.7 GHz, un GPU *Adreno 320*, una memoria RAM de 2 GB y una memoria interna de 32 GB.

4.3 MODELADO Y TEXTURAS

En el presente trabajo no se tratará la creación de modelos con una dinámica "paso a paso", comúnmente presentado así por los libros enfocados en programas de creación de modelos tridimensionales, en cambio se abordarán los conceptos más importantes y las herramientas más útiles esperando cubrir lo más posible este tema.

Se describirá el programa de modelado en tercera dimensión *MODO*, así como el proceso de creación de modelos tridimensionales en dicho programa. Después se crearán e integrarán las texturas en el programa *Blender*, dicho programa y proceso serán explicados. Se escogió este flujo de trabajo debido a la facilidad y rapidez en que se trabaja la geometría en *MODO* y a la variedad de opciones que cuenta *Blender* para la configuración de texturas y variedad de formatos a exportar.

También se integrará un modelo y sus texturas al *framework LibGDX*; por último se le asignará un cuerpo rígido al modelo integrado para agregarlo en el mundo físico de *Bullet*.

4.2.1 MODO

"Modo es un programa de modelado en tercera dimensión, animación y *rendering*. Originalmente creado por *Luxology*." "La principal fusión de varias tecnologías (Modelado, escultura, pintura, *rigging*, animación, iluminación, texturizado, simulación y *rendering*) permite al usuario el apalancamiento de todas las tecnologías al mismo tiempo." (What Is modo? s.f.)

En la imagen izquierda de la figura 4.2 se muestra un modelo donde resaltan en color amarillo una colección de puntos, estos puntos poseen valores X, Y y Z, designando su

posición en el espacio tridimensional. Cada posición es referida como vértice; no tienen volumen o tamaño, simplemente son posiciones. (What Is 3D? s.f.) En la imagen del centro de la figura 4.2, se muestra el mismo modelo donde resaltan en color amarillo las conexiones entre vértices, a estas conexiones se les conoce como bordes y al igual que los vértices, no poseen volumen pero si longitud. Por último, la imagen derecha de la figura 4.2 muestra el mismo modelo, en esta ocasión las caras del modelo resaltan en color amarillo, el grupo de vértices y bordes definen una superficie conocida como polígono.

Vértices, bordes y polígonos, explicados de manera gráfica en la Figura 4.2, son conocidos como modos de selección del componente y son de gran utilidad para la manipulación de modelos y mapas UV.

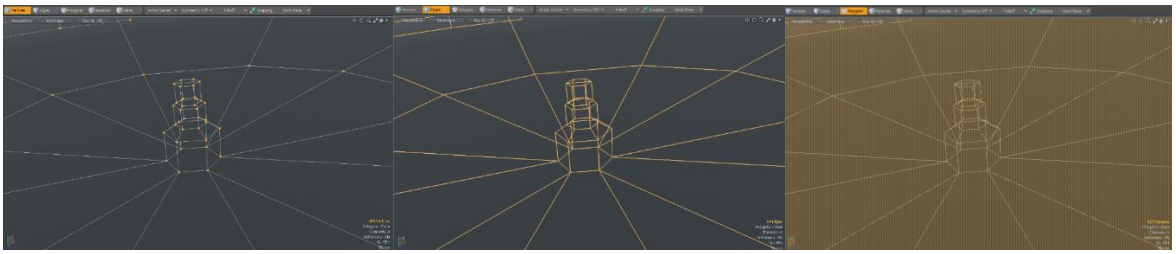


Figura 4.2: Vértices, bordes y polígonos en MODO

Realizar los modelos deseados, en este caso un complejo arquitectónico universitario requiere un arduo trabajo constante y una inversión mayor de tiempo. La elección de MODO permitirá la conclusión de los modelos gracias a su simple interfaz y variedad de herramientas.

Se tiene que tener en claro el objeto a modelar, la facultad de informática cuenta con una compleja y elegante arquitectura, por lo tanto es importante prestar atención a los detalles. Para este trabajo se utilizó una colección de fotografías de la facultad en diferentes puntos y perspectivas. En la figura 4.3, la imagen izquierda es una de las referencias con las que se trabajó el modelo mostrado en la imagen derecha.

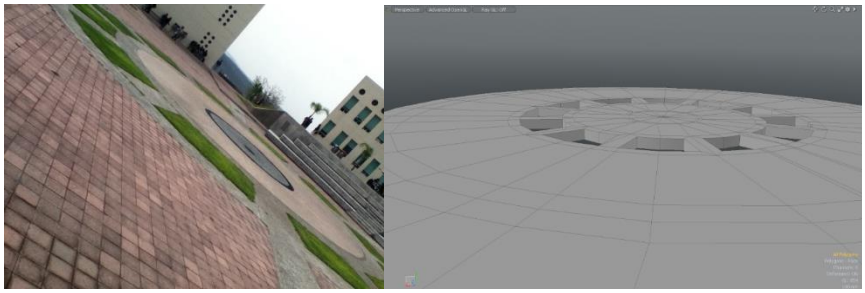


Figura 4.3: Fotografía de referencia (izquierda) y modelo en MODO (derecha)

Los modelos creados en este trabajo partieron de un objeto primitivo; "MODO ofrece una variedad de diferentes herramienta para la generación rápida de geometría. Objetos primitivos como esferas y cubos son una buena manera de construir formas básicas que pueden ser editadas, agregadas, modificadas y de otra manera, modeladas en la forma en que el usuario desee usando cualquiera de las herramientas de edición de deformación o *mesh.*" (Creating Geometry. s.f.) En la figura 4.4 se muestra un cubo como objeto primitivo creado.

Para crear un modelo complejo es necesario hacer uso de las herramientas que brinda MODO, este software cuenta con un catálogo amplio de herramientas y describir todas o la mayoría no es el alcance del presente trabajo, sin embargo se presentarán las de mayor uso e importancia.

En la segunda imagen de la Figura 4.4, se muestra el uso de la herramienta *"Polygon Extrusion"* o extrusión de polígono, el cual "reposiciona el polígono seleccionado en el espacio...; superficies de polígonos adicionales son creados entre la nueva locación del polígono y la original locación de las caras seleccionadas". (Polygon Extrude. s.f.) Dicha herramienta facilita la creación de nuevos polígonos y a definir una compleja figura manipulando la longitud del área extraída.

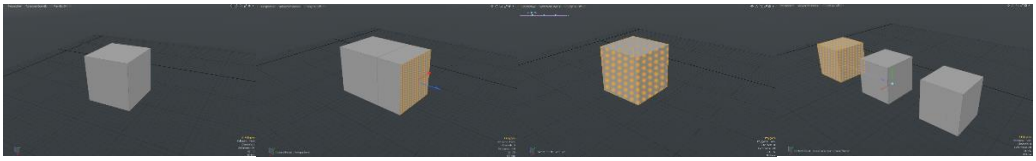


Figura 4.4: Figura primitiva y aplicación de distintas herramientas en *MODO*

Otra herramienta de constante uso es *Loop Slice* o bucle de rebanadas, "una poderosa herramienta que permite rebanar bucles de bordes dentro de la geometría" (Loop Slice, s.f.).

Se pueden especificar el total de rebanadas adicionadas a la geometría, así como la manipulación de estos bordes generados. Este tipo de herramientas permiten agregar detalle al modelo manipulando los bordes y vértices agregados. La muestra de su uso puede encontrarse en la tercera imagen de la figura 4.4.

MODO ofrece una vasta variedad de herramientas para clonar objetos, herramientas como clonación de curva, clonación de *Bezier*, arreglo, arreglo radial, entre otras; pero dentro del grupo *Duplicate*, la herramienta más simple y usada en el presente trabajo fue *Clone* o clonación; dicha herramienta "duplicará la geometría actualmente seleccionada en un desplazamiento lineal determinado por la entrada del usuario" (Clone, s.f.). El uso de la clonación es idóneo para sillas, ventanas, puertas, escritorios, columnas, y en general para todos aquellos modelos que fueran a repetirse en el programa. La muestra de su uso puede encontrarse en la última imagen de la figura 4.4.

Un concepto importante dentro del modelado en tercera dimensión es la topología, este concepto describe la efectividad del flujo del borde en la geometría, puede ser buena o mala dependiendo de ciertos factores. La atención a los factores que influyen en la calificación de una topología es importante cuando se está desarrollando modelos orgánicos (Hess, 2010); en el presente trabajo se modelarán únicamente edificio, es decir modelos inorgánicos, por lo tanto no se profundizará en este concepto y con respecto al flujo de los bordes en la geometría, solo será importante al considerar los diferentes materiales que existen en un modelo y como los bordes pueden potencialmente separar materiales.

Otra consideración importante es el ahorro de polígonos; es importante subrayar la importancia de mantener una reducción constante en la cuenta de vértices, es por eso que la interfaz de MODO cuenta con un contador de vértices, consulta que es importante para este tipo de trabajos donde el ahorro de vértices es primordial.

4.2.2. MAPEO UV

Las consideraciones y herramientas anteriormente explicadas son parte de lo que se necesitará para la creación de modelos, habiendo concluido esta parte será necesario asignar texturas para lograr el aspecto deseado; por razones de conversión de formatos de modelos tridimensionales, se asignarán las texturas y materiales a los modelos en el *software Blender*,

pero antes, es necesario preparar los modelos para la asignación de materiales y texturas, para esto se hará uso de los mapas UV.

"El mapeo UV es una forma muy precisa de mapear texturas 2D a una superficie 3D. Cada vértice de un *mesh* tiene sus propias coordenadas UV que puede desenvolverse y colocar planamente como una piel... Este mapeo es especialmente útil cuando se usan imágenes 2D como texturas." (Mapping, s.f.)

Para realizar la elaboración de los mapas *UV* se hará uso de la interfaz *UV Viewport* o ventana *UV*. Como lo muestra la Figura 4.5, dicha ventana contiene los valores *UV* en una cuadricula con área de 0 a 1; en la Figura 4.5 también se puede apreciar los siguientes puntos: el eje +U es el equivalente al eje de las abscisas y el eje +V es el equivalente al eje de las ordenadas.

En la Figura 4.5 se tiene el modelo mostrado en la Figura 4.3 representado por valores UV en la ventana UV, dichos valores son de tipo flotante y son posicionados proporcionalmente a través de la textura (Working with UV Maps, s.f.).

Para la generación de los valores UV presentado en la Figura 4.5, se tuvo que seleccionar los polígonos o bordes que se desean mapear y enseguida de esto, invocar alguna de las herramientas que se encuentran en el panel UV. Para el presente trabajo se hizo uso de la herramienta *UV Unwrap* o desenvolvedor UV, la cual "provee una solución elegante para crear mapas UV desde cualquier clase de *mesh*, desde una superficie dura a una orgánica." (UV Unwrap, s.f.)

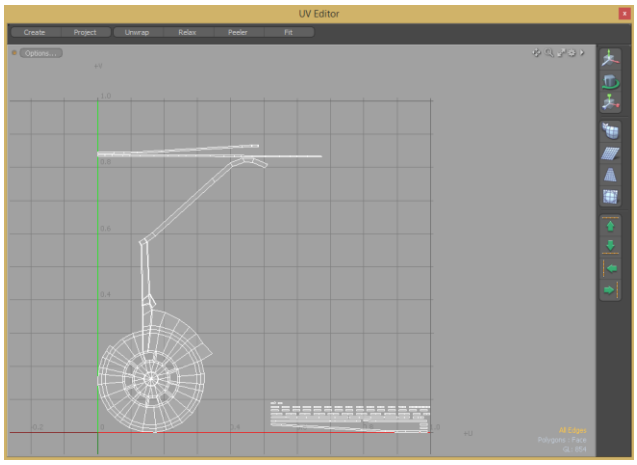


Figura 4.5: Ventana UV.

Para entender la herramienta *UV Unwrap* se provee el siguiente ejemplo; se invita al lector a imaginar un animal muerto al cual se tendrá que remover su piel, al acostar la piel removida debe de ser lo más plana posible sin estirarse mucho, por eso es necesario hacer los cortes necesarios para que no ocurra un acomodo extraño de piel. (UV Unwrap, s.f.).

Pensando aún en el ejemplo anterior, en la Figura 4.6 se muestra los bordes seleccionados que representarán los cortes a realizar. Identificar los bordes que deben seleccionarse para obtener un mapeo es resultado de la práctica constante. Es un tanto obvio que en las esquinas del modelo no se notará el corte, sin embargo en ocasiones (sobre todo en modelos orgánicos) es inevitable tener un corte notable y queda a decisión del desarrollador donde colocar dicho corte.

Uno de los objetivos principales de generar un mapeo UV es hacerlo con la menor cantidad de cortes posibles, pero se debe de tener cuidado al momento de intentar reducir lo más posible los cortes ya que pueden aumentarse las distorsiones. (Working with UV Maps, s.f.) La experiencia y la experimentación ayudan a encontrar un equilibrio entre el número de cortes y el número de distorsiones.

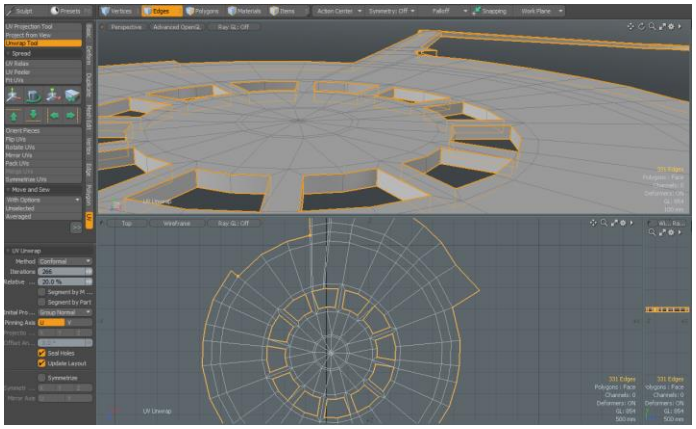


Figura 4.6: Selección de bordes para generación de mapa *UV*.

Siempre se tiene que evitar la sobre-posición de valores UV's, la herramienta Pack UV's o empaquetamiento de UV's ayuda a organizar las islas UV para aprovechar lo más posible el espacio UV y al final evita la sobre-posición entre valores UV. Aunque en ciertos

casos, el sobre-posicionamiento será una técnica que ayude a ahorrar uso de memoria en texturas y aumentar la resolución de la textura, conservando la imagen su mismo tamaño.

Modo soporta un gran rango de formatos entre ellos se encuentra LWO, OBJ, ABC, 3DM, DAE, FBX, entre otros. El último formato mencionado, FBX o *Autodesk fbx*, será el formato al que se exportarán de *Modo* e importarán en *Blender*; dicha dinámica se usará en todos los modelos del presente trabajo.

4.2.3. BLENDER

"*Blender* es una suite de creación de contenido tridimensional complemente equipado, ofreciendo una amplia gama de herramientas esenciales incluyendo modelado, *rendering*, animación, edición de video, VFX, composición, tutorización, *Rigging*, muchos tipos de simulaciones y creación de juegos." (Blender Foundation, s.f.-a) En el presente trabajo, *Blender* será utilizado únicamente para la generación de texturas y materiales.

"Cada superficie en *Blender* requiere de un material. Los materiales define la forma en que la luz interactúa con la superficie." (Hess, 2010) La substancia que muestra un objeto es representada por cualidades de la superficie como lo puede ser el color, el brillo, el reflejo, etcétera.

Para crear un material se seleccionará al objeto que se ha importado. En la pestaña de materiales dentro del editor de propiedades se encuentra la opción de crear un nuevo material, al crear un nuevo material se asignará al objeto seleccionado y el color del objeto cambiará de acuerdo a su material.

El *pre-visualizador* de materiales es un *"render en vivo"* que cambia al momento de hacer ajustes en la configuración del material (Hess, 2010), será de gran ayuda al integrar las texturas buscadas.

El siguiente paso será agregar una textura al material creado. "Las texturas son como capas adicionales encima del material" (Blender Foundation, s.f.-b). Existen dos tipos de texturas: texturas de imagen, las cuales usan un archivo de imagen, y texturas procedurales, que son generados al vuelo por modelos matemáticos (Hess, 2010); en el presente trabajo se trabajará únicamente con las texturas de imagen, ya que las texturas procedurales pueden

llegar a ser limitadas y los resultados deseados se encontraron en la aplicación de texturas de imagen.

Las texturas de imagen cuestan espacio de memoria, considerando que el programa se implementará en dispositivos móviles, se mantendrá el tamaño de las imágenes lo más pequeño posible. "Para el uso más eficiente de memoria, las imágenes de la textura deben de ser cuadradas, con dimensiones en potencia de 2, como lo son 32x32, 64x64, 128x128, 256x256, 1024x1024, 2048x2048, 4096x4096" (Blender Foundation, s.f.-b)

"Las texturas necesitan de un mapeo de coordenadas para determinar cómo se aplicarán al objeto." (Mapping, s.f.) En la Figura 4.7, la imagen izquierda muestra el panel de mapeo dentro de la sección de texturas. En esta parte se especificará un mapeo por medio de las coordenadas UV que se generó y se mostró con anterioridad.

En la figura 2.3, la imagen izquierda muestra una textura aplicada y un mapa UV en el editor UV; se aumentó el tamaño del mapa para formar un azulejo. "Cualquier textura aplicada usando un mapa UV siempre llenará el espacio entero 0-1; cualquier parte afuera de esta área simplemente se repetirá la textura." (Working with UV Maps, s.f.) La imagen derecha muestra el mapa en su totalidad.

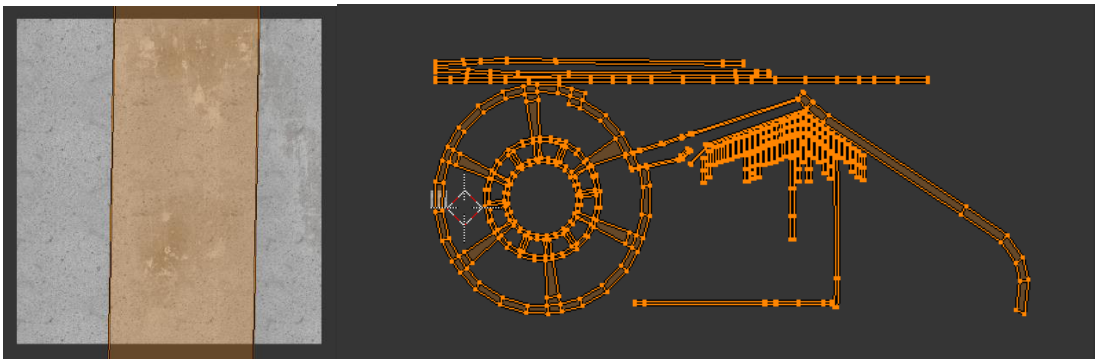


Figura 4.7: Editor *UV* en *Blender*.

Se procede a asignar los materiales y las texturas definidas, para realizar esta acción se debe seleccionar las caras del modelo. La vista 3D permite la selección de bordes, vértices y caras, las mismas posibilidades que *Modo* ofrece; en la imagen izquierda de la Figura 4.8 se muestra la selección de una colección de caras a las cuales se les asignará un material.

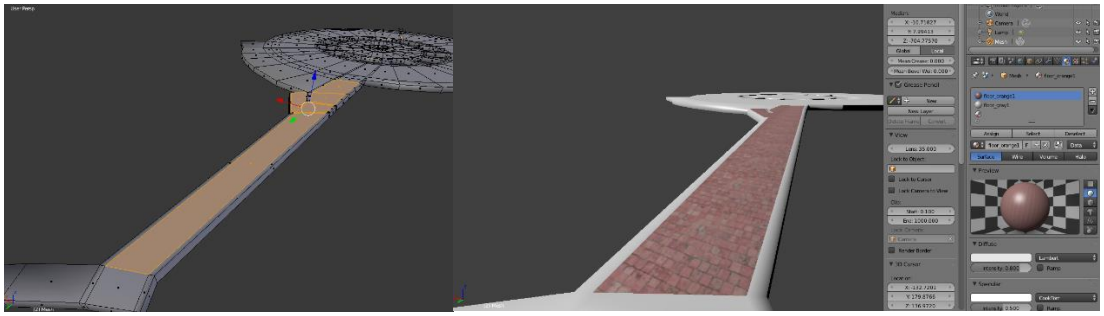


Figura 4.8. Asignación de material a polígonos seleccionados.

"La gran ventaja de las imágenes de textura es su atajo a la verosimilitud. Su desventaja es que tienen un principio y un final.... presenta un problema único cuando se trata de repetir a través de una superficie larga como muros o pisos." (Hess, 2010) Se tratará de ocultar los patrones más visibles, pero siendo considerado este programa para dispositivos móviles, se genera una limitante para el número de texturas con el que se piensa contar. Lo preferente será reducir la memoria asignada a las texturas empleadas en el programa, hay opciones para la reducción de carga, como lo son: usando compresión de texturas, materiales compartidos y textura de atlas. (McDermott, 2011)

En la Figura 4.9, se muestra el modelo que se pretende exportar, contando con diferentes texturas asignadas. Por otra parte, el *renderizado* mostrado en la Figura 4.9 será el resultado que se desea obtener en el proyecto de *Android Studio*.

"*LibGDX* soporta tres formatos de modelos llamados *Wavefront OBJ*, *G3DJ* y *G3DB*." (Nair y Oehlke, 2015) Para el presente trabajo se hará uso del formato *G3DJ*, el cual es de formato *JSON* y es editable con la ayuda de un simple editor de texto.

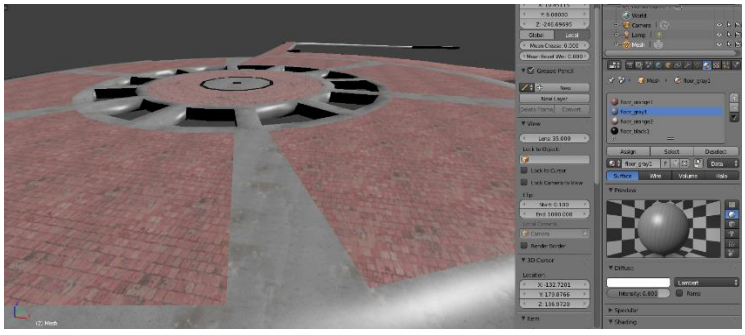


Figura 4.9: Modelo en *Blender* preparado para exportar.

Para exportar un archivo *FBX* a *3GDJ* es necesario contar con el aditamento *Blender G3D Exporter*, cuyo único objetivo de este aditamento es exportar a dicho formato.

4.2.4. IMPLEMENTACIÓN EN LIBGDX

Para implementar el modelo tridimensional al proyecto *LibGDX*, es necesario ubicarlo en la carpeta de recursos ubicada en las carpetas del proyecto *Android* y establecer que de dicha carpeta se tomarán los recursos, sin importar que se esté ejecutando la versión de escritorio. Ubicado el modelo en la carpeta de recursos, se procede a codificar su llamado con ayuda de la clase *AssetManager*. En el Código 4.1 se muestra la implementación de *AssetManager*, esta clase ayuda a la carga y manejo de recursos; el uso de esta clase es la manera recomendada de cargar recursos, en este caso, los modelos tridimensionales (Managing your assets, s.f.).

```
public Model model;
public ModelInstance instance;
public AssetManager assets;

public void create () {
      //Código
    assets = new AssetManager();
    assets.load('uaq_allmodels4_ab.g3dj', Model.class);
    assets.finishLoading();
    model = assets.get('uaq_allmodels4_ab.g3dj', Model.class
        ↪ );
    instance = new ModelInstance(model);
    instance.transform.rotate(1, 0, 0, -90);
      //Código
}
```

Código 4.1: Integración de un modelo *G3DJ* en *LibGDX*.

Para cargar un modelo, *AssetManager* necesita conocer el tipo de recurso a cargar. Al invocar el método *load*, el primer parámetro especifica la dirección y nombre del modelo dentro de la carpeta recursos; el segundo parámetro es el tipo de recurso, en este caso se especifica el tipo *Model*. Se utiliza la sentencia *finishLoading*, la cual asegura que todos los recursos se han cargado (Managing your assets, s.f.). Al cargar exitosamente el modelo, se llama a este con el método *get* utilizando los mismos parámetros que en *load*, dicho modelo lo recibe la instancia *model* de tipo *Model*. En el Código 4.1, se muestra a *model* como parámetro en el constructor de *ModelInstance* para así poder manipular con facilidad la

instancia de modelo; como regla para el presente proyecto, se rotará menos noventa grados las instancias con modelos *G3DJ* cargados, esto se realiza debido a un cambio de rotación ocurrido en la transición de *Blender* a *LibGDX*.

El modelo se integró a *LibGDX* con éxito; sin embargo únicamente se dibujaría, más no se tendría una interacción con los cuerpo rígidos existentes en el programa. En el Código 4.2, se muestra el uso de la librería *Bullet* para asignarle al modelo importado propiedades que harían de este, un cuerpo rígido estático. Como se abordó en el capítulo anterior, es necesario contar con una figura de colisión, para esto se invocará el método *obtainStaticNodeShape,* alimentado por un arreglo de nodos obtenido del modelo importado.

```
public void create () {
        //Código
      btCollisionShape arenaShape = Bullet.
          ↪ obtainStaticNodeShape(model.nodes);
      btRigidBody.btRigidBodyConstructionInfo bodyInfo = new
          ↪ btRigidBody.btRigidBodyConstructionInfo(0,new
          ↪ btDefaultMotionState(instance.transform),
          ↪ arenaShape,Vector3.Zero);
      btRigidBody body = new btRigidBody(bodyInfo);
      world.addCollisionObject(body);
        //Código
   }
```

Código 4.2: Cuerpo rígido de un modelo.

La figura de colisión se almacena en la variable *arenaShape*, enseguida se crea la construcción de información del cuerpo rígido y se agrega al cuerpo, un objeto *MotionState* con la transformación de la instancia del modelo para efectuar la rotación a dicho modelo. El cuerpo rígido se agrega en el mundo discreto y el resultado se puede apreciar en la imagen izquierda de la Figura 4.10; en la imagen derecha se muestra el mismo programa en ejecución con la omisión del dibujado del modelo, quedando en pantalla únicamente los bordes de la figura de colisión que dibuja la clase *DebugDrawer*.

Es importante mencionar que la geometría creada en *MODO* es distinta a la presentada en la imagen derecha de la figura 4.10. Al momento de exportar de *Blender* a *LibGDX,* se generan los polígonos triangulares vistos en la imagen, dicha adición da como resultado modelos con más polígonos y por lo tanto, más gasto de memoria.

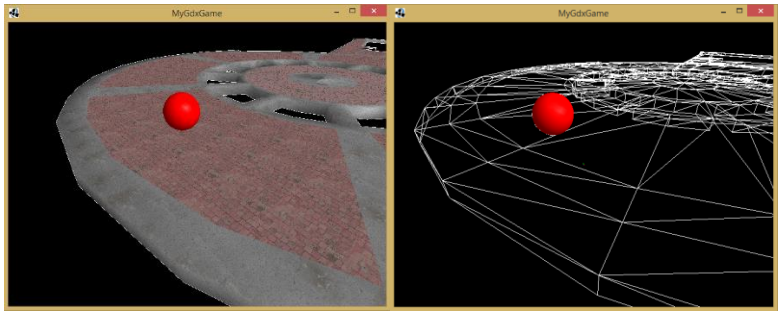


Figura 4.10: Modelo importado a *LibGDX* (izquierda) y figura de colisión de modelo importado a *LibGDX* (derecha).

4.3. PERSONAJE CINEMÁTICO

Existen diferentes perspectivas de cámara en las simulaciones en tercera dimensión como son los videojuegos. Para este trabajo se optó por una vista en primera persona debido a que la identidad y estética del personaje no es relevante, y la percepción en primera persona concentra al usuario en el complejo arquitectónico y sus detalles.

En el método *create* se empieza con la implementación del modelo (como se muestra en el código 4.3), el cual fue creado e importado a *LibGDX* en el anterior capitulo. En seguida, se configura *Bullet* con la inclusión de una variable nombrada *ghostPairCallback* de tipo *btGhostPairCallback*.

La variable *broadphase* de tipo *btDbvtBroadphase* invoca al método *getOverlappingPairCache*, el cual devuelve un objeto de tipo *btOverlappingPairCache*, esta clase provee una interfaz para el manejo de superposición de pares. La clase *btOverlappingPairCache* cuenta con el método *setInternalGhostPairCallback*, al cual se ingresará la variable *ghostPairCallback* y de esta manera se podrá hacer uso de objetos fantasmas o *btPairCatchingGhostObject's* en el mundo físico. Si no se llevara a cabo esto, el personaje principal no colisionaría con los objetos importados y únicamente caería al vacío.

```
private btDefaultCollisionConfiguration collisionConfiguration
    ↪ ;
private btCollisionDispatcher dispatcher;
private btDbvtBroadphase broadphase;
private btSequentialImpulseConstraintSolver solver;
private btDiscreteDynamicsWorld world;

btGhostPairCallback ghostPairCallback;
btPairCachingGhostObject ghostObject;
btConvexShape ghostShape;

btKinematicCharacterController characterController;

public void create () {
    //Creación del ambiente
    //implementación del modelo
    //Configuración de Bullet

    ghostPairCallback = new btGhostPairCallback();
    broadphase.getOverlappingPairCache().
        ↪ setInternalGhostPairCallback(ghostPairCallback);
}
```

Código 4.3: Inclusión de *btGhostPairCallback.*

Como se muestra en el código 4.4, se inicializan los valores de velocidad y tamaño correspondientes al personaje principal al cual se le asignará la forma de una capsula debido a que es la más recomendada y usada como figura alrededor del personaje.

Es necesario inicializar la variable *ghostObject* de tipo *btPairCachingGhostObject*, al cual se le asigna la matriz de transformación de la variable *player* de tipo *ModelInstance*.

Se inicializa la variable *ghostShape* de tipo *btConvexShape* (una clase heredada de *btCapsuleShape*), ingresando la misma altura y radio a la asignada al objeto *player*. Por último, se invoca el método *setColisionShape* de la clase *btGhostPairCallback* con el parámetro *ghostShape*, de esta manera, se tienen los dos parámetros necesarios para inicializar la variable *characterController*.

btKinematicCharacterController es una clase experimental que cuenta con un controlador de personaje no físico; utiliza la figura fantasma o *btGhostShape* que se ingresó como parámetro para ejecutar solicitudes de colisión para un personaje que pueda subir escaleras, deslizarse, etcétera (Coumans, 2015). Para inicializar *btKinematicCharacterController*, se ingresa como parámetros las variables anteriormente

configuradas: *ghostObject* y *ghostShape*, así como un valor flotante que representa la altura de escalón.

```
public ModelInstance player;
float velocity;
float height;

btGhostPairCallback ghostPairCallback;
btPairCachingGhostObject ghostObject;
btConvexShape ghostShape;

btKinematicCharacterController characterController;

public void create () {
    //Código

    velocity = 30f;
    height = 22f;
    player = addCapsule(80f, 200f, 50f);

    ghostObject = new btPairCachingGhostObject();
    ghostObject.setWorldTransform(player.transform);
    ghostShape = new btCapsuleShape(height/3, height);
    ghostObject.setCollisionShape(ghostShape);
    characterController = new btKinematicCharacterController(
        ↪ ghostObject, ghostShape, 5f);

    world.addCollisionObject(ghostObject,
          (short) btBroadphaseProxy.CollisionFilterGroups.
              ↪ CharacterFilter,
          (short) btBroadphaseProxy.CollisionFilterGroups.
              ↪ StaticFilter);
    world.addAction(characterController);

//Se agrega el modelo cargado con anterioridad al mundo físico
}
```

Código 4.4: Configuración del personaje.

Aún en el código 4.4, el método *addCollisionObject* perteneciente a *btCollisionWorld* (del cual hereda *btDiscreteDynamicsWorld*), agregará al objeto fantasma. Aparte de agregar *ghostObject* en sus parámetros, se especifica que colisionará con objetos estáticos. Por último, se invoca al método *addAction* perteneciente a *btDiscreteDynamicsWorld* y se inserta un único parámetro de tipo *btActionInterface* (del cual hereda *btKinematicCharacterController*), la cual es una "simple interfaz que permite acciones como

carros y personajes sean actualizados adentro de *btDynamicsWorld*" (Interface btActionInterface, s.f.).

En el código 4.5 se presenta las instrucciones necesarias para rotar la cámara del personaje; hacia arriba en caso de que se presione la tecla N y hacia abajo en caso de que se presione la tecla M. Para efectuar la rotación se hará uso del método *rotate,* perteneciente a la clase Camera, dentro de este método se hace la rotación del vector ascendente y de dirección (vectores pertenecientes a la clase Camera), estas rotaciones son efectuadas por el método *rotate* perteneciente a la clase Vector3.

Dentro de las operaciones internas del método *rotate* se crea un objeto *Quaternion* y se le asigna el vector de eje y los grados a rotar.

```
public void render () {
      //Código

      if(Gdx.input.isKeyPressed(Input.Keys.N)){
         cam.rotate(cam.direction.cpy().crs(Vector3.Y), -5f);
      }
      else if(Gdx.input.isKeyPressed(Input.Keys.M)){
         cam.rotate(cam.direction.cpy().crs(Vector3.Y), 5f);
      }
        //Código
   }
```

Código 4.5: Rutinas de rotación vertical en la cámara del personaje.

El *Quaternion* creado se establece como una matriz 4x4 con el método *set* perteneciente a la clase *Matrix4,* y la matriz resultado es multiplicada por el vector (ascendente o de dirección).

Para la simulación 3D, un *quaternion* es una manera de representar una rotación con cuatro valores de tipo flotante, dentro del programa se podría almacenar la orientación de un objeto en un *quaternion,* y si es necesario después de n rotaciones del objeto, se puede restaurar la orientación usando el *quaternion* con los valores almacenados. Los *quaternions* también cuentan con la ventaja de que pueden ser interpolados entre sí, dicha característica no se utilizará en este trabajo (Madhav, 2014).

En el desarrollo de simulaciones en tercera dimensión, los *quaternions* son una materia importante, basta y compleja, la cual no se profundizará ya que *LibGDX* cuenta con una serie de métodos que facilitan al usuario el uso de los *quaternions* y las operaciones que se pueden efectuar con ellos. Inclusive no se hará uso directo de los *quaternions*, son únicamente objetos que se utilizan detrás de los métodos de rotación que se invoquen.

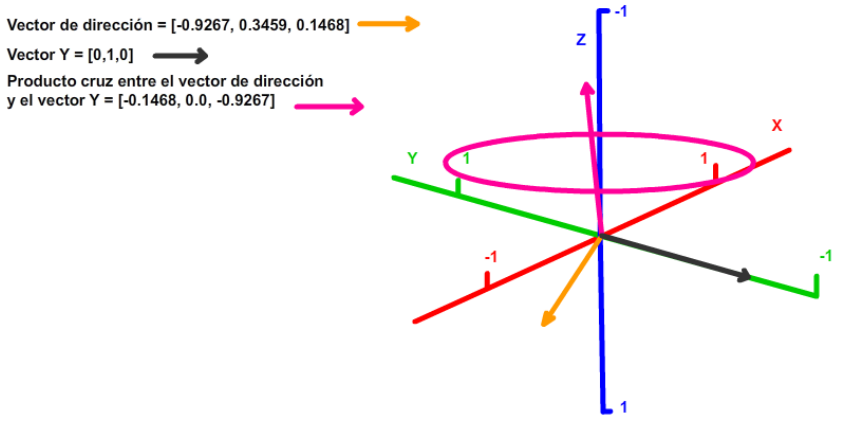


Figura 4.11: Representación del vector de dirección, vector Y, y el producto cruz entre ambos vectores.

El primer parámetro ingresado en el método *rotate*, es el vector resultado del vector copia de la dirección por el producto cruz del vector Y. Se utiliza la copia del vector de dirección para no afectar al vector de dirección original, el vector Y es sencillamente [0, 1, 0].

El vector de dirección y el vector Y, ambos comparten un mismo plano; el producto cruz encuentra un vector que es perpendicular a este plano conocido como una normal al plano (Madhav, 2014). En la Figura 4.11 se puede apreciar la representación del vector de dirección, el vector Y y el producto cruz entre ambos vectores. El vector de dirección representado por la línea naranja junto con el vector ascendente, serán rotados por el eje del vector resultante del productor cruz representado por la línea morada.

En cada actualización donde se ingrese la tecla N o M se disparará la operación donde el vector de dirección constantemente está cambiando, pero el vector Y continúa siendo [0,

1, 0], de esta manera se logrará la rotación vertical deseada representada por resultados reales y mostrada en la Figura 4.12.

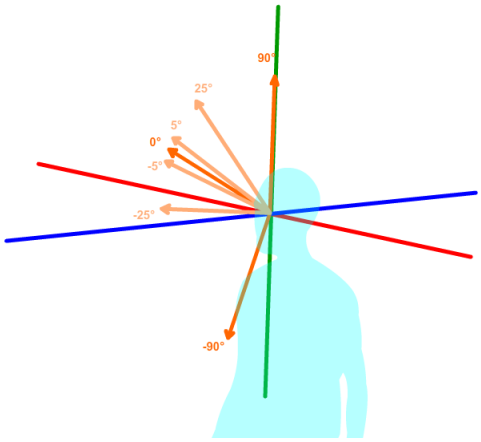


Figura 4.12: Rango de rotación vertical del personaje.

Contando con la rotación vertical de la cámara, continúa la rotación horizontal, la cual será más sencilla e inmediata ya que hará uso del método *rotateAround* perteneciente a la clase Camera. En el código 4.6 se tiene una rutina en la cual, al presionar la tecla *Left*, el personaje volteará hacia al lado izquierdo, y del lado opuesto si se presiona la tecla *Right*. Al presionar cualquiera de estas dos teclas, se rotará la matriz de transformación del objeto *player* y se asignará dicha matriz al objeto *ghostObject* con el método *setWorldTransform*. La rotación de la matriz de transformación del objeto *player* es opcional, porque al rotar una capsula, la transformación no es muy relevante y únicamente se podría rotar la cámara.

Para utilizar el método *rotateAround*, se ingresa como parámetro un objeto *Vector3*, que será el punto principal. En el código 4.6 se puede ver el ingreso de valores pertenecientes a la matriz de transformación de la variable *player*; los valores obtenidos son las coordenadas X, Y y Z del punto central del objeto, de esta manera rotará en el punto central del personaje principal. El segundo parámetro a ingresar es un objeto *Vector3* que corresponde al eje a rotar alrededor, en este caso será el eje Y representado en un *Vector3* como [0, 1, 0]; Por último se ingresa un valor flotante que será el ángulo a rotar.

```
public void render () {
        if (Gdx.input.isKeyPressed(Input.Keys.LEFT)) {
         player.transform.rotate(0, 1, 0, 5f);
         ghostObject.setWorldTransform(player.transform);
         cam.rotateAround(new Vector3(player.transform.val
             ↪ [12], player.transform.val[13], player.
             ↪ transform.val[14]), new Vector3(0, 1, 0), 5f);
         }
      else if (Gdx.input.isKeyPressed(Input.Keys.RIGHT)) {
         player.transform.rotate(0, 1, 0, -5f);
         ghostObject.setWorldTransform(player.transform);
         cam.rotateAround(new Vector3(player.transform.val
             ↪ [12], player.transform.val[13], player.
             ↪ transform.val[14]), new Vector3(0, 1, 0), -5f)
             ↪ ;
      }

        // Código
}
```

Código 4.6: Rutinas de rotación horizontal en la cámara del personaje.

El personaje del presente trabajo ya cuenta con las rotaciones necesarias para permitirle al usuario el cambio de perspectivas, únicamente falta agregar la opción de caminar hacia delante o hacia atrás dependiendo de la tecla que se accionó.

Se declaran y asignan los vectores *characterDirection* y *walkDirection*, como se puede observar en el código 5. *CharacterDirection* será el vector que contendrá la dirección actual del personaje, *walkDirection* será el vector por el cual determinará si se camina hacia adelante o hacia atrás.

A la variable *characterDirection* se le asignan los valores [-1, 0, 0], a parte se invoca el método *rot* ingresando como parámetro la matriz de transformación del personaje. El método *rot* multiplica el vector por las primeras tres columnas de la matriz, esencialmente solo se aplica para rotaciones y escalas (Zechner, 2017). El resultado se normaliza y ahora se cuenta con la dirección del personaje.

Al accionar la tecla *Up*, se sumará al vector *walkDirection* el vector *characterDirection*; debido a la asignación de los valores [0, 0, 0] a la variable *walkDirection*, el resultado de la suma será igual al vector *characterDirection*. Sin embargo el resultado cambiaría si se acciona la tecla *Down*, donde se suman los componentes negativos de

characterDirection; dicha sentencia determina si el personaje se mueve hacia delante o hacia atrás.

```
public void render () {
        //Código
      Vector3 characterDirection = new Vector3(),
          ↪ walkDirection = new Vector3();
      characterDirection.set(-1,0,0).rot(player.transform).nor
          ↪ ();
      walkDirection.set(0, 0, 0);
      if (Gdx.input.isKeyPressed(Input.Keys.UP)) {
         walkDirection.add(characterDirection);
      }
      else if (Gdx.input.isKeyPressed(Input.Keys.DOWN)) {
         walkDirection.add(-characterDirection.x, -
             ↪ characterDirection.y, -characterDirection.z);
      }
      walkDirection.scl(velocity * Gdx.graphics.getDeltaTime()
          ↪ );
      characterController.setWalkDirection(walkDirection);
      world.stepSimulation(Gdx.graphics.getDeltaTime(), 5);
      cam.position.set(player.transform.val[12], player.
          ↪ transform.val[13] + (height/3), player.transform.
          ↪ val[14]);
      cam.update();
      ghostObject.getWorldTransform(player.transform);
        //Código
   }
```

Código 4.7: Rutina para permitir al personaje caminar hacia delante o hacia atrás.

El producto escalar en muchos casos permite el cálculo de la magnitud por un vector en la dirección de otro (Millington, 2007); como se puede observar en el código 4.7, el producto escalar es usado por el vector *walkDirection* ingresando como parámetro la velocidad multiplicada por el tiempo delta.

Finalizando las operaciones por los vectores, se utiliza el método *setWalkDirection* perteneciente a la clase *btKinematicCharacterController*; se ingresa como parámetro no una dirección, ni una velocidad, sino la cantidad para incrementar la posición en cada iteración de la simulación (Bullet Collision Detection & Physics Library: btKinematicCharacterController Class Reference, s.f.).

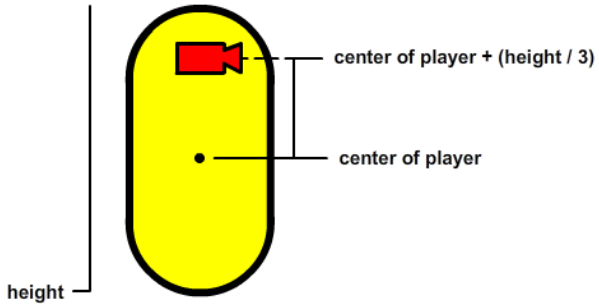


Figura 4.13: Figura del personaje.

Al termino del movimiento del personaje, se debe actualizar la posición de la cámara, de otra manera dejaría de ser una perspectiva en primera persona, para esto se toma de referencia la posición del personaje; al utilizar los valores de la matriz de transformación de la variable *player*, se obtiene el centro de la figura esférica, la cual es el personaje. En la imagen 4.13 se puede apreciar la representación de la figura del personaje principal, el alto de la figura está contenido en la variable *height.* Para elevar la cámara a un alto que simule la posición de la cara del personaje se suma el centro del personaje con la división de *height* entre tres, dicha operación se ingresa como parámetro en la asignación de la posición en el eje de las ordenadas de la cámara.

4.4. INTEGRACIÓN

En este trabajo se han abordado los conceptos principales relacionados a la creación de una aplicación en un mundo tridimensional utilizando las librerías de *LibGDX*; contando con la manera de llevar a cabo la aplicación, se procede a plantear las clases con sus respectivas relaciones en un diagrama de clases UML. "El Lenguaje unificado de modelado es una notación gráfica para dibujar diagramas de conceptos de software" (Martin, 2003), en este caso, para diseño de software.

Antes de iniciar la codificación, es útil el planteamiento de clases, objetos que se saben estarán eventualmente en el programa y se proyectan en el diagrama ayudando al programador en su desarrollo (Martin, 2003). Es importante el diseño del diagrama, una jerarquía bien diseñada de clases es la base para el rehúso, el uso de polimorfismo, encapsulación y herencia permiten crear un código limpio, sensible, fácil de leer y elástico (Schildt, 2007).

La arquitectura e implementación de código de esta investigación, están basados en el código creado por Suryakumar Balakrishnan Nair y Andreas Oehlke presentado en el libro *Learning LibGDX Game Development Second Edition*; aunque la aplicación que estos autores explican difiere bastante con la del presente trabajo.

4.4.1. CLASES AUXILIARES

Las clases auxiliares serán aquellas que se usarán como utilidades a lo largo del programa. Se cuenta con tres clases auxiliares: *Constants*, *Asset* y Loading, dichas clases se describen a continuación.

En el presente trabajo se hace bastante uso de la clase *Constants*, conformada por variables de tipo *String* que hacen referencia en su mayoría a archivos dentro de la carpeta de recursos; es importante que los nombres de las variables sean relevantes a la intención y sean fáciles de ubicar permitiendo al desarrollador agilizar su labor (Martin, 2009). Las variables son estáticas, es decir, pueden ser accedidas antes de que cualquier objeto de su clase sea creado (Schildt, 2007), facilitando así su constante llamado. También las variables creadas en la clase *Constants* son declaradas como finales, previniendo que el contenido sea modificado (Schildt, 2007).

```
public class Constants {
    public static final String MODEL_OUTSIDE1 = "exterior1.
        ↪ g3dj";
    public static final String MODEL_OUTSIDE1_TAG = "
        ↪ MODEL_OUTSIDE1";
    public static final String MODEL_OUTSIDE2 = "outside2.g3dj
        ↪ ";
    public static final String MODEL_OUTSIDE2_TAG = "
        ↪ MODEL_OUTSIDE2";
    public static final String MODEL_GRASS1 = "cesped.g3dj";
    public static final String MODEL_GRASS1_TAG = "
        ↪ MODEL_GRASS1";
    public static final String
        ↪ MODEL_COMPLEXS_BUILDING3_FLOOR1_PART1 = "
        ↪ edificio3_floor1.g3dj";
    public static final String
        ↪ MODEL_COMPLEXS_BUILDING3_FLOOR1_PART1_TAG = "
        ↪ MODEL_COMPLEXS_BUILDING3_FLOOR1_PART1";
    //Código...
}
```

Código 4.8: Clase *Constants*.

En el código 4.8 se muestra la clase *Constants* y alguno de sus componentes; debido a la enorme cantidad de recursos (y por lo tanto de variables) no se consideró presentar la clase *Constants* en su totalidad.

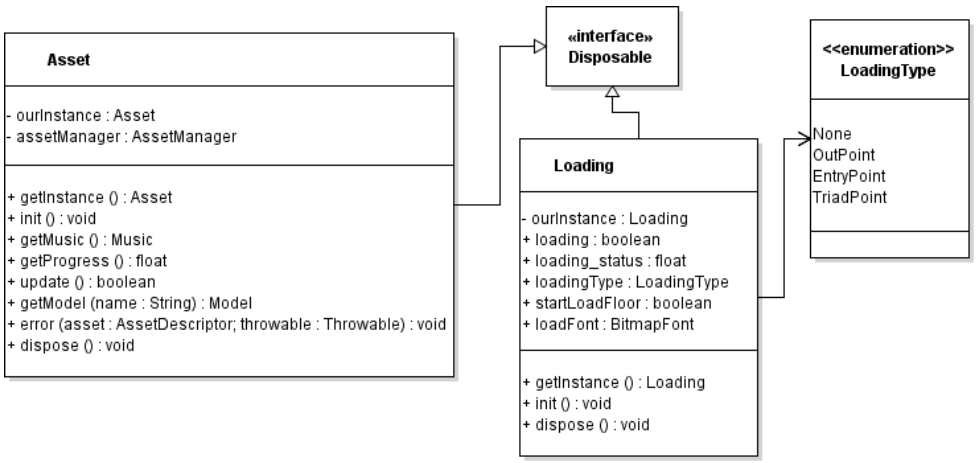


Figura 4.14: Clases auxiliares.

La clase *Asset* será de suma importancia, ya que será la encargada de organizar y estructurar los recursos de la aplicación; lo primero que se puede notar es el uso de un patrón de diseño llamado *singleton*. La intención de este patrón de diseño, es asegurar que una clase únicamente posea una instancia haciendo responsable a la misma clase del rastreo de esta única instancia, de esta manera se asegura que no exista otra instancia (Gamma et al., 2016). Definiendo al constructor como privado e inicializando la clase en una variable privada y estática de tipo *Asset*, se excluye la creación de otra instancia fuera de la clase *Asset*; estas medidas cobran sentido al reflexionar la existencia de múltiples instancias que controlen los recursos (Oehlke y Nair 2015).

Se utiliza la clase *AssetManager* para la carga y manejo de recursos. Por otra parte, se implementa la interfaz *AssetErrorListener* y se invoca *setErrorListener;* esto para utilizar el método *error* e imprimir posibles errores en la carga de recursos.

Por último, se cargan los recursos con el método *loadAsset* (definido en la misma clase) introduciendo las direcciones contenidas en *Constants,* junto con la clase del recurso (en su mayoría de tipo *Model).*

En la clase *Loading* se conserva: un valor booleano indicando si la aplicación está en estado de carga, un valor flotante indicando su número de carga y un objeto tipo *enum* con los tipos de carga que otras clases requerirán.

4.4.2. MUNDO FÍSICO

Los complejos arquitectónicos con los que contará la aplicación, deberán tener un cuerpo rígido en el mundo físico. Si no fuera así, el personaje principal sería incapaz de colisionar contra paredes, subir escalera o inclusive no podría pararse sobre el suelo. Por eso, es de gran utilidad la clase *BulletObject,* ya que los objetos cuya colisión contra el personaje sea relevante, será necesario que extiendan de esta clase.

Es importante mencionar que al crear un objeto que extiende de *BulletObject*, no significa una integración inmediata al mundo físico, ya que se deben de usar métodos que se abordarán en la clase *BulletWorld*.

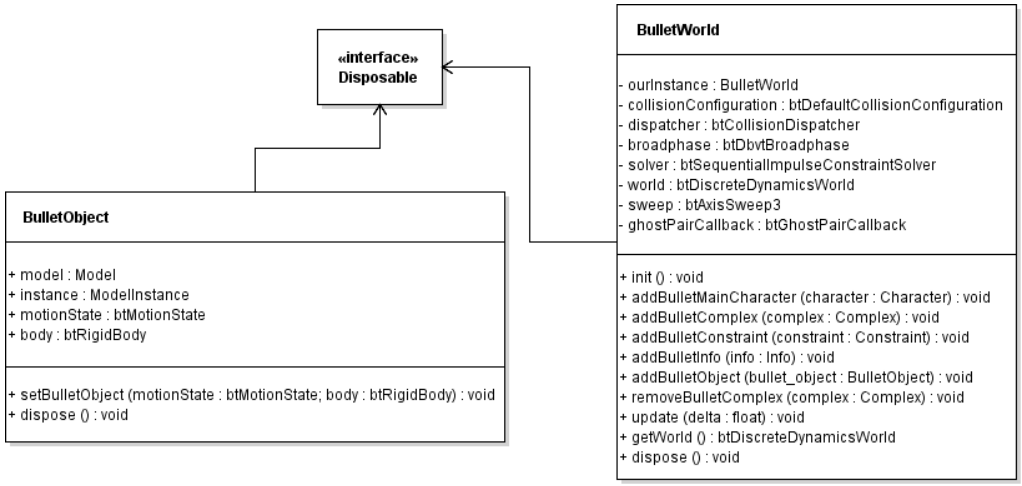


Figura 4.15: Clases para implementar un mundo físico.

Parecido a la clase *Asset*, se utiliza el patrón de diseño *singleton* para manejar el uso de una única variable, ya que solamente debe existir una clase que controle la física.

Se cuenta con el método *init* para iniciar la configuración de *Bullet*, expuesta en capítulos anteriores. También se cuenta con los métodos *addBulletMainCharacter*, *addBulletComplex* y *addBulletInfo*. El método *addBulletMainCharacter* es el encargado de agregar al personaje principal al mundo físico y el cual estará en control del usuario. El método *addBulletComplex* se encargará de agregar los complejos arquitectónicos y el método

addBulletInfo se encargará de agregar cubos de información; estos dos últimos métodos utilizan el método privado *addBulletObject*. En *addBulletObject* es importante notar que en el componente *userData* se almacena el objeto, esto se debe a consultas posteriores dentro de la aplicación. Por último, aún en el método *addBulletObject*, se invoca al método *setBulletObject* perteneciente a la clase *BulletObject* para almacenar componentes, los cuales tendrán que invocar al método *dispose* al término de la aplicación.

La clase *BulletWorld* también cuenta con el método *removeBulletComplex*, método encargado de remover el cuerpo rígido del mundo discreto *Bullet*, el cual será invocado regularmente por la clase *WorldController*.

4.4.3. COMPLEJO ARQUITECTÓNICO

Dentro del mundo tridimensional en la aplicación, existirán diferentes objetos que colisionarán contra el usuario ya sean edificios, exteriores, componentes de piso, etcétera. Es pertinente que estos objetos hereden de la clase *Complex*, ya que la numerosa cantidad de clases pueden volverse relativamente cortas al encapsular la configuración de la instancia del modelo en el constructor de una clase padre.

Antes que nada, es importante hacer mención de la interfaz *InterfaceModelObject*, ya que casi todos los objetos que manejen un modelo y una instancia de modelo, tendrán que implementar el método *models_allowed*, el cuál especifica los modelos permitidos en un objeto; cada clase definirá su colección de etiquetas y direcciones de modelos.

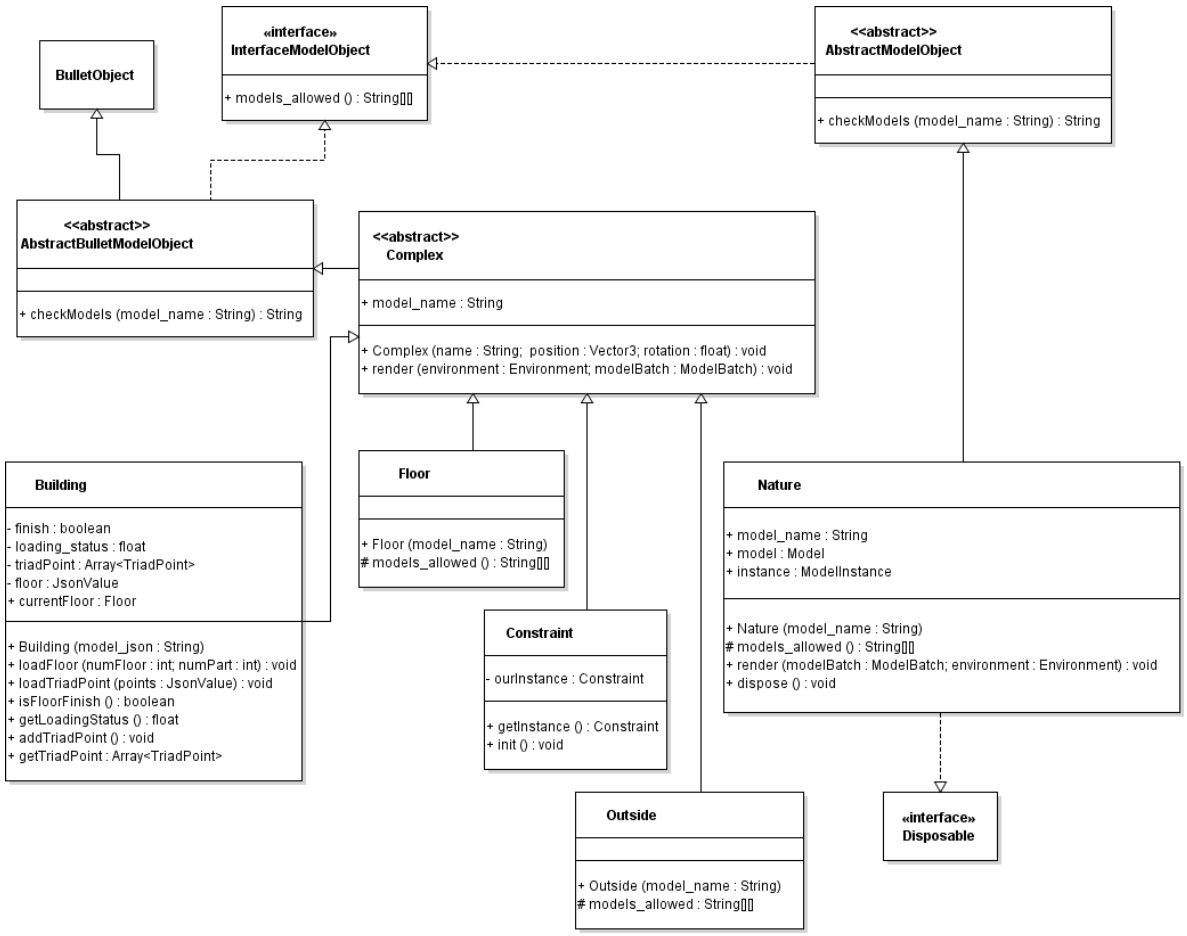


Figura 4.16: Clases pertinentes al complejo arquitectónico.

Existirán dos clases abstractas que implementarán la interfaz *InterfaceModelObject*, *AbstractModelObject* y *AbstractBulletModelObject*. Ambas clases son similares y cuentan con el método *checkModels*, el cual devuelve una cadena de texto que contiene información sobre el modelo permitido, o en caso de ser un modelo no permitido, devuelve un valor nulo. *AbstractBulletModelObjet* hereda de *BulletObject* y se usará en aquellos objetos que posean un cuerpo físico; esta herencia es la única diferencia entre *AbstractModelObject*.

Como se puede notar en la Figura 4.16, la clase *Complex* es abstracta, esto para que las subclases que hereden de *Complex* tengan que definir el retorno de *models_allowed* y dicho retorno no tenga que ser definido en la clase padre.

En el constructor de *Complex* se verifica si el modelo es permitido. Por último, se cuenta con el método *render* para dibujar el complejo.

En la clase *Building* presentada en el código 4.9, se puede observar los tres edificios definidos en el método *models_allowed* y una serie de variables que manejarán una dinámica de carga y descarga de pisos. En esta lectura, cuando se refiere al piso perteneciente a un edificio, se refiere al modelo el cual contiene los inmuebles que conforma un piso (mesas, sillas, botes de basura, bancas, etcétera).

La creación de un edificio será en base a un archivo JSON presentado en el código 4.10.

```
public class Building extends Complex {
    @Override
    public String[][] models_allowed() {
        return new String [][] {
                { Constants.MODEL_BUILDING1_1_TAG, Constants.
                    ↪ MODEL_BUILDING1_1 },
                { Constants.MODEL_BUILDING2_TAG, Constants.
                    ↪ MODEL_BUILDING2 },
                { Constants.MODEL_BUILDING3_TAG, Constants.
                    ↪ MODEL_BUILDING3 }
        };
    }
```

Código 4.9: Modelos permitidos en la clase *Building*.

JavaScript Object Notation o *JSON* es un simple formato de intercambio de información, si bien es una notación diseñada para la tecnología web, también es usado en diferentes contextos; el presente trabajo haciendo uso de la tecnología *LibgGDX* es un buen ejemplo de ello (Droettboom, 2016). Al inicializar un objeto *Building*, es necesario introducir como parámetro una ruta en la carpeta de recursos que apunte a un archivo *JSON*, el esquema *JSON* está compuesto por el nombre y por el objeto *floor*, el cual posee una colección de pisos que a su vez posee una colección de partes; esto quiere decir que un piso se puede dividir en n partes, y dentro de cada parte se especifica el modelo correspondiente al piso. Otro componente de suma importancia es el objeto *point*, el cual es conformado por una colección de puntos de carga, este concepto se abordará posteriormente.

```
{
        "model": "MODEL_BUILDING1_1",
        "floor": [
        { "floor": 3,
                "part": [
                { "part": 1,
                "modelFloor" : "
                    ↪ MODEL_COMPLEXS_BUILDING1_FLOOR3_PART1"
                }
                ]
        }
                /* ... */
        ],
        "point": [
        {"typePoint":"MIDDLE_POINT"
        ,"position":{"x": 166.0, "y": 131.0, "z": -666.0}
        ,"dimension":{"x": 4.0, "y": 36.0, "z": 38.0}
        ,"rotation":0}
        {"typePoint":"END_POINT"
        ,"position":{"x": 148.0, "y": 133.0, "z": -696.0}
        ,"dimension":{"x": 50.0, "y": 36.0, "z": 4.0}
        ,"rotation":0
        ,"floor":3
        ,"part":1}
        {"typePoint":"END_POINT"
        ,"position":{"x": 196.0, "y": 121.0, "z": -688.0}
        ,"dimension":{"x": 44.0, "y": 36.0, "z": 4.0}
        ,"rotation":0
        ,"floor":2
        ,"part":1}
        /* ... */
        ]
}
```

Código 4.10: Archivo *JSON* correspondiente al mapa de un edificio.

El constructor de la clase *Building*, será el encargado de manejar los datos obtenidos del archivo JSON gracias a la clase *JsonReader,* la cual analiza el archivo y construye un *DOM (Document Object Model)* de objetos *JsonValue* (Reading & writing JSON, s.f.). El constructor también es encargado de almacenar la colección de pisos en una variable *JsonValue* y cargar los puntos de carga del edificio gracias a la función *loadTriadPoint*.

La clase *Building* cuenta con el método *loadFloor*, cuya principal función es la carga de un piso dado a los parámetros especificados al momento de invocar al método. El piso cargado se almacena en la variable *currentFloor* y enseguida se inicia la ejecución de un hilo cuyo propósito es generar una transición más amigable para el usuario, porque puede existir el caso (sobre todo en dispositivos móviles), donde la carga inmediata de un modelo

disminuya drásticamente el cuadro por segundo, presentando así una fluidez inconsistente al usuario. En base a una serie de pruebas se concluyó que la implementación de la transición entre pisos, es el resultado más favorable para la aplicación. En la Figura 4.17 se presenta una muestra de la transición resultado de una carga de piso.

Figura 4.17: Carga de un piso.

Aún en la clase *Building*, el método *isFloorFinish* regresará un valor booleano indicando si el hilo ejecutado al cargar un piso ha concluido. En caso de que el hilo se encuentre en ejecución, se llamará al método *getLoading* Status, el cual regresará un valor flotante indicando el porcentaje de carga.

La clase *Floor* la cual hereda de *Complex*, será donde prácticamente la clase padre se encargará de hacer las asignaciones necesarias, únicamente en la clase hija se definirán los modelos permitidos y en el constructor se hará uso de *super*. La palabra clave *super* permite a una subclase referirse de inmediato a una *super* clase (Schildt, 2007).

La última clase que hereda directamente de *Complex* es *Outside*, dicha clase es la encargada de los exteriores de los edificios y se complementa con la clase *Building*, en una dinámica que se explicará posteriormente.

La clase *Nature* hereda de la clase abstracta *AbstractModelObject*, por lo tanto no contará con un cuerpo físico, esto quiere decir que no existirán colisiones entre el personaje y los objetos *Nature's*. Los árboles y el cielo son los únicos modelos definidos como naturaleza, por esta razón el usuario podrá atravesar los árboles o arbustos puestos en el

recorrido al igual que el cielo. Aunque el modelo que define el cielo del recorrido no es alcanzable, ya que es un cubo que envuelve a todos los modelos y por las restricciones que se implementarán en el recorrido, en ningún momento el usuario va a tener mucha cercanía con las caras de este modelo, únicamente lo visualizará a lo lejos. En la Figura 4.18 se puede apreciar la caja de cielo o *Skybox* (sin textura) envolviendo a los demás modelos.

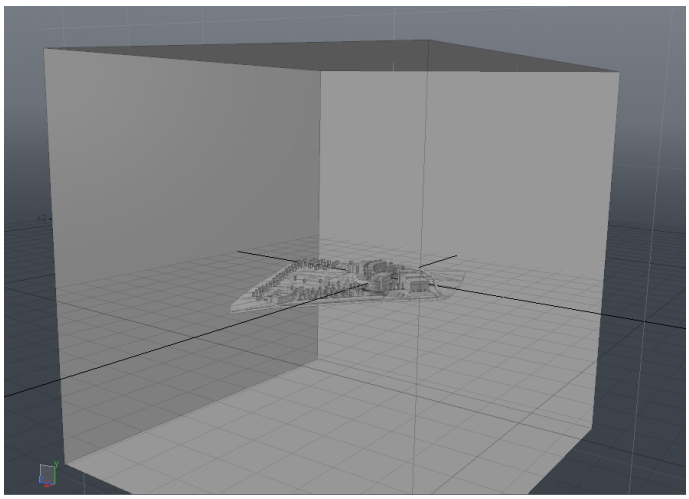


Figura 4.18: Modelo *Skybox.*

En el código 4.12 se presenta un ciclo entre los materiales del modelo implementado en el constructor de la clase *Nature*. En dicho ciclo, a cada material se le agrega un atributo *FloatAttribute.AlphaTest,* usado para descartar pixeles cuando el valor alfa es igual o menor que el especificado (Material and environment, s.f.); esto quiere decir que detectará que parte de la imagen de la textura es transparente o no y dibujará en base a ello.

```
for(Material material : model.materials) {
        material.set(FloatAttribute.createAlphaTest(0.90f));
}
```

Código 4.11: Ciclo entre materiales.

Se planteó desde un principio un recorrido limitado donde los puntos de interés se encuentran en una área de la facultad, desafortunadamente existen más instalaciones de la facultad de Informática que no se pudieron cubrir y por este motivo es necesario definir los límites a aquellos puntos que si se cubrieron. Para esta tarea se cuenta con la clase *Constraint*, la cual cuenta con una única instancia y un modelo ya definido en el método inicializador.

Este modelo cubre aquellas partes que restringen al usuario de caer en áreas no contempladas en el recorrido.

4.4.4. PUNTOS DE CARGA

En la aplicación existirá una dinámica de puntos de carga que ayudarán al óptimo rendimiento del programa. Existirán dos grupos de puntos de carga, cada grupo estará conformado por dos tipos de puntos de carga. En la Figura 4.19 se puede entender la simple jerarquía que corresponde a los puntos de carga.

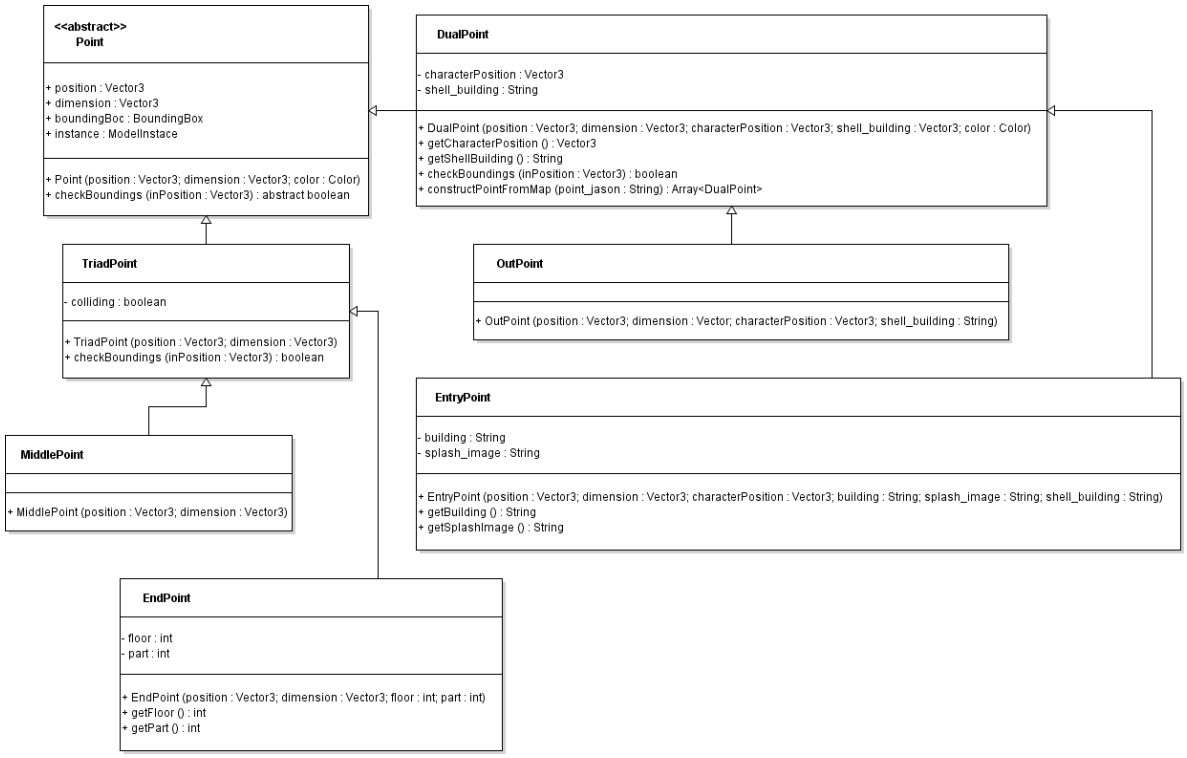


Figura 4.19: Clases de puntos de carga.

Los dos grupos de puntos de carga heredarán de *Point*, dicha clase es la encargada de crear una caja dado los parámetros proporcionados por el constructor de la clase; un vector que contiene la posición y otro sus dimensiones serán datos suficientes para crear una figura primitiva. Se utiliza la clase *BoundingBox* la cual es la encargada de encapsular una caja delimitadora (Class BoundingBox, s.f.), esta clase permite identificar el posicionamiento del personaje principal y si está contenida en la caja delimitadora que envuelve a la figura primitiva. La rutina que detecte esta colisión debe estar colocada en el método

checkBoundings, cada grupo de punto de carga definirá su respectiva rutina y por eso es importante que *checkBoundings* sea un método abstracto, así como la clase *Point*.

El grupo de puntos de carga *TriadPoint* se utiliza en los interiores de edificios. Sus posiciones y dimensiones se encuentran en un archivo *JSON* correspondiente al mapa del edificio. Como el nombre de la clase lo indica, el uso de esta clase debe estar conformada por tres puntos de carga, dos de este grupo deben ser de tipo *EndPoint* y el punto de carga sobrante será de tipo *MiddlePoint*, formando una Triada de tipo *TriadPoint*. En la Figura 4.19 se puede observar la presencia de un grupo conformado por *TriadPoint's*, utilizando dicha imagen se puede ejemplificar el uso de este concepto: el personaje principal desea usar las escaleras para llegar al segundo piso, por el momento los únicos inmuebles existentes en el espacio tridimensional están en el primer piso, en el momento en que el personaje pasa sobre el primer punto de carga (ubicado en la parte derecha de la imagen y etiquetado como "Final") se detecta un primer contacto, si el personaje sigue su camino y llega al punto de carga de en medio, lo que sucederá será que todos los complejos interiores del edificio desaparecerán. En este punto, si el personaje decide volver al primer piso, al colisionar de nuevo con el primer punto de carga, este cargará los complejos del primer piso, en caso de que el personaje decida llegar al segundo piso, entonces se cargarán los complejos correspondientes al segundo piso, de esta manera se trata de mantener un control de los modelos existentes en el mundo tridimensional.

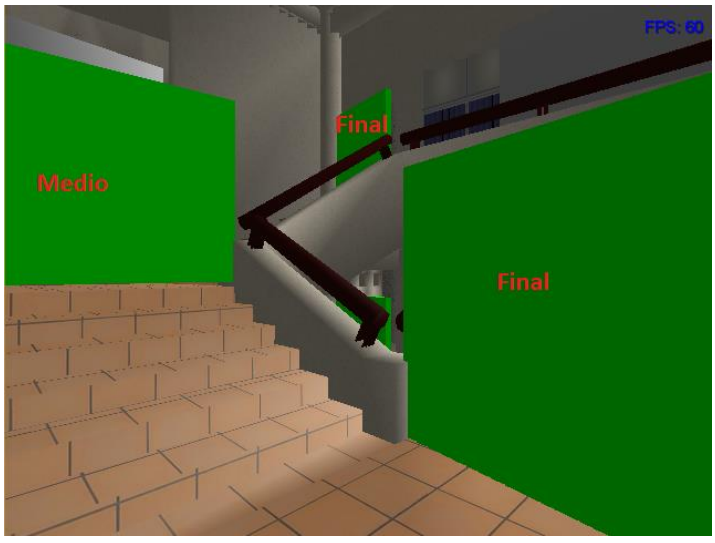


Figura 4.20: Puntos de carga, *TriadPoint's*.

Cabe mencionar que la aplicación de este concepto ideado para el presente programa también es utilizado en partes dentro de un mismo piso, es decir, en un piso pueden existir N partes y para cargar modelos de diferentes partes, se hace uso de la triada de puntos de carga.

Las subclases que heredan de *TriadPoint*: *EndPoint* y *MiddlePoint*, no tienen mayor definición, el único rasgo diferente radica en la clase *EndPoint,* la cual contiene variables numéricas indicando el número de piso y el número de parte.

El grupo de puntos de carga *DualPoint* se utiliza en los exteriores de edificios. Las posiciones y dimensiones que conforman los puntos de carga *DualPoint's* se encuentran en un único archivo *JSON* y se construyen a partir del método estático perteneciente a la clase *DualPoint*.

En la clase *DualPoint* se plantea el uso de dos variables, una contiene la posición del personaje y la otra contiene la ruta del cascarón del edificio. En el recorrido, un edificio estará conformado por dos modelos diferentes, el interior del edificio y el exterior del edificio (o cascarón de edificio); cuando el personaje está posicionado afuera de un edificio, es decir, en el exterior, y observa un edificio o conjunto de edificios por fuera, se puede notar que son modelos simples carentes de detalle. En el momento en que el personaje ingresa a un edificio, se desecha el exterior y se carga el interior del edificio, el cual cuenta con más detalle y el número de polígonos del modelo es mayor al del exterior.

Al igual que en la clase *TriadPoint*, *DualPoint* cuenta con su propia rutina para definir la respuesta en el método *checkBoundings*. En la Figura 4.21 se puede apreciar una representación de los puntos de carga *DualPoint,* donde las líneas verdes son los puntos de entrada y las líneas azules son los puntos de salida.

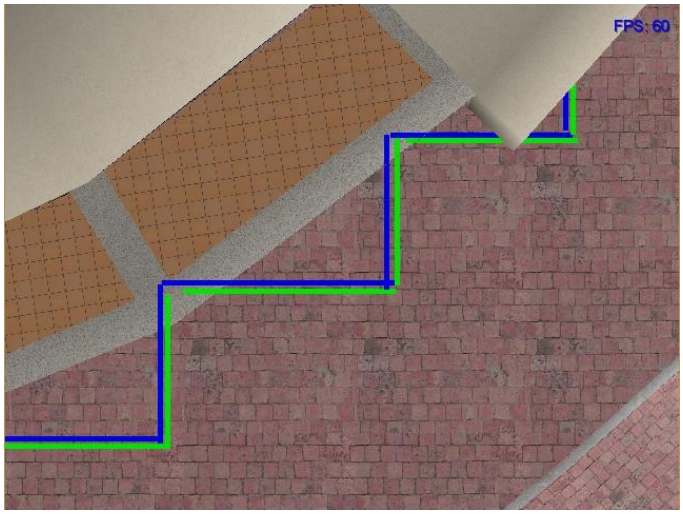


Figura 4.21: Puntos de carga, *DualPoint's*.

En la clase *EntryPoint* se encuentran dos variables que definen el comportamiento del ingreso a un edificio. La primera variable cuenta con el modelo del interior del edificio, el cual se cargará al ingresar; la segunda variable indica la imagen que se utilizará en la transición de exterior a interior. Por último, en la clase *OutPoint* no existe mayor complejidad y solo se utiliza la palabra clave *super*, en referencia al constructor de la superclase *DualPoint*.

4.4.5. CÁMARAS

Como se mencionó con anterioridad, existen dos tipos de cámaras: ortogonal y de perspectiva. La cámara correspondiente a la vista en primera persona del personaje es de perspectiva, como se ha manejado a lo largo del presente trabajo. Se agregan dos cámaras ortogonales, una cámara enfocada a la interfaz gráfica de la aplicación y la otra para propósitos de depuración.

Figura 4.22: Cámara en primera persona del personaje (izquierda) y cámara de depuración (derecha)

La cámara de personaje es *Singleton*, únicamente existirá un personaje, y por lo tanto una cámara. Es importante dotar de un número alto al componente *far*, ya que este permitirá visualizar la caja de cielo cuyas caras se encuentran a una larga distancia de los modelos principales.

La clase *CameraCharacter* cuenta con los métodos *moveLeft*, *moveRight*, *moveDown* y *moveUp*, dichos métodos permitirá rotar al usuario en dirección izquierda, derecha, hacia abajo y hacia arriba respectivamente. También cuenta con el método *updateCamera*, el cual será llamado por la clase *Character* y actualizará la posición de la cámara de acuerdo a la del personaje.

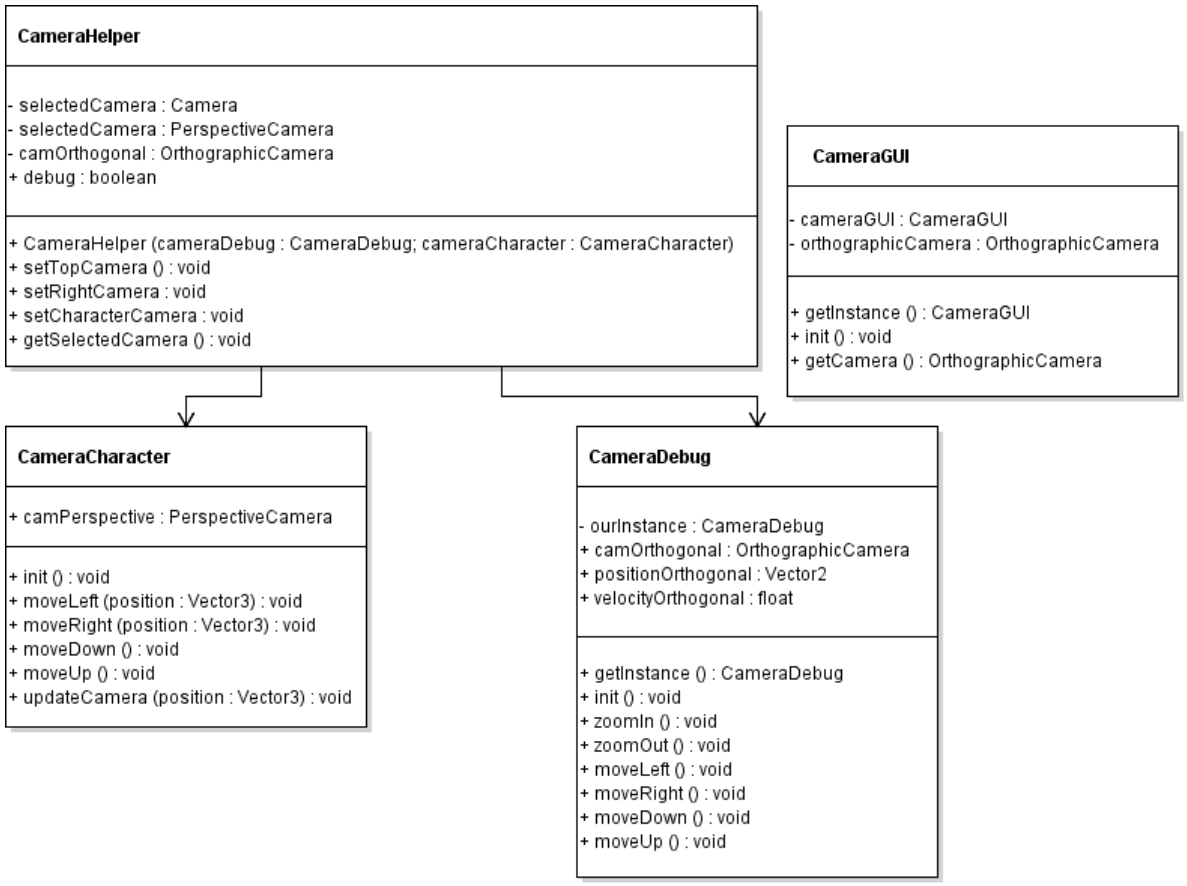


Figura 4.23: Clases de cámara.

Para el desarrollo del mundo tridimensional fue necesario la creación de una clase que permitiera una visualización en diferentes perspectivas. La clase *CameraDebug* permite

dicha tarea manejando una cámara ortogonal y contando con métodos que permitan el cambio de posición dicha cámara. Los métodos *moveLeft*, *moveRight*, *moveDown* y *moveUp* permiten el cambio de posición hacia la izquierda, derecha, hacia abajo y hacia arriba respectivamente; también se cuenta con el método *zoomIn* y *zoomOut*, estos métodos permiten un acercamiento o un alejamiento de la cámara. Cabe mencionar que al igual que la clase *CameraCharacter*, solamente se requiere una instancia de *CameraDebug*.

La clase *CameraHelper* se encarga de la gestión entre la cámara de perspectiva encargada por el personaje, y la cámara de depuración. Ambas cámaras son de diferente tipo de objeto, pero ambas heredan de la clase Camera; por esta razón existe que en la presente clase, por medio del método *getSelecetedCamera*, se devuelva un objeto Camera, la cual puede ser Ortogonal o de perspectiva. Dentro de *CameraHelper* existe el método *setTopCamera* y *setRightCamera*, los cuales sirven para cambiar de perspectiva la cámara ortogonal; también existe el método *setCharacterCamera*, el cual selecciona a la cámara de perspectiva del personaje.

Por último se cuenta con la clase *CameraGUI*, esta clase maneja la cámara ortogonal con la que se presentará la interfaz gráfica de usuario. Su consulta será frecuente en la creación o inicialización de varias clases y será la primera cámara en usarse.

4.4.6. PANTALLAS

Se plantea el uso de tres pantallas en la aplicación, dos de ellas serán introductorias y la tercera prevalecerá hasta el fin de la aplicación. Se explicarán aquellas clases que manejan diferentes pantallas según su orden de aparición.

La interfaz *Screen* será implementada en las tres clases referente a pantallas. *Screen* contiene varios métodos similares a *ApplicationListener* e incluye un par más como *show* y *hide* (Extending the simple game, s.f.).

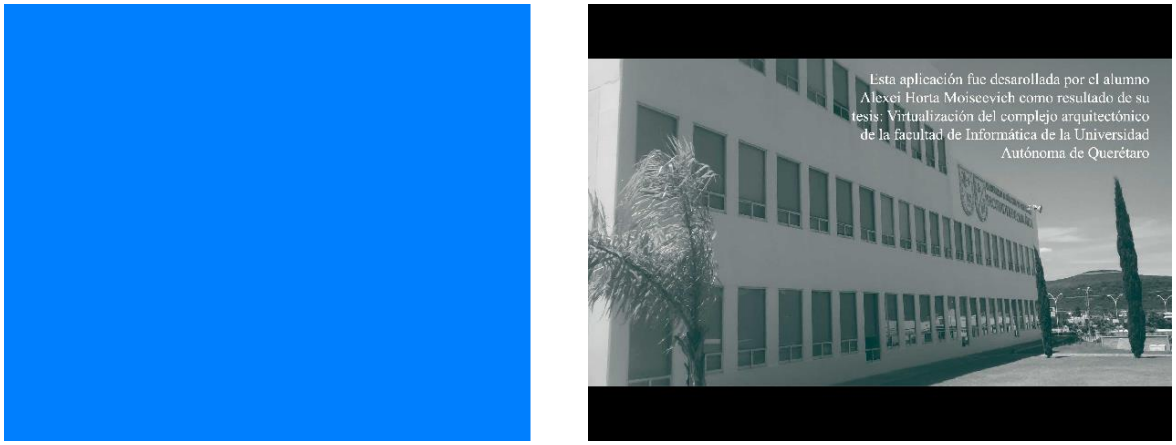


Figura 4.24: Pantalla de carga (izquierda) y pantalla de bienvenida (derecha).

En las clases que implementen *Screen* será necesario introducir como parámetro en el constructor, la clase que invocó dichas pantallas. Esta clase que se introdujo como parámetro extiende de la clase *Game*, la cual provee una implementación de *ApplicationListener* para uso del desarrollador, entre otros métodos auxiliares para la definición y manejo de pantallas. La dinámica entre la clase *Game* y *Screen*, crea una simple y poderosa estructura para el proyecto (Extending the simple game, s.f.). En la clase *LoadScreen* se puede apreciar el ingreso de la clase procedente *MyGdxGame*.

El método show en la clase *LoadScreen* es equivalente a *create*, en dicho método se configura la matriz de proyección de la variable *shapeRenderer* y se asigna a cero la variable flotante *progress*, donde se registrará el proceso que haga la carga de recursos contenidos en *Asset*.

En el método *render* de la clase *LoadScreen*, se actualiza la variable *progress* dado al método *getProgress* perteneciente a la clase *Asset; s*e puede visualizar una animación fluida de la representación gráfica del progreso gracias al método de interpolación lineal *lerp* perteneciente a la clase *MathUtils*, y a la figura dibujada por *shapeRenderer*.

Al término de la carga de recursos se invoca al método *setScreen* perteneciente a la clase *Game*, la cual establece la pantalla actual y se indica que *SplashScreen* será la pantalla que aparecerá después de *LoadingScreen*. Por esta razón es importante incluir la clase procedente en el constructor.

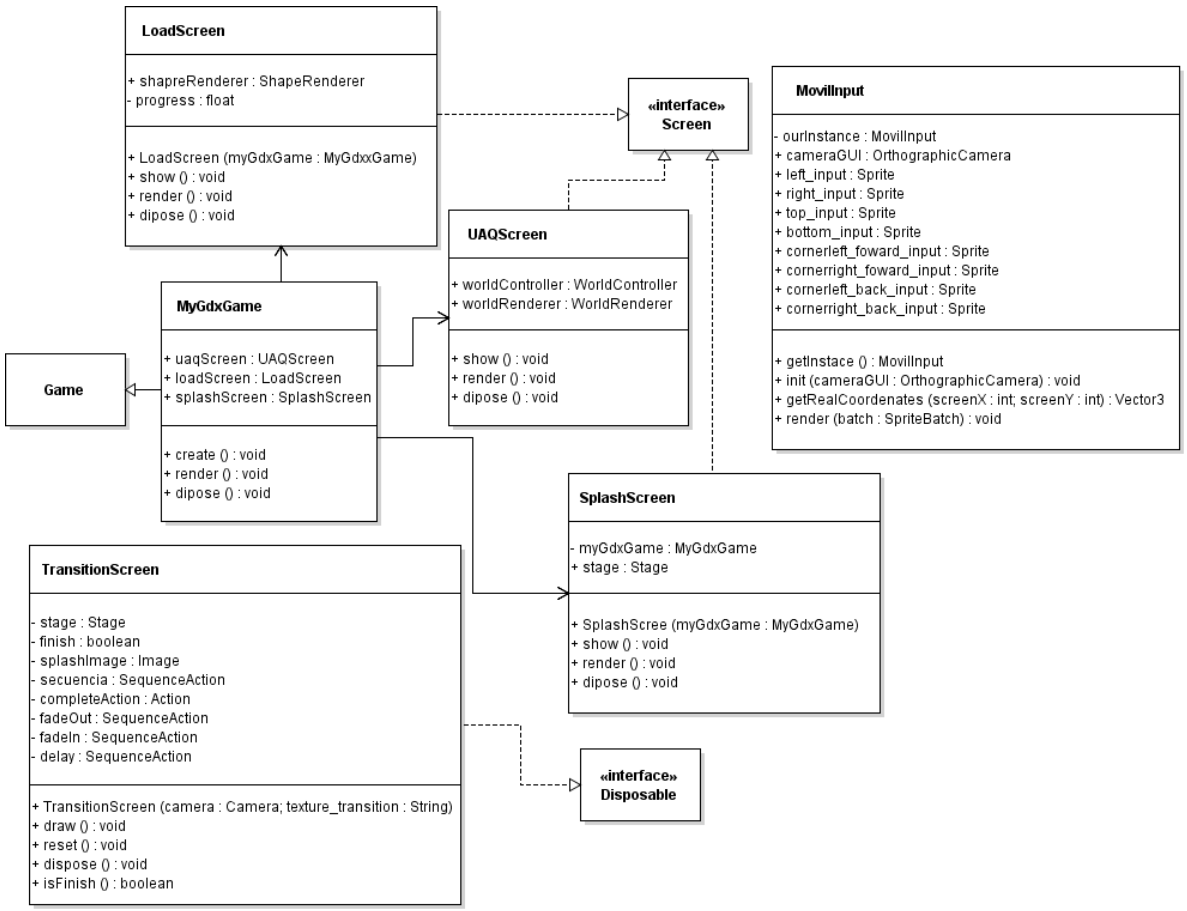


Figura 4.25: Clases de pantalla.

La clase *SplashScreen* es similar a *LoadScreen*. El constructor cuenta con un parámetro de tipo *MyGdxGame*. En el método show se inicializa un objeto de tipo *Stage*, esta clase maneja el dibujado y el ingreso de datos para todos los actores pertenecientes al escenario (Bose, 2014). Se agrega como actor, una imagen de tipo *Image*; un actor "es un nodo en el gráfico con una posición, tamaño, origen, escala, rotación y color. Cada nodo Actor contiene su propio sistema de coordenadas" [Bose J. 2014]. Por último, se agrega como acción una variable de tipo *SecuenceAction*, esta clase ejecuta un número de acciones una por una según el orden en que se defina (Class SequenceAction, s.f.). Las acciones son utilizadas para proveerles secuencias animadas a los actores, en el caso de *SplashScreen*, se define una espera de 6 segundos y enseguida se ejecuta la acción *completeAction*, en la cual se invoca al método *setScreen* para efectuar un cambio a la pantalla *UAQScreen*.

La clase *Stage* cuenta con el método *draw*, la cual dibujara todo lo perteneciente al escenario (Bose, 2014), este método se coloca en *render*.

Para concluir la triada de clases que implementan la interfaz *Screen*, en la clase *UAQScreen* no es necesario crear un constructor como en las demás clases, ya que la presente clase será la última y no existirán mecanismos para regresar a las pantallas anteriores. *UAQScreen* presenta todo el contenido y las dinámicas que se programaron con ayuda de dos clases: *WorldController* y *WorldRenderer*. Ambas clases son complejas y se abordarán posteriormente, solo queda mencionar que en el método *render,* se invocará al método *update* perteneciente a la clase *WorldController,* y al método *render* perteneciente a la clase *WorldRenderer*.

Para el manejo de pantallas se requiere una clase que extienda de *Game* y defina la pantalla inicial (*LoadScreen*), este es el trabajo de *MyGdxGame*. En dicha clase se inicializa objetos de tipo *UAQScreen, LoadScreen, SplashScreen, CameraGUI* y *Asset*.

La clase *TransitionScreen* visualiza una pantalla de transición que se activa en el momento en que el usuario se adentra o se sale de un edificio. *TransitionScreen* no implementa una interfaz *Screen,* ya que existe dentro de la pantalla *UAQScreen,* y parecido a la pantalla *SplashScreen*, se dibuja un escenario con un par de acciones más en la secuencia de acciones. Al término de la animación se indica en una variable booleana que la transición ha acabado.

Cuando se pretenda ejecutar una pantalla de transición, será necesario invocar al método *reset*; en este método, cada acción se debe de resetear permitiendo un rehúso de dichas acciones.

Será necesario dibujar una interfaz para los dispositivos móviles, una interfaz de usuario que cuente con una serie de botones para permitirle al personaje moverse a través del mundo creado. Esta interfaz de usuario será creada en la clase *MovilInput*, en ella se manejan ocho variables de tipo *Sprite*, esta clase describe una región de textura y la geometría donde será dibujada (Spritebatch, Textureregions, and Sprites, s.f.).

Figura 4.26: Controles móviles.

En el método *init* se les da una posición y un tamaño (en base a la cámara *cameraGUI)* a cada una de las variables de tipo *Sprite*, de esta manera se acomodan las piezas para que estén bien posicionadas y no se transpongan entre ellas. En la Figura 4.26, donde se puede apreciar el resultado final, se tiene en las esquinas superiores, botones que accionan un avance del personaje, y en las esquinas inferiores, botones que accionan un retroceso del personaje. Las flechas que aparecen del lado superior, inferior, izquierdo y derecho, accionan el cambio de dirección de la cámara al lado en que se especificó.

También se cuenta con el método *getRealCoordenates,* el cual devolverá las coordenadas necesarias para detectar registro de tacto en los *Sprite's* contenidos en *MovilInput*. Por último, en el método *render* se dibujan los *Sprites's* contenidos en *MovilInput.*

4.4.7. PUNTOS DE INFORMACIÓN

Dentro del recorrido, se planteó desde un principio presentar información sobre los puntos de interés que podrá consultar el usuario. Para esto se creó la clase *Info*, la cual maneja el modelo tridimensional representando un punto de información, también maneja los componentes de la interfaz desplegada al seleccionar el punto de información. En la Figura 4.27 se puede ilustrar la representación y utilidad del punto de información.

En el constructor de la clase *Info* se configura las propiedades del modelo y se le asigna una posición según el ingreso al parámetro *position*. También se inicializa un objeto

ShapeRenderer, el cuál dibujará un cuadro de fondo que abarcará toda la pantalla, y por último, un objeto *Stage* donde se colocarán los componentes de texto.

Figura 4.27: Punto de información (izquierda) e Interfaz desplegada al seleccionar el punto de información (derecha).

En el método *createControls* perteneciente a *Info*, se crean los componentes que pertenecerán al escenario *stage*. Son dos tipos de componentes: componentes de texto, creado en el método *createTexts*, y el botón de salida, creado en el método *createOutButton*.

En el método *createTexts* se utilizan dos objetos de tipo *Label*, una para el título y el sobrante para la descripción. Para ambos objetos se configura el estilo de texto y se agregan a un objeto de tipo *Table*, esta clase maneja un grupo de controles que establecen el tamaño y posición de sus hijos usando una tabla lógica, similar a las tablas HTML (Table, s.f.); de esta manera se colocan los cuadros de texto ocupando la posición y el tamaño que se puede apreciar en la parte derecha de la Figura 4.27.

En el método *createOutButton* se crea un objeto de tipo *TextButton*, un botón con un texto integrado. Al igual que *createText*, se agrega el componente a una tabla y se devuelve un objeto *Table* al término del método. En el botón se especifica que en el momento en que ocurra un cambio, se detone el método *outOfInfo*, cambiando el valor de la variable *isInfo* perteneciente a la clase *WorldController* e indicando que la información ya no debe presentarse.

Por último, el método *constructInfoFromMap* construye los puntos de información a partir de un archivo *JSON*; se especifica una posición, el texto del título y el texto de descripción en cada nodo *JSON*.

4.4.8. LÓGICA Y RENDER

Se explicó la serie de clases necesarias para poblar un mundo tridimensional, ahora se tiene que crear la clase que contendrá los complejos, la naturaleza, etcétera. En la clase *World* se tienen una serie de variables que conformarán el mundo tridimensional que el usuario visualizará; la variable más importante de todas es *character* y es de tipo *Character*, esta variable únicamente se inicializará en *World*.

Se cuenta con una variable *complex* de tipo *Complex*, en la cual albergará el modelo de los inmuebles actuales, esta variable cambia frecuentemente.

La colección de puntos *DualPoint's* es contenida en la variable *dualPoints*, la cual se construye en el método *init* de la presente clase a través del constructor *constructPointFromMap*, ingresando como parámetro la dirección de un archivo *JSON*.

La colección de puntos *TriadPoint's* es contenida en la variable *triadPoints*, dicha variable se asigna al adentrarse al interior de un edificio.

La colección de complejos exteriores es contenida en la variable *outsides*, en el método *init* de la presente clase se nutrirá dicha colección y será frecuentemente modificada.

Las variables *skyBox* y *trees*, ambas de tipo *Nature*, se inicializarán en el método *init* de la presente clase.

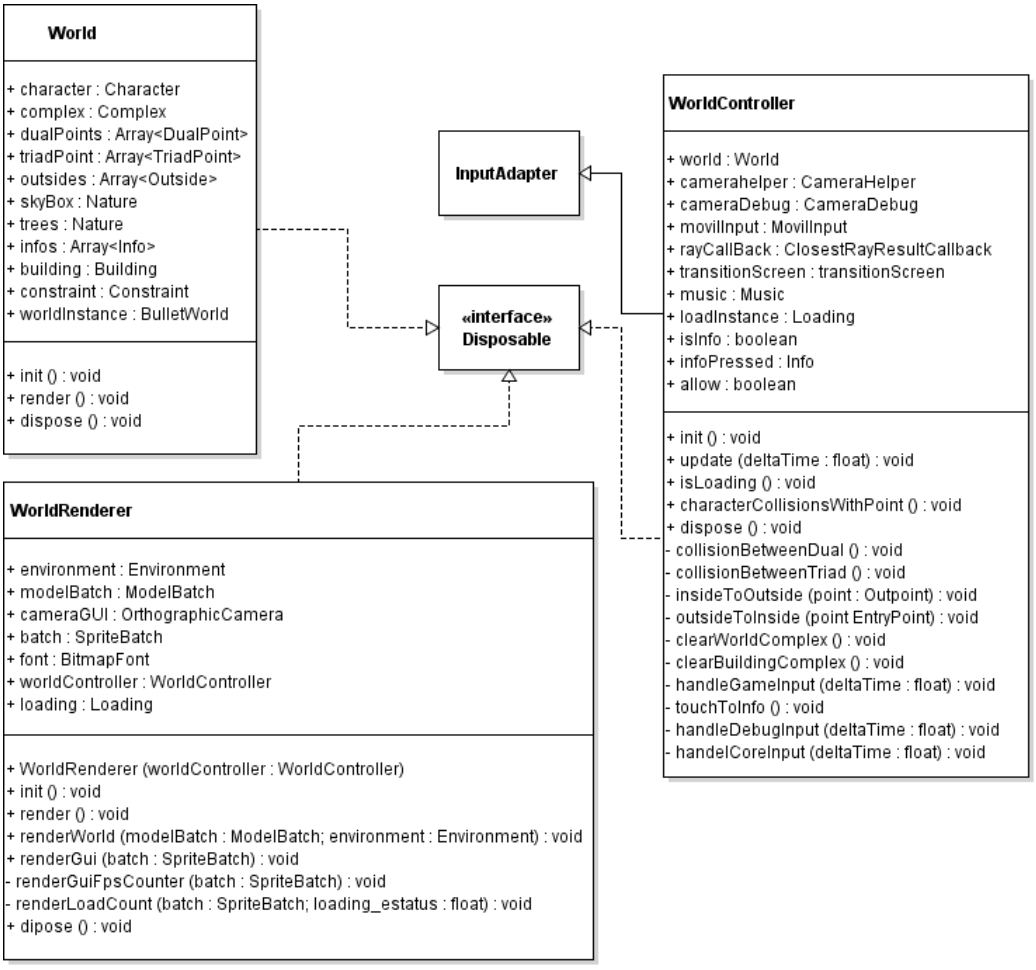


Figura 4.28: Clase *World, WorldController* y *WorldRenderer*.

Se encontrará una colección de tipo *Info*, la cual se construye en el método *init* de la presente clase por medio del método *constructInfoFromMap*.

En la clase *World* también se cuenta con la variable *building* de tipo *Building*, dicha variable contendrá el interior de un edificio, al principio se le asigna un valor nulo.

También se cuenta con la variable *constraint* de tipo *Constraint*, la cual contendrá las restricciones impidiendo al usuario alcanzar ciertas áreas.

Por último, se cuenta con la variable *worldInstace* de tipo *BulletWorld*, esta variable contiene las mecánicas del mundo físico y es necesaria inicializarla para que existan comportamientos aplicando las leyes de la física en el recorrido. En el método *init* de la presente clase, se utilizan los métodos pertenecientes a *BulletWorld: addBulletComplex*,

addBulletConstraint y *addBulletMainCharacter*, para realizar asignaciones de algunos objetos al mundo físico.

Se requerirá una clase que contenga toda la lógica del producto, aquella que inicializará y modificará el mundo tridimensional, esta clase se llamara *WorldController*.

Es necesario extender la clase *WorldController* a *InputAdapter*, la cual hereda de *InputProcessor*, necesaria para recibir eventos de ingreso desde una computadora o un dispositivo móvil (Interface InputProcessor, s.f.); En el método *init* se establece *WorldController* como el procesador de ingreso por medio del método *setInputProcessor*. Enseguida se inicializa la instancia de carga, el mundo tridimensional, la cámara de depuración y la cámara de ayuda. También en el mismo método se reproduce la música y se establece una repetición infinita de dicha música.

El método *update* perteneciente a la clase *WorldController* es de suman importancia, aquí ejecutará toda la lógica programada del programa. Se empieza con una condición, en caso de que se encuentre cargando el programa se dirige al método *isLoading*, donde se encuentra el tipo de carga que está ocurriendo y actúa según la correspondiente. Si no se está cargando, entonces se actualiza la física del mundo; también se detecta ingreso alguno por el método *handleCoreInput* y por último se detecta colisiones entre el personaje y los puntos de carga.

En el método *handleCoreInput* se tienen las primeras instrucciones comentadas impidiendo que en el producto final se pueda ingresar al modo de depuración. *HandleGameInput*, aquel método que controlará el ingreso del usuario, tiene dos modalidades: en caso que se detecte una computadora de escritorio o un dispositivo móvil con sistema operativo Android. Si fuera una computadora de escritorio, los controles están conformados por las flechas indicando la dirección a la que viajara el personaje y las teclas N y M, para rotar hacia arriba o hacia abajo la cámara del personaje. En caso de seleccionar y presionar botón derecho del mouse dentro de la pantalla se accionará el método *touchToInfo*.

En el método *init* perteneciente a la clase *WorldController*, se inicializo la variable *rayCallBack* de tipo *ClosestRayResultCallback*, clase la cual hereda de *RayResultCallback* y

devuelve la colisión más cercana que el rayo detecta desde una locación de inicio (Dickinson 2013).

La técnica *Raycasting* puede emplearse para encontrar objetos debajo del cursor desde la cámara de perspectiva (Dickinson, 2013), esta técnica se usará para detectar el ingreso del usuario al tocar un cubo de información, ya sea desde una computadora de escritorio o un dispositivo móvil.

La clase *ColestRayResultCallback* cuenta con el método *hasHit*, la cual regresa un valor booleano indicando si hubo una colisión entre el rayo y algún objeto con cuerpo físico. Al detectar colisión alguna, se compara si ha sido con un cubo de información, en caso de que sea así, se guarda el objeto de tipo *Info* en la variable *infoPressed* y se indica si la interfaz del cubo de información debe presentarse por medio de la variable de tipo booleana *isInfo*.

En el método *handleCoreInput*, en caso de ser un dispositivo móvil, se detecta si las coordenadas que ingreso el usuario a tocar la pantalla están contenidas en alguno de los múltiples objetos *Splash's* que maneja la clase *MovilInput*. Se colocó esta rutina adentro de un ciclo permitiendo múltiple ingreso de coordenadas.

El método *characterCollisionsWithPoints* detecta dos tipos de colisión: entre puntos de carga *Triad* y puntos de carga *Dual*. La colisión entre el personaje y los puntos de carga *Dual* es encargada por el método *collisionBetweenDual* y en esta se analiza si se ha colisionado entre un punto de salida (*OutPoint*) o un punto de entrada (*EntryPoint*).

En el método *insideToOutside* se maneja la colisión entre un punto de salida, se cambia de posición al personaje, se elimina los complejos del piso actual del edificio del que se está saliendo y se asigna la variable *complex* de *World* a los complejos exteriores. Enseguida, se elimina el interior del edificio, se regresa el exterior del edificio del que se está saliendo y se integra a la colección *outsides*, indicando el tipo de carga como *OutPoint* y eliminando los puntos de carga *Triad's*. Por último, se establece la pantalla de transición respecto a la salida de cualquier edificio.

En el método *outsideToInside* se maneja la colisión entre un punto de entrada, se cambia de posición al personaje, se elimina los complejos exteriores y se asigna los

complejos del primer piso del edificio que se ingresará a la variable *complex* de *World*. También se asigna el nuevo edificio a la variable *building* de *World*, se agregan los puntos de carga, se elimina el exterior del edificio al cual se ingresó, se indica el tipo de carga como *EntryPoint*; por último, se establece la pantalla de transición de acuerdo a la información que posea el punto de carga de entrada.

La colisión entre el personaje y los puntos de carga *Triad* es manejada por el método *colisionBetweenTriad*, en este método se analiza si se ha colisionado con un punto de carga final (*EndPoint*) o un punto de en medio (*MiddlePoint*). Si se ha colisionado con un *MiddlePoint*, se vaciará el contenido de la variable *complex* perteneciente a *World*. En caso de que el personaje haya colisionado con un punto de carga final, se asignará complejos pertenecientes a algún piso interior.

Queda únicamente *renderizar* lo creado y actualizado en la lógica de la aplicación, para esto se utilizará la clase *WorldRenderer*. El método *render* de la presente clase será de suma importancia y en este se llamarán dos métodos: *renderWorld* y *renderGui*.

En el método *renderWorld*, únicamente se invoca *render* perteneciente a *World*, dibujando el entorno tridimensional que se ha creado.

Por último, en el método *renderGui*, en caso de que existiera un estado de carga de tipo *TriadPoint* o *EntryPoint*, se llama al método *renderLoadCount*, el cuál presenta en pantalla un conteo en porcentajes. En caso de que el estado de carga sea de tipo *OutPoint*, se dibuja la pantalla de transición.

Si la clase *WorldController* indicara que debe presentarse información de la caja de información, se dibuja la interfaz por medio de la variable *infoPressed*.

Por último si se detectara un dispositivo móvil con sistema operativo *Android*, se invoca al método *render* perteneciente a la clase *MovilInput* para presentar en pantalla los controles para dispositivos móviles.

Cabe mencionar que se cuenta con el método *renderGuiFpsCounter* para propósitos de prueba, dicho método dibuja un conteo del cuadro por segundo obtenido. Los resultados obtenidos gracias a este método se presente en el siguiente capítulo.

5. RESULTADOS Y DISCUSIÓN

El producto final obtenido de esta investigación se materializa en un archivo *APK* para dispositivos móviles, y un archivo *JAR* para computadoras de escritorio. En la Figura 5.1 se puede apreciar la ejecución de la aplicación en un dispositivo móvil con sistema operativo Android.

La aplicación para dispositivos móviles es similar al programa destinado a computadoras de escritorio, con la excepción de los controles de usuario y del rendimiento monitoreado por medio de un conteo de cuadros por segundo.

Figura 5.1: Recorrido virtual ejecutado en un dispositivo móvil.

El conteo de cuadro por segundo que se registró será de suma importancia ya que un *fps* bajo conllevaría a que la aplicación no se comporte como lo esperado. Por esta razón, se presentan una colección de registros del cuadro por segundo que tuvo el recorrido virtual en diferentes áreas de la facultad y en distintas plataformas.

Se presenta el registro de *fps* que se obtuvo en el área exterior de la facultad. En la Tabla 5.1 se pueden apreciar los resultados.

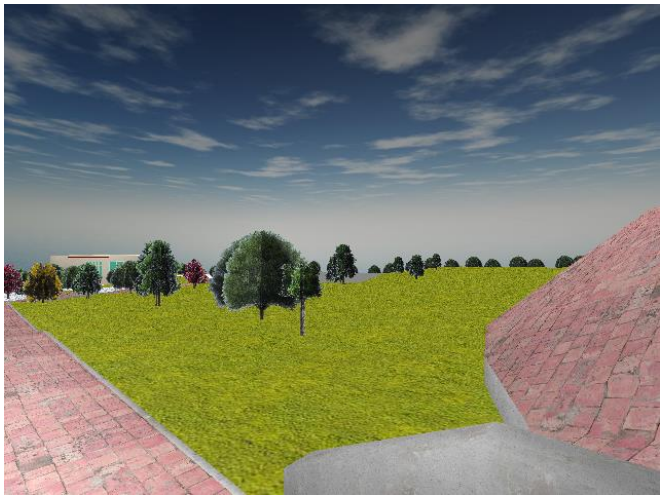


Figura 5.2: Exterior.

Tabla 5.1. Cuadro por segundo monitoreado en el área exterior en la aplicación.

	Celular Galaxy Grand Prime	Celular HTC One	Computadora Inspiron 5737
Exterior	32-35	32-35	55-60

Se realizaron los interiores de tres edificios, cada uno cuenta con diferente número de polígonos en sus modelos, diferente número de texturas aplicadas y diferente número de inmuebles presentados. Es por eso que se analiza el cuadro por segundo detectado en el interior de todos los edificios. Ocurre el caso de que en un edificio, un piso contiene más modelos que el otro, por esta razón, en la Tabla 5.2 se divide un edificio en partes y en pisos.

El número de edificio en la Tabla 5.2 es en base a la Figura 5.3, donde se puede observar una numerología sobre cada uno de los edificios que corresponden al orden de realización.

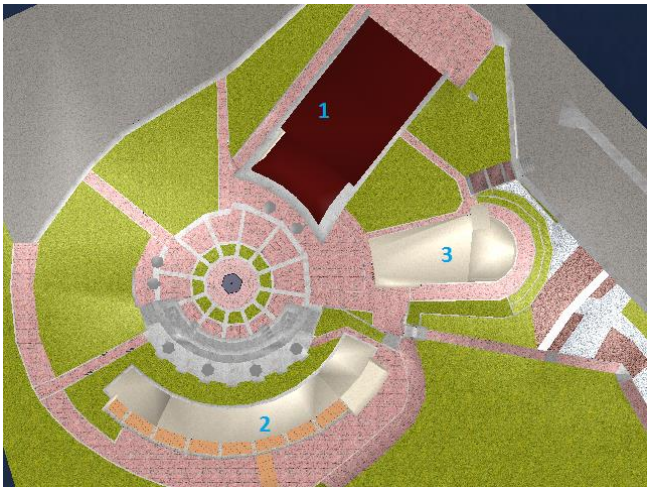


Figura 5.3: Edificios.

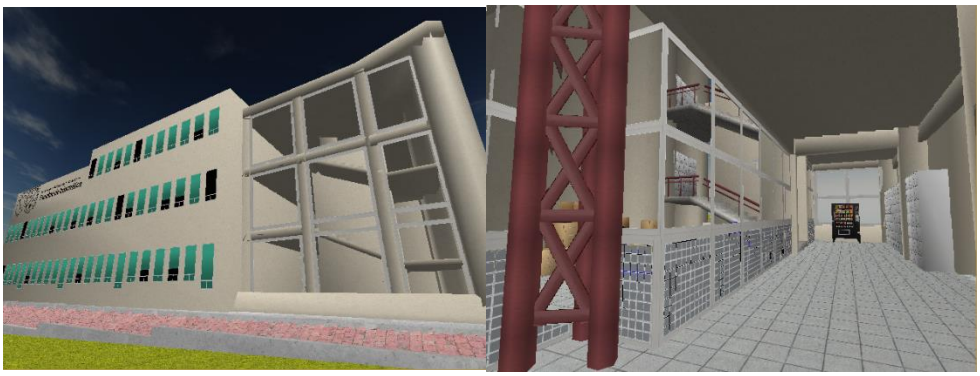


Figura 5.4: Edificio 1, exterior (izquierda) e interior (derecha).

Figura 5.5: Edificio 2, exterior (izquierda) e interior (derecha).

Figura 5.6: Edificio 3, exterior (izquierda) e interior (derecha).

Tabla 5.2 Cuadro por segundo monitoreado en los interiores de edificios en la aplicación.

Edificio – Piso - Parte	Celular Galaxy Grand Prime	Celular HTC One	Computadora Inspiron 5737
1 – 1 – 1	19-23	19-23	55-60
1 – 1 – 2	19-23	19-23	55-60
1 – 2 – 1	19-23	19-23	55-60
1 – 3 – 1	15-23	15-23	55-60
2 – 1 – 1	19-23	19-23	55-60
2 – 1 – 2	19-23	19-23	55-60
2 – 2 – 1	15-21	15-21	55-60
2 – 2 – 2	15-21	15-21	55-60
2 – 3 – 1	15-21	15-21	55-60
3 – 1 – 1	31-42	31-42	55-60
3 – 2 – 1	31-42	31-42	55-60
3 – 3 – 1	31-42	31-42	55-60

Como se puede apreciar en la Tabla 5.2, el rendimiento en un dispositivo móvil varía en ocasiones y el conteo de cuadros por segundo puede llegar a un nivel bajo. Es importante optimizar las aplicaciones orientadas a teléfonos celulares tratando de hacer un ahorro en los recursos del programa.

En el presente trabajo se redujo el número de polígonos de los modelos más complejos, se comprimieron varias imágenes, se desarrolló una rutina para que la velocidad del personaje sea acorde al cuadro por segundo detectado, y por último, se manejó un sistema de puntos de carga que permite una carga de modelos dinámica. Estas técnicas ayudaron a crear una aplicación que se puede ejecutar en un dispositivo móvil de una capacidad de procesamiento ordinaria.

Se lograron los objetivos planteados; se crearon los modelos previstos (con texturas) y se desarrolló una dinámica de puntos de información donde se puede consultar información pertinente sobre la facultad.

Para concluir, antes de abordar la realización de la aplicación, se abordó la manera en que se debe de programar una aplicación tridimensional en *LibGDX* involucrando conceptos pertenecientes a la tecnología *OpenGL ES*, se abordó la manera en que un motor físico es creado e implementado usando únicamente el *framework LibGDX*, y se abordó como las librearías de física aplicada *Bullet* son implementadas en un proyecto de *LibGDX*; el contenido del presente trabajo hace énfasis al involucramiento del lector con un entendimiento profundo del desarrollo de este tipo de programas. Claro, se podría realizar esta misma aplicación usando únicamente *OpenGL* y *Android*, pero dicho propósito llevaría a realizar una tarea exhaustiva. En el presente trabajo se encontró en *LibGDX* un punto medio entre una tecnología de bajo nivel y una tecnología de alto nivel, el desarrollador tiene la libertad de oscilar entre estos dos niveles, y esa misma libertad hace de *LibGDX*, una tecnología interesante y permisible.

REFERENCIAS BIBLIOGRÁFICAS

Bose, J. (2014). Libgdx game development essentials. Birmingham: Packt Publishing Limited.

Brothaler, K. (2013). OpenGL ES 2 for Android: a quick-start guide. Dallas, TX: Pragmatic Bookshelf.

Chen, B., Huang, F., Lin, H., & Hu, M. (2010). VCUHK: Integrating the Real into a 3D Campus in Networked Virtual Worlds. 2010 International Conference on Cyberworlds. doi:10.1109/cw.2010.25

Dickinson, C. (2013). Learning Game Physics with Bullet Physics and OpenGL. Birmingham: Packt Publishing.

Droettboom, Michael. (2016). Undersanding JSON Shema Realse 1.0. Spase Telescope Science Institute.

Gamma, E., Helm, R., Johnson, R. E., & Vlissides, J. (2016). Design patterns: elements of reusable object-oriented software. Boston, MA: Addison-Wesley.

Hess, D. R. (2010). Blender foundations: the essential guide to learning Blender 2.5. Oxford: Focal.

Madhav, S. (2014). Game programming algorithms and techniques: a platform-agnostic approach. Upper Saddle River, NJ: Addison Wesley.

Martin, R. C. (2003). UML for JAVA programmers. Upper Saddle River: Prentice Hall.

Martin, R. C. (2015). Clean code: a handbook of agile software craftsmanship. Upper Saddle River, N.J: Prentice Hall.

Millington, I. (2010). Game physics engine development: how to build a robust commercial-grade physics engine for your game. Amsterdam: Morgan Kaufmann.

McDermott, W. (2011). Creating 3D game art for the iPhone with unity: featuring modo and Blender pipelines. Burlington, MA: Focal Press.

Murdock, K. L. (2014). Autodesk 3ds Max 2014 bible. Indianapolis: J. Wiley & Sons.

Nair, S. B., & Oehlke, A. (2015). Learning LibGDX game gevelopment: wield the power of the LibGDX framework to create a cross-platform game. Birmingham: Packt Publishing.

Schildt, H. (2014). Java: the complete reference. New York, NY: McGraw-Hill Education.

Sommerville, I. (2000). Software engineering. Harlow: Addison-Wesley.

Song, P., Zhang, A., & Yang, T. (2010). Study and Practice of Virtual Campus Modeling and Touring. 2010 International Conference on Multimedia Technology. doi:10.1109/icmult.2010.5629636

Wright, R. S. (2011). OpenGL superbible: comprehensive tutorial and reference. Upper Saddle River: Addison-Wesley.

Young, H.D., R. A. Freedman, A. L. Ford, F. W. Sears, and M. W. Zemansky. (2009). Física universitaria. México D.F: Addison-Wesley.

Zechner, M., and R. Green. (2012). Beginning Android games. New York: Apress.

REFERENCIAS ELECTRÓNICAS

Blender Foundation. (s.f.-a). Introduction. Recuperado Marzo 03, 2017, desde https://www.blender.org/manual/getting_started/about/introduction.html

Blender Foundation. (s.f.-b). Introduction. Recuperado Marzo 04, 2017, desde https://www.blender.org/manual/zh.cn/render/blender_render/textures/introduction.html

Bullet Collision Detection & Physics Library: btActionInterface Class Reference. (s.f.). Recuperado Marzo 03, 2017, desde http://bulletphysics.org/Bullet/BulletFull/classbtActionInterface.html

Bullet Collision Detection & Physics Library: btKinematicCharacterController Class Reference. (s.f.). Recuperado Marzo 03, 2017, desde http://bulletphysics.org/Bullet/BulletFull/classbtKinematicCharacterController.html

Collision Shapes. (2013). Recuperado Marzo 03, 2017, desde http://www.bulletphysics.org/mediawiki-1.5.8/index.php/Collision_Shapes

Coumans, E. (2015). Bullet 2.83 Phisics SDK Manual. Recuperado Marzo 3, 2017, desde https://github.com/bulletphysics/bullet3/blob/master/docs/Bullet_User_Manual.pdf

Creating Geometry. (s.f.). Recuperado Marzo 03, 2017, desde http://modo.docs.thefoundry.co.uk/modo/701/help/pages/modotoolbox/creatinggeometry.html

Extending the simple game. (2016). Recuperado Marzo 03, 2017, desde https://github.com/libgdx/libgdx/wiki/Extending-the-simple-game

Khronos Group. (s.f.). GlClear - OpenGL 4 Reference Pages. Recuperado Marzo 03, 2017, desde https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glClear.xhtml

Loop Slice. (s.f.). Recuperado Marzo 03, 2017, desde http://modo.docs.thefoundry.co.uk/modo/801/help/pages/modotoolbox/tools/loopslice.html

Managing your assets. (2017). Recuperado Marzo 03, 2017, desde https://github.com/libgdx/libgdx/wiki/managing-your-assets

Material and environment. (2015). Recuperado Marzo 03, 2017, desde https://github.com/libgdx/libgdx/wiki/Material-and-environment

Reading & writing JSON. (2017). Recuperado Marzo 03, 2017, desde https://github.com/libgdx/libgdx/wiki/Reading-&-writing-JSON

Rigid Bodies. (2014). Recuperado Marzo 03, 2017, desde http://bulletphysics.org/mediawiki-1.5.8/index.php/Rigid_Bodies

Spritebatch, Textureregions, and Sprites. (2016). Recuperado Marzo 03, 2017, desde https://github.com/libgdx/libgdx/wiki/Spritebatch,-Textureregions,-and-Sprites

Table. (2015). Recuperado Marzo 03, 2017, desde https://github.com/libgdx/libgdx/wiki/Table

The application framework. (2017). Recuperado Marzo 08, 2017, desde https://github.com/libgdx/libgdx/wiki/The-application-framework

The life cycle. (2017). Recuperado Marzo 08, 2017, desde https://github.com/libgdx/libgdx/wiki/The-life-cycle

UV Unwrap. (s.f.). Recuperado Marzo 03, 2017, desde http://modo.docs.thefoundry.co.uk/modo/801/help/pages/modotoolbox/vmaps/unwraptool.html

What Is 3D? (s.f.). Recuperado Marzo 03, 2017, desde http://modo.docs.thefoundry.co.uk/modo/701/help/pages/gettingstarted/whatis3d.html

Working with UV Maps. (s.f.). Recuperado Marzo 03, 2017, desde http://modo.docs.thefoundry.co.uk/modo/601/help/pages/modotoolbox/workingwithuvmaps.html

Zechner, M. (2017). Class BoundingBox. Recuperado Marzo 03, 2017, desde https://libgdx.badlogicgames.com/nightlies/docs/api/com/badlogic/gdx/math/collision/BoundingBox.html

Zechner, M. (2017). Class ModelInstance. Recuperado Marzo 3, 2017, desde https://libgdx.badlogicgames.com/nightlies/docs/api/com/badlogic/gdx/graphics/g3d/ModelInstance.html

Zechner, M. (2017). Class SequenceAction. Recuperado Marzo 03, 2017, desde https://libgdx.badlogicgames.com/nightlies/docs/api/com/badlogic/gdx/scenes/scene2d/actions/SequenceAction.html

Zechner, M. (2017). Class Vector3. Recuperado Marzo 03, 2017, desde https://libgdx.badlogicgames.com/nightlies/docs/api/com/badlogic/gdx/math/Vector3.html

Zechner, M. (2017). Interface InputProcessor. Recuperado Marzo 03, 2017, desde https://libgdx.badlogicgames.com/nightlies/docs/api/com/badlogic/gdx/InputProcessor.html