Xtremme Pinball
Informe técnico

M. Besio P. García I. Scena

9 de febrero de 2009

1. Introducción

En el presente informe se describen los lineamientos generales tenidos en cuenta al llevar a cabo un desarrollo que permita la simulación y visualización en 3D de un juego en tiempo real. La implementación fue realizada en lenguaje *JAVA* y se utilizaron herramientas tales como un motor gráfico que se ocupe de los aspectos de la visualización y bibliotecas para el manejo de la física.

El juego desarrollado es un pinball. Una bola es lanzada sobre un panel con obstáculos (imanes, rampas, spinners, bumpers, puertas y objetos estáticos) que la van desviado de su camino y que van haciendo sumar puntaje al usuario. Mediante el uso de flippers, el jugador debe intentar evitar que la bola se pierda por entre medio de ellos o por las vías de escape laterales. El objetivo del juego es sumar la mayor cantidad de puntos posibles.

2. Herramientas utilizadas

A continuación se hace una breve descripción de las herramientas usadas para el desarrollo del juego.

2.1. JMonkey Engine 2 (JME2)

JME es un motor gráfico escrito en *JAVA* y distribuido bajo licencia *BSD*, que ofrece una API basada en un grafo para la descripción de la escena. Si bien la idea de su diseño es la de permitir el uso de cualquier sistema de rendering, en la actualidad el único soportado es *LWJGL*.

Además de ocuparse del rendering de la escena y de su representación en memoria mediante un grafo, *JME* permite la utilización de texturas, ilumunación, input y sonido entre otras cosas.

La versión de JME utilizada es la 2.0. La rasterización y shading se realizan mediante *OpenGL* (la biblioteca que usa es *LWJGL*, un wrapper de jni para dicho estándar), y para el audio se utiliza la librería *OpenAL*.

Para más información ver [1]

2.2. JME Physics 2

Para el manejo de la física se utilizó *JME Physics2*. Esta herramienta provee una interfaz entre *JME* y *ODE* (Open Dynamics Engine), facilitando la inclusión de fuerzas y todo tipo de dinámica entre los objetos que componen la escena.

Para su funcionamiento, *JME Physics* hace uso de la biblioteca de ODE mediante *odejava*, que vía Java Native Interfaces se comunica con la biblioteca nativa de ODE.

Para más información ver [2]

2.3. FenGUI

FengGUI es una API basada en *OpenGL* y escrita puramente en *JAVA* que permite programar interaces gráficas de usuario (GUI) en diversas aplicaciones. Provee todos los componentes típicos de una *GUI*, como botones, sliders, áreas de texto y tabs.

Durante el desarrollo del juego, se la utilizó para el diseño del menú. La razón de su elección fue principalmente que provee una *API* completa, fácil de usar, con buena integración con *JME* por estar escritos ambos en *JAVA* y permite la obtención de un producto de mayor calidad que Swing a un menor precio (esfuerzo).

Para más información ver [3]

3. Fases y diseño del proyecto

A continuación se hace una breve reseña sobre las fases en las que se puede dividir la implementación y algunos puntos del diseño utilizado para llevar adelante el desarrollo.

La implementación puede dividirse en diversas áreas, donde cada una de ellas cubre aspectos diferentes en lo que respecta al desarrollo del juego.

3.1. Conversor de X3D a JME

En la implementación, la descripción de la escena se efectúa por medio de 3 archivos *X3D*. El primero de ellos, *Room.x3d* describe la habitación en la que está contenida la máquina. El segundo, *Machine.x3d*, describe a la máquina en sí, y el tercero, cuyo nombre depende del theme que se quiera utilizar (cuya elección se hace desde el menú de juego nuevo), contiene información sobre la mesa de juego con todos los componentes que la conforman. Esta información corresponde a los modelos visuales de los elementos y a su *metadata* correspondiente. Para el modelado de los modelos visuales de

mesa, habitación, máquina y componentes se utilizó el programa *Blender*, aprovechando una opción de exportación a formato X3D.

Como la composición de la escena viene dada en formato X3D, y JME utiliza un grafo para llevar a cabo la descripción de la misma, es necesario efectuar una conversión entre ambos formatos. La clase encargada de realizar la traducción es *X3dToJme* y la misma se encarga de crear un DOM con el xml y recorrer cada uno de los nodos del árbol para convertirlos en nodos de *JME*. Todas las figuras complejas son interpretadas como mallas de triángulos para poder rasterizarlas con *OpenGL*. En el archivo *X3D* se encuentra la *metadata*, o información que no pertenece al formato *X3D*, y que describe el comportamiento de cada uno de los elementos de la mesa. Así, cada mesa, por ejemplo, puede tener la cantidad de flippers que desee, y cada flipper puede tener su propia forma y *metadata*. Con este diseño se busca aumentar la extensibilidad y flexibilidad del modelo. Todos los elementos de la mesa pueden tener *metadata*, por ejemplo, se puede configurar que un bumper sea fijo o que salte al contacto con la bola cambiando su propiedad de "jumper". Por simplicidad, los elementos que no tienen *metadata* son considerados obstáculos y son entes completamente estáticos. El archivo *X3D* además tiene la información del theme de la mesa a utilizar. Con este esquema, se permite el desacople de la lógica del juego, del modelo y del motor del mismo. Un mismo archivo *X3D* se puede utilizar con distintas lógicas cambiando una línea del mismo y una misma lógica puede tener varios modelos asociados.

En el X3D, la mesa se encuentra en un ángulo de 0 grados y la rotación se efectúa al momento de la carga. Para reducir el tiempo de carga, la misma se paraleliza porque el acceso, análisis y cálculo de normales de tres archivos con grandes cantidades de datos numéricos es una operación costosa en tiempo.

La iluminación de la escena se realizó con dos luces. La primera es una luz puntual ubicada cerca del techo de la habitación, por detrás de la posición natural de la cámara al jugar. Con esta luz puntual se ilumina toda la escena para que se puedan notar los detalles del cuarto. La segunda luz es un spot que apunta a la máquina para poder iluminar mejor sus componentes y está cerca de la intersección entre techo con la pared opuesta a la máquina.

3.2. Gamestates

El flujo normal desde que se inicia la aplicación lleva al juego a través de diferentes estados. Desde la presentación del menú hasta la simulación en sí, pasando por la pantalla de carga, se atraviesan lo que podedmos llamar *GameStates*. Cada uno de ellos se caracteriza por tener cargado cierto entorno de trabajo para garantizar el correcto funcionamiento del juego. *Xtremme Pinball* cuenta con tres *GameStates* principales: el estado de menu (*MenuGameState*), el estado de carga (*LoadingGameState*) y el estado de juego propiamente dicho (*PinballGameState*). Cada uno de ellos hereda indirectamente de una clase de JME denominada *GameState* que le provee la útil

funcionalidad de activarse y desactivarse mediante el uso del método *setActive(boolean)*. De esta manera, para cambiar de estado al juego bastará con una simple llamada de desactivación del estado actual y de activación del estado deseado, por ejemplo, para pasar de la simulación al menú principal con la presión de la tecla de menú.

3.3. Diseño de componentes

En el package *components* se encuentran las clases que implementan cada uno de los componentes que pueden ubicarse sobre la mesa de juego. Algunos de ellos son dinámicos: flippers, spinners, plunger, doors y bumpers saltarines (los que tienen forma de hongo) y otros estáticos: magnets, bumpers estáticos (los triangulares ubicados arriba de los flippers), sensores y obstáculos fijos.

Los componentes Bumper, Flipper y Magnet, implementan la interfaz ActivableComponent. Mediante ello, los mismos pueden ser activados y desactivados por la lógica del juego según sea necesario. Por ejemplo, en el theme *Just Race!*, los imanes se activan únicamente cuando el usuario logró cierta cantidad de rebotes contra bumpers sin perder vidas o cuando realizó cierto número de pasajes por la rampa.

Cada una de las clases que los implementan determinan el comportamiento de los mismos. Por ejemplo, los bumpers saltarines, cuando están activos, al sufrir una colisión con una bola, ejercen dos acciones: por un lado, aplican una fuerza sobre la bola cuya intensidad es proporcional a la velocidad que la misma traía al momento del impacto. Por otro, una fuerza vertical es aplicada sobre el bumper, a fin de hacerlo saltar. Los imanes, cuando están activos, ejercen una fuerza atractora sobre todas las bolas de la mesa que se encuentren dentro de su radio de acción. Esta fuerza es suficiente para desviar la trayectoria de las bolas, pero no para retenerlas. En las implementaciones realizadas, los imanes están ubicados debajo de la mesa (no son obstáculos visibles) en zonas elegidas con la finalidad de atraer las bolas hacia una vía de escape lateral.

Un aspecto importante en el diseño de los componentes es la elección del material con el que están hechos, ya que esto repercute directamente en la densidad que tendrán, en el coeficiente de rozamiento y el rebote contra otros materiales. Al material usado para los spinners se le seteó un valor de densidad que permitiera no hacerlos muy pesados, para evitar que los mismos giraran de forma indefinida cada vez que una bola los golpeara. Para los bumpers se creó un material con una densidad sufuciente como para que la bola no los corriera de lugar al impactarlos, y para los flippers se utilizó plástico, un material que viene incorporado en *JME*.

3.4. Lógica del juego

Para la implementación de la lógica y reglas del juego, se diseñó la clase *GameLogic* que implementa las reglas y comportamientos default de un juego de pinball. Cuando finaliza el juego desactiva los flippers. Al perder una bola, si es que ésta era la única que quedaba en la mesa, se decrementa en uno la cantidad de vidas restante. Detecta el abuso en el uso de tilt y efectúa la penalización correspondiente. También provee una serie de sonidos y mensajes de usuario default para ciertos eventos tales como perder una bola o mover un flipper.

En caso de querer agregar reglas nuevas, misiones o secuencias que brinden bonus por su completitud, sonidos acordes al theme de la mesa seleccionada o bonificaciones especiales (por ejemplo dar bolas extra o bonus en puntaje o vidas), basta con generar una clase que extienda a *GameLogic* y le agregue el comportamiento necesario. En la implementación presentada, por ejemplo, el theme *Just Race!*, a través de la clase *CarsThemeGameLogic*, agrega sonidos relacionados con autos para cada evento posible y hace override de los mensajes default para simular una carrera de automóviles.

La idea es que mediante el agregado de un archivo X3D (que describa la mesa a agregar), sonidos, texturas y un archivo *.class* que contenga el binario de una clase que extienda a *GameLogic*, se pueda cambiar totalmente el look, audio y reglas del juego. Es como si se pusiera un nuevo 'cartucho' de juego que reutilice toda la implementación desarrollada y agregue un comportamiento y aspecto visual que lo diferencie del resto de los otros 'cartuchos'.

3.5. Inputs de usuario

El manejo de las input de usuario se llevó a cabo a través de la API provista por JME. La forma de utilización consiste en instalar listeners para cada tecla a utilizar y asociarles la accion que deban ejercer. Por ejemplo, la tecla SPACE realiza tilt sobre la mesa, aplicando una fuerza sobre las bolas que están sobre la misma y generando un movimiento de la cámara que da la sensación de que la mesa se está moviendo.

3.6. Aplicación de texturtas

La aplicación de texturas se realiza durante el proceso de conversión de formato *X3D* a *JME* detallado en la subsección 3.1. Esto se hace tomando los nombres de archivo de las texturas especificados en el *X3D* y envolviendo la geometría del nodo que la requiera, para ya tenerlas incluidas en el grafo de salida. Los colores son especificados a través de materiales asignados a los elementos en el *X3D*.

En la figura 1 se observa un screenshot del juego cuando aún no se le habían aplicado texturas y colores. Mientras que en la figura 2 se expone el

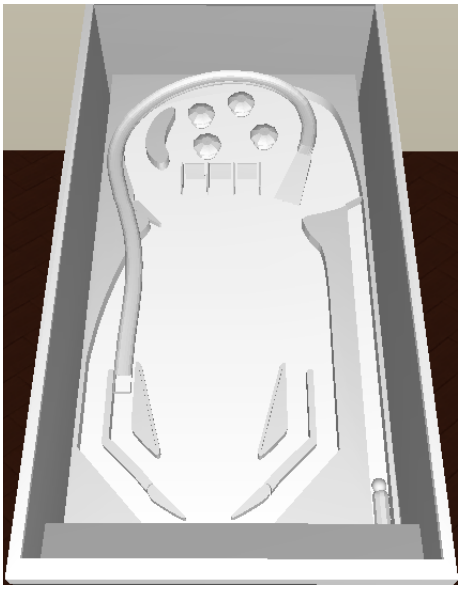

Figura 1: *Sin texturas aplicadas.*

resultado obtenido luego de la aplicación de texturas y colores a la mesa y sus componentes.

4. Problemas encontrados

Uno de los principales problemas que se tuvieron durante el desarrollo del proyecto fue la poca documentacion disponible sobre *Feng-gui*, *JME* y *JME Physics*, por lo que se tuvo que recurrir mucho a los foros disponibles en las páginas de ambos proyectos. Los aspectos que no se pudieron resolver por los medios anteriores, fueron solucionados experimentando por prueba y error.

La dll de ODE provista por JME contenía asserts de debug, lo cual causaba que durante el juego surgieran excepciones runtime con la información de debug. Para solucionar este problema se tuvo que recompilar la dll sin los asserts problemáticos, con todo lo que ello acarrea para poder realizar los binds con las funciones de java.

5. Posibles extensiones

Como posible extensión se puede mencionar la de implementar una feature que permita almacenar y ver los puntajes más altos. Si bien no requiere de gran trabajo, no se ha implementado por motivos de tiempo. Otra extensión posible es el diseño de más mesas de juego que se agreguen a las que el

Figura 2: *Con texturas aplicadas.*

usuario pueda elegir en el menú inicial.

Por último, debido a la gran complejidad del lenguaje de programación de shades de *OpenGL* (*glsl*) se decidió omitir la creación de shaders particulares para acortar los tiempos de desarrollo. Por lo tanto, para la iluminación se utilizó el shader default de *OpenGL*. Como una posible extensión para mejorar el aspecto visual del juego se podrían implementar los shaders correspondientes.

Referencias

[1] `http://www.jmonkeyengine.com/`

[2] `https://jmephysics.dev.java.net/`

[3] `https://fenggui.dev.java.net/`

Índice