UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Escuela de Ingeniería Informática

Motor de juegos en Ruby para Android

Proyecto de Final de Carrera

Garoe Dorta Pérez

Tutor: Agustín Trujillo Pino

Las Palmas de Gran Canaria, Junio de 2014

Índice general

1.Introducción....1
1.1.Estado del arte....1
1.1.1.Dispositivos móviles actuales....2
1.1.2.Motores de juegos recientes....4
1.1.2.1.Gráficos....4
1.1.2.2.Físicas....6
1.1.2.3.Completos....7
1.2.Objetivos....8
1.2.1.Objetivos directos....8
1.2.2.Objetivos indirectos....9
1.2.3.Investigación previa....10
2.Requisitos Hardware y Software....13
2.1.Desarrollo del motor....13
2.2.Uso del motor para el desarrollo de videojuegos....15
2.3.Software extra....17
2.4.Herramientas en detalle....17
2.4.1.Android SDK....18
2.4.2.Ruby....19
2.4.3.RVM....21
2.4.4.Ruboto....21
2.4.4.1.Interprete interactivo....22
2.4.4.2.Generador de aplicaciones 23
2.4.5.Bundler....24
2.4.6.Travis CI....25
2.5.Resumen....27
3.Análisis y Diseño....29
3.1.Análisis....29
3.1.1.Requisitos....29
3.1.1.1.Glosario de conceptos....29
3.1.1.2.Identificación de actores....29
3.1.1.3.Identificación de procesos....30
3.2.Diseño....31
3.2.1.Arquitectura del sistema....31
3.2.2.Diagrama de clases....32
4.Desarrollo....39
4.1.Gosu-Android....39
4.1.1.Versión 0.1→ IRB....39
4.1.2.Versión 0.2→ Aplicación Ruboto....44
4.1.3.Versión 0.3 → Optimización....48
4.1.4.Versión 0.4 → Java....49
4.1.5.Versión 0.5 → Profesionalidad....52
4.1.5.1.Juiciness....60
4.1.6.Versión en desarrollo....61
4.2.Chipmunk Android....61
4.3.Interacción con la comunidad....63
4.3.1.Código de la comunidad....63
4.3.2.Código para la comunidad....63
5.Conclusiones y Trabajo futuro....67

5.1.Conclusiones.....67
5.2.Trabajo futuro.....68
6.Bibliografía.....69
7.Anexo.....70
7.1.Wiki.....70

1. Introducción

Los juegos de ordenador son una disciplina que empezó en 1947 cuando se presentó la patente en Estados Unidos para un “Dispositivo de entretenimiento de rayos catódicos”[0]. Al principio se consideraron una perdida de tiempo y esfuerzo, y continuaron siendo vistos como una disciplina banal hasta la década de los 80s, cuando se consolidaron como una industria sólida del entretenimiento. Desde entonces, su popularidad no ha hecho más que crecer año tras año, generando a día de hoy beneficios multimillonarios.

Aunque en un principio se programaba cada videojuego de manera artesanal, pronto se observó que muchas partes del desarrollo se repetían en la creación de éstos. Por esta razón surgieron los motores de videojuegos para agilizar el desarrollo de los mismos. El uso de motores fomenta además la reutilización de código, ya que proporcionan a los desarrolladores características muy útiles tales como:

- **Manejo de vídeo:** proporcionan una API que abstrae al programador del tipo de librería que se está usando para hacer las operaciones de dibujo.
- **Manejo de sonido:** con esta API se permite la reproducción de distintos formatos de sonido, así como controlar parámetros del tipo, volumen, panning, etc.
- **Motor de físicas:** permite simular fácilmente en tiempo real sistemas de cuerpos rígidos, fluidos, etc.
- **Soporte para animaciones:** con esta característica se facilita la reproducción de animaciones en los modelos que se usen.
- **Soporte multiplaforma:** permite desarrollar el mismo código para distintas plataformas, tanto software (Windows, Mac OS X, Linux) como hardware (PC, Smartphones, Consolas).

En un primer momento la idea del proyecto fue desarrollar un videojuego usando el lenguaje de programación Ruby. Sin embargo al poco tiempo la idea derivó en desarrollar un motor propio de videojuegos para Smartphones, ya que durante el desarrollo del mismo se observaron ciertas carencias en las tecnologías usadas para la creación del videojuego.

1.1. Estado del arte

Una vez establecida la idea principal del proyecto, procedemos a explicar cual es estado del arte tanto de Smartphones como de motores de videojuegos. Antes de comenzar, es importante recalcar que el documento no pretende ser una guía exhaustiva del estado del arte actual, sino mostrar cuales son los los actores mas importantes, además de los minoritarios pero aun así relevantes para este proyecto.

1.1.1. Dispositivos móviles actuales

Hoy en día tenemos cuatro plataformas principales para las cuales se pueden desarrollar aplicaciones móviles: BlackBerry, WindowsPhone, iPhone y Android. Que pasaremos a describir en detalle a continuación.

BlackBerry

La compañía Blackberry [1] lleva desarrollando estos smartphones desde 1999, todos usan el sistema operativo **BlackBerry OS**. Los dispositivos siempre se han desarrollado para el consumidor empresarial, es decir son dispositivos de gama alta y con precios poco asequibles. En la actualidad solo tiene un 3% del mercado de smartphones y se baraja una posible compra de la empresa. Las aplicaciones se encuentran en la **BlackBerry World**, para publicar se han de comprar claves para firmar la aplicación por uno 20$.

Windows Phone

Windows Phone [2] es el desarrollo de Windows (software) junto con Nokia (Hardware) para competir en el mercado de smartphones. En concreto está diseñado para el público general al contrario que la BlackBerry. A pesar de ofrecer buenos dispositivos a precios razonables y realizar grandes gastos en marketing solo tiene un 3% de cuota de mercado. La razón más argumentada es su tardanza, ya que la compañía que llego al mercado en 2010. Para desarrollar aplicaciones en la **Windows Phones Store** se ha de usar el Visual Studio, cuya versión completa cuesta 99$ al año.

iPhone

Apple es el creador de iPhone [3], estos dispositivos llevan en venta desde 2007. Usan el sistema operativo **iOS.** Tienen una base de clientes solida y fiel, a pesar de unos precios bastante altos,

además las nuevas versiones tiene una tasa de adopción muy alta. Sin embargo para poder publicar aplicaciones en la **AppStore** se ha de comprar una licencia de desarrollador que tiene un coste de 99$ anuales. Sin embargo la ULPG tiene un acuerdo por el cual se ofrecen licencias gratuitas para estudiantes.

Android

Google es el principal desarrollador del sistema operativo software libre **Android** [4], está basado en **Linux** y es publicado con la licencia Apache. Por otra parte tenemos a las empresas de hardware que fabrican los smartphones: Acer, Amazon, Archos, Asus, HTC, Samsung, Sony, Toshiba y ViewSonic. El modelo de negocio se basa en la venta de dispositivos para las compañías de hardware y la venta de aplicaciones para Google. Debido a la variedad de empresas que fabrican los dispositivos se ofrecen dispositivos para todos los públicos. Sin embargo el principal problema es la fragmentación del mercado: distintas versiones de Android, distintos tamaños de pantalla, etc.

Por otra parte se trata del sistema operativo más popular, con una cuota de mercado del 79.3%. Además de ser permitir desarrollar y publicar aplicaciones en **Google Play** sin costo para el programador.

Debido a que Android es el sistema operativo para dispositivos inteligentes más popular y además desarrollar aplicaciones en él, es gratuito. Es la opción elegida como plataforma para implementar el proyecto.

1.1.2. Motores de juegos recientes

Podemos distinguir entre motores que proporcionan facilidades para manejar vídeo, sonido y eventos de entrada. Motores de físicas que se integran con los primeros y motores completos que ya vienen con todo incluido. Debido a que uno de los objetivos del proyecto era escribir el motor en Ruby la mayoría de la investigación del estado del arte actual se centró en motores que permitieran escribir juegos en Ruby.

1.1.2.1. Gráficos

Ray

Motor de juegos open source [5] desarrollado principalmente en C pero pensado para ser usado con Ruby. Es multiplaforma, con soporte para Windows, Mac OS X y Linux, además permite tanto dibujo 2D como 3D usando OpenGL en ambos casos. No es muy popular puesto que solo tiene un desarrollador, no tiene actualizaciones desde hace más de dos años, su canal de IRC se encuentra vacío, y parece por tanto un proyecto abandonado.

G3DRuby

Se trata de un port [6] para Ruby de G3D, un motor open source para aplicaciones gráficas en 3D. Proporciona una serie de rutinas y estructuras comunes para cualquier aplicación gráfica. Permite usar de manera más sencilla OpenGL y sockets. Tiene soporte para Windows, OS X y Linux. Aunque permite crear juegos sencillos está más enfocada a aplicaciones gráficas de manera genérica.

AndEngine

Se trata de un motor [7] open source para juegos desarrollado en exclusiva para Android, está

escrito y destinado a ser usado con Java. Solo permite gráficos en 2D y usa OpenGL. Aunque se puede desarrollar desde cualquier PC, con Windows, Mac OS X o Linux. Se está volviendo cada vez más popular, tiene un desarrollo muy activo, aunque con poca documentación. Por otra parte, recientemente se publicó un libro tutorial sobre el.

Gosu

Gosu [8] es un motor open source para desarrollo de juegos en Ruby y C++. Es multiplaforma con versiones para Mac OS X, Windows y Linux. Además las versión en C++ también incluye soporte para iPhone y iPad.

Está desarrollado principalmente en C++ y es especialmente popular en competiciones con límite de tiempo, debido a su facilidad de uso y al hecho de ser muy eficiente. Cuenta con una comunidad pequeña pero activa, que se comunica principalmente a través de foros. Para vídeo solo tiene soporte para 2D usando OpenGL y para audio usa OpenAL en Mac OS X, SDL_mixer en Linux y Audiere en Windows. Soporta dispositivos de entrada varios, tales como mandos de juegos. Además de soporte para muchos formatos de audio y imágenes. Para conseguir la integración entre C++ y Ruby usa el software SWIG.

SWIG proporciona herramientas para permitir llamar a funciones de C/C++ desde otros lenguajes, pasando tipos de datos complejos, evitando que la memoria sea liberada, etc. El programador ha de escribir una lista que contiene todas las funciones que será visibles por el interprete. SWIG automáticamente generará el código para funciones simples, pero las que sean complejas el programador tendrá que ayudar al programa especificando datos adicionales.

Debido a que este es el motor más completo para Ruby y además es software libre será el motor usado para el proyecto.

Chingu

Este motor [9] open source es una extensión de Gosu, proporciona abstracciones a más alto nivel y está escrito enteramente en Ruby. Proporciona soporte para las mismas plataformas que Gosu ya que usa Ruby. Además entre otras características extiende Gosu mediante la inclusión de: detección

de colisiones, animaciones, desplazamiento por paralaje, partículas y manejo de estados para niveles del videojuego.

Löve

Motor [10] software libre multiplataforma, con soporte para Windows, Mac OS X y Linux. Permite dibujar gráficos en 2D, únicamente se pueden programar juegos en Lua. Tiene una comunidad pequeña pero muy activa. Los desarrolladores también tienen son muy activos en cuanto a desarrollo del motor, además es muy sencillo de usar.

1.1.2.2. Físicas

Chipmunk

Chipmunk [11] es un motor en tiempo real de físicas de cuerpos sólidos, ha sido diseñado para ser eficiente, rápido y fácil de usar. Además tiene bindings para Objective-C lo cual lo hace muy popular para usarlo con iPhone. Además de tener también bindings para Phyton, Ruby, Haskell y algunos más de manera no oficial. Tiene un sistema de colisiones flexible con capas, grupos de exclusión y callbacks. Ofrece una versión gratuita y una de pago. Sin embargo la gratuita no permite su uso para Android. Recientemente han sacado un versión de pago para usar junto con Unity para juegos 2D.

Teniendo en cuenta que Gosu no tiene motor de físicas y Chipmunk se puede integrar con Gosu este es el motor elegido para la parte de físicas.

Havok Physics

Motor de físicas [12] desarrollado principalmente para videojuegos. Permite colisiones en

tiempo real y dinámicas de cuerpos rígidos en tres dimensiones. Así como constrainst dinámicos, tal como físicas de ragdoll. Es uno de los motores de físicas más usados en juegos comerciales de mucho presupuesto tal como Half-Life 2, Guild Wars 2, etc. Es totalmente multiplataforma, con soporte para PC, dispositivos móviles y consolas. Es de código cerrado y pone a disposición de los desarrolladores una versión gratuita, sin embargo esta es solo es compatible con Windows.

1.1.2.3. Completos

Unity

Uno de los motores [13] más populares, es de código cerrado y está escrito en C# y C++. Permite desarrollar videojuegos para PC, navegador, dispositivos móviles y consolas. Sin embargo solo es posible programar desde Windows y Mac OS X. Tiene dos licencias, la gratuita orientada para desarrolladores independientes, que permite publicar para PC. Y la licencia de pago, que incluye más características, como el renderizado a textura o la iluminación global. Posteriormente se ha de comprar la licencia para cada plataforma en la que se quiera publicar por separado. Tiene una comunidad enorme y muy activa.

Irrlicht

Motor [14] de juegos open source escrito en C++. Es multiplataforma, tiene soporte para Windows, Mac OS X y Linux. Irrlicht es popular debido a su pequeño tamaño y compatibilidad con hardware tanto nuevo como viejo, una curva de aprendizaje no muy alta y una comunidad muy activa y amigable.

Existen bindings para varios lenguajes, aunque no de manera oficial, como .NET, Java, Perl, Ruby, Python, Lua, Delphi y algunos más. El que más nos interesa, el de Ruby, se encuentra en un estado experimental. Cuando se probó, no funcionó y además parecía estar abandonado pues no había actividad reciente en torno a el.

Panda3D

Motor de videojuegos [15] gratuito desarrollado por Disney que incluye gráficos, audio, I/O, detección de colisiones y otras características relevantes para la creación de juegos 3D. Es de código abierto y software libre desde 2008. Está preparado para programar bajo Python, pensado para obtener una programación rápida y gestión de memoria avanzada, aunque también se puede utilizar con C++, dado que está programado en este lenguaje. Se ha utilizado para el desarrollo de juegos comerciales y algunos proyectos open source. Además, algunas universidades lo enseñan por tener una curva de aprendizaje bastante corta.

La comunidad a su alrededor es pequeña pero activa. Por lo general, las preguntas realizadas en su foro se responden con rapidez. Desafortunadamente por el momento no da soporte a plataformas móviles, es compatible con Microsoft Windows, Linux, OS X y FreeBSD.

1.2. Objetivos

Entre los objetivos que cubre el proyecto vamos a hacer dos distinciones. Objetivos directos, que son los que se plantearon directamente con el proyecto. Y objetivos indirectos, que derivan de los directos y de las tecnologías elegidas para el desarrollo del mismo.

1.2.1. Objetivos directos

El objetivo principal del proyecto es crear un **motor** de juegos en **Ruby** para dispositivos **Android**. Dicho motor a pesar de estar escrito en Ruby tiene que estar integrado en Java puesto que todos los programas para Android se escriben en dicho lenguaje. Para lograr que el proyecto tenga un impacto más allá de como mero ejercicio académico se optó por implementar la **API** de los motores **Gosu** y **Chipmunk**. De esta manera extendemos ambos motores para que soporten también plataformas móviles Android. A partir de ahora denominaremos al código que permite a Gosu funcionar en Android como **Gosu-Android** y a su contrapartida de Chipmunk como **Chipmunk-Android**.

Por otra parte, se incluye entre los objetivos que el código se ha de desarrollar usando Linux, en concreto **Ubuntu**. Además ha de ser **software libre** y usar herramientas que también lo sean en la medida de lo posible.

Para dar una visión más clara de lo que aporta el motor, se muestra a continuación una tabla

comparativa entre los motores mencionados previamente y el nuestro:

Motor	Linux	Windows	Mac	Android	Lenguaje programación	Software libre	Tipo
Ray	✓	✓	✓	✗	Ruby	✓	Gráficos
G3DRuby	✓	✓	✓	✗	Ruby	✓	Gráficos
AndEngine	✓	✓	✓	✓	Java	✓	Gráficos
Gosu	✓	✓	✓	✗	C++/Ruby	✓	Gráficos
Chingu	✓	✓	✓	✗	Ruby	✓	Gráficos
Löve	✓	✓	✓	✗	Lua	✓	Gráficos
Chipmunk	✓	✓	✓	✗	C/Ruby	✓	Físicas
Havok Physics	✗	✓	✗	✓	C++	✗	Físicas
Unity	✗	✓	✓	✓	C++	✗	Completo
Irrlicht	✓	✓	✓	✗	C++	✓	Completo
Panda3D	✓	✓	✓	✗	C++	✓	Completo
Gosu-Android	✓	✓	✓	✓	Ruby	✓	Gráficos
Chipmunk-Android	✓	✓	✓	✓	Ruby	✓	Físicas

Tal como se observa en la tabla, Gosu-Android es el único motor gráfico que se puede usar desde **Linux**, **Windows** y **Mac OS X**, para **Android** programando con **Ruby**, mientras en Chipmunk-Android es la única opción para físicas.

1.2.2. Objetivos indirectos

Todo proyecto de cierta envergadura necesita un gestor de versiones para poder llevar a cabo un desarrollo con ciertas garantías de éxito, en nuestro caso optamos por usar **git**, del cual tenía un

conocimiento bastante limitado, por lo que se buscó ampliar dicho conocimiento. Además el repositorio git fue alojado en **github**, por lo que también hay que familiarizarse con dicha plataforma. Puesto que aporta herramientas muy útiles para el trabajo comunitario, tal como el poder reportar issues, o crear pull requests.

- Para poder desarrollar aplicaciones para Android en necesario aprender a usar el **Android-SDK**. Este proporciona herramientas tales como un **emulador** para una variedad de dispositivos, manejo de archivos o consultar logs de ejecución.

- Aprender a manejar la plataforma **Ruboto**, que permite crear programar usando Ruby para dispositivos Android.

- Aprender a manejar la plataforma **Bundler**, puesto que es la herramienta que usa Ruboto para incluir librerías en sus proyectos.

- Aprender a manejar **Travis CI**, pues esta plataforma de integración continua es usada tanto por Ruboto como por Bundler.

- Aprender a manejar y configurar **Rake**, ya que esta herramienta es usada tanto en Ruboto como en Bundler para automatizar tareas de compilación, ejecución además de otras tareas repetitivas.

- Por último, también se ha de aprender como funcionan las comunidades de software libre, es decir, usar **IRC**, suscribirse a **listas de correos**, participar en **foros** de discusión, resolver **issues** y **contribuciones** de desarrolladores, etc.

- Crear una wiki con información acerca del motor.

1.2.3. Investigación previa

Una vez que se han establecido los objetivos y antes de empezar a establecer requisitos, se procedió a recabar información acerca de la plausibilidad del proyecto. Para ello se leyeron varios artículos acerca de la idoneidad de Ruby como lenguaje para el desarrollo de videojuegos. Su eficiencia y posibles problemas a la hora de desarrollar un motor usando Ruby en dispositivos con capacidad de computo limitada [16].

El principal problema que plantean es la eficiencia, al ser Ruby un lenguaje interpretado es difícil alcanzar velocidades de ejecución rápidas.

Por otra parte el colector de basura también es un problema, ya que interrumpirá el videojuego mientras realice su tarea.

Teniendo en cuenta los comentarios de este foro hay tres opciones, que en orden de complejidad son:

1. Escribir todo el código en Ruby, comprobar si es lo suficientemente eficiente.

2. Escribir todo/parte del código en Java y luego wrapear dicho código para que sea llamado desde Ruby.

3. Escribir parte del código en C/C++ usando el Android NDK (Native Development Toolkit) o JNI (Java Native Interface), luego wrapear esta parte para Java y de nuevo wrapear para Ruby.

Debido a que no se existían otras motores en Ruby para Android, el rendimiento real era desconocido, por tanto la opción más sensata fue ir avanzando en complejidad a medida que fuera siendo necesario.

2. Requisitos Hardware y Software

A continuación se procede a describir las aplicaciones software y los recursos hardware necesarios para poder llevar a cabo el desarrollo del proyecto. En primer lugar podemos diferenciar dos tipos de requisitos distintos: de desarrollo del motor y de uso del motor para la creación de videojuegos.

2.1. Desarrollo del motor

Aquí se engloban los requisitos software y hardware que fueron necesarios únicamente para programar el código del motor.

PC

En primer lugar es necesario disponer de un PC. Los sistemas operativos en los que se probó con éxito el motor son **Ubuntu**, **Windows** y **Mac OS X**. Sin embargo el desarrollo fue llevado a cabo en su mayoría bajo Ubuntu, puntualmente en Windows y su funcionamiento en Mac OS X es conocido debido a su uso por parte de algunos usuarios de la comunidad.

Android-SDK

Este software proporciona todas las herramientas necesarias para poder desarrollar aplicaciones para Android. Es importante destacar que para su funcionamiento es necesario tener instalado un entorno de desarrollo Java, se usaron con éxito tanto el **Oracle JDK** como su contrapartida de software libre **OpenJDK**.

Interprete Ruby

Como el código a desarrollar será escrito en Ruby es necesario tener un interprete instalado, los interpretes que se probaron con éxito fueron **MRI** y **JRuby**.

Ruboto, Rake y Bundler

Es necesario instalar estas gemas para el desarrollo del motor, **Ruboto** es la implementación de Java en Android, **Rake** nos servirá como utilidad de compilación y **Bundler** para instalar dependencias.

Git y Github

Para facilitar el desarrollo la herramienta elegida para la gestión de versiones fue **Git** [17]. Además para darle la mayor difusión posible, permitir forks fácilmente y desarrollar en distintos ordenadores con facilidad, se decidió subir el código al repositorio público de **Github** [18].

Eclipse

IDE [19] creado originalmente para desarrollar aplicaciones en Java. Hasta hace poco era el único entorno destinado a desarrollar aplicaciones para Android mediante el plugin **ADT**. Además permite compilar fácilmente archivos .jar a partir de código Java para su uso.

Aptana

IDE [20] para Ruby, proporciona un resaltado de sintaxis muy bueno para este lenguaje, así como detección de errores en tiempo real. Por otra parte, es bastante útil su integración con Git (importar proyectos, moverse entre ramas, etc). Para instalarlo, se debe descargar el paquete .deb de su página web.

Geany

Editor [21] muy sencillo usado para cambios pequeños de código, tiene resaltado de sintaxis tanto para Java como para Ruby. Además de permitir compilar y ejecutar proyectos sencillos sin necesidad de realizar una configuración previa.

Gedit

Editor [22] sumamente simplista usado para cambios pequeños de código o consultar el contenido de algún archivo rápidamente, tiene resaltado de sintaxis tanto de Java como de Ruby.

***** Todos los entornos de desarrollo y editores aquí mencionados fueron usados en el proyecto, sin embargo se puede usar cualquier otro, puesto que se incluye un comando Rake para compilar el código escrito en Java. Mientras que los archivos en Ruby no requieren ningún tratamiento especial.

2.2. Uso del motor para el desarrollo de videojuegos

Aquí se incluyen los requisitos necesarios para usar el motor.

Salvo Git y Github los requisitos son los mismos que para el desarrollo del motor. Pudiéndose usar el IDE o editor de texto que se considere más conveniente, para modificar los archivos Ruby con el código del juego que se esté creando.

Existen varias opciones para probar el código del videojuego escrito usando el motor:

Emulador del Android-SDK

El emulador de Android proporcionado por el Android-SDK, permite simular dispositivos hechos a medida, es decir elegir cualquier combinación de memoria, procesador, tamaño de pantalla, versión de Android, etc. Esta opción no es recomendable en absoluto debido a la lentitud tanto de arranque como de ejecución del emulador.

BlueStacks

Este emulador [23] de código cerrado está disponible para Windows y Mac OS X. No ha sido diseñado para el desarrollo de aplicaciones, sino para su uso en aplicaciones finales que ya existen en Google Play. Sin embargo, se le pueden instalar fácilmente las aplicaciones siempre que dispongamos del **.apk** o usando **Ruboto**, puesto que también funciona con **adb**. Además es muy rápido y fácil de usar, sin embargo al no disponer de cliente para sistemas Linux no fue usado con asiduidad.

Dispositivo físico

Cualquier dispositivo con Android es válido, tanto **tablets** como **smartphones**. Esta es la opción recomendada siempre que se disponga de uno, puesto que es el objetivo final para el cual se crea el programa. Durante el desarrollo del proyecto se dispuso de un tablet con Android 3.0 en el que se probaron las aplicaciones. Es importante resaltar que el dispositivo no tiene porque estar rooteado para poder desarrollar y probar código en el, aunque puede facilitar algunas tareas.

2.3. Software extra

En esta sección se presenta software usado en el proyecto pero no necesario para su desarrollo.

Libreoffice

LibreOffice
The Document Foundation

Esta suite ofimática fue usada para redactar esta memoria.

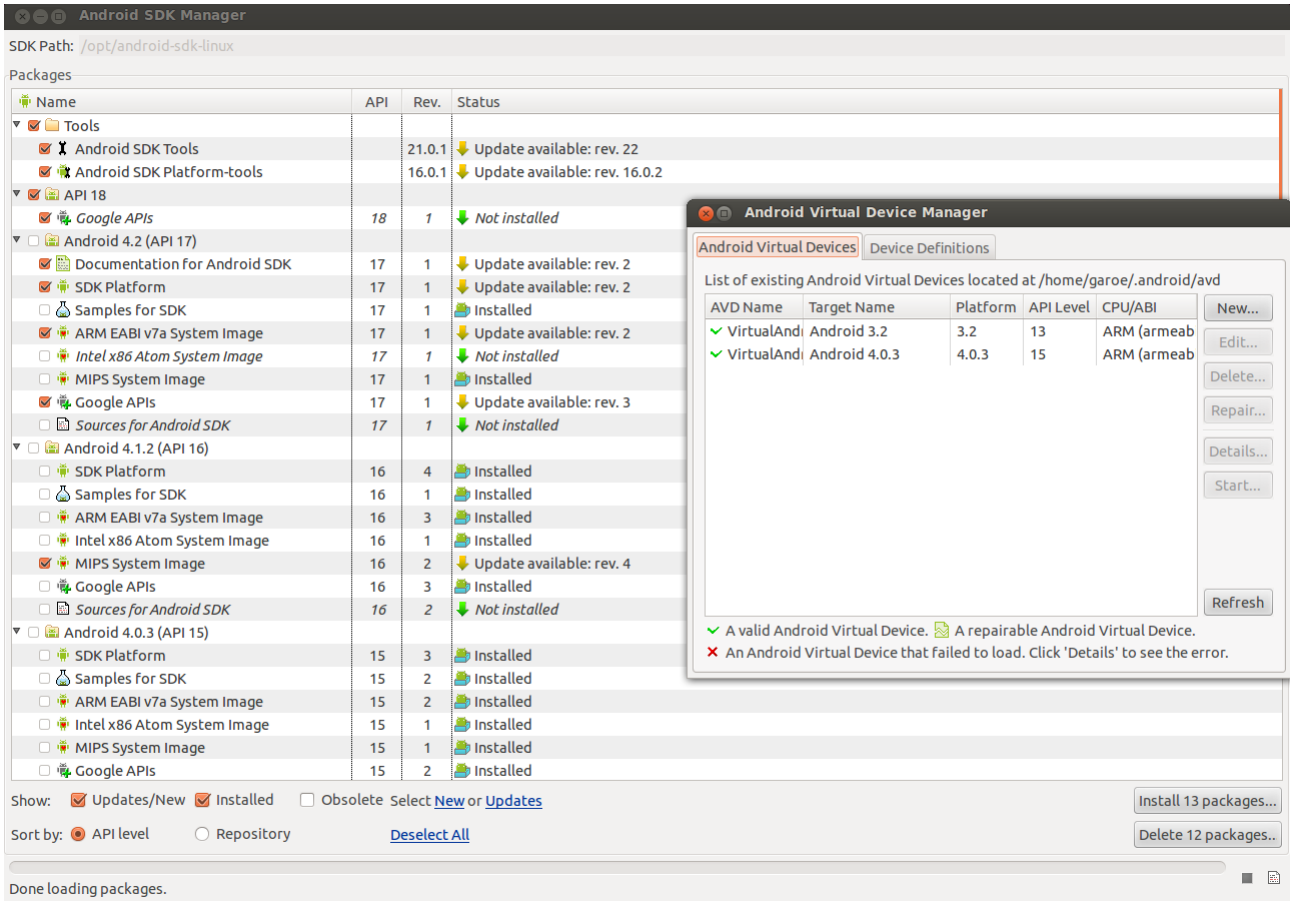
AndroidScreenCapture

Este software permite capturar imágenes a traves de USB de un dispositivo Android. Fue usado para sacar las capturas de pantalla que se muestran a lo largo de la memoria.

2.4. Herramientas en detalle

Durante la introducción y en los requisitos se nombraron varias herramientas software que se explicaron brevemente. Sin embargo antes de continuar con el resto de la memoria conviene explicar algunas de ellas en mayor profundidad, puesto que en caso de no hacerlo no se podrá entender plenamente los apartados posteriores.

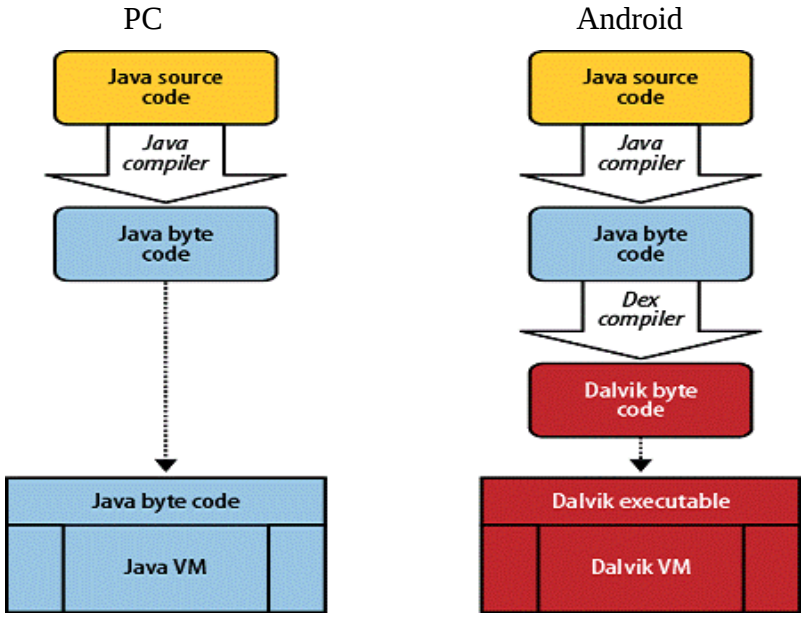
2.4.1. Android SDK



Para crear aplicaciones para dispositivos Android, Google pone a disposición de los desarrolladores el Android-SDK [24] de manera gratuita. Una vez descargado e instalado permite instalar los componentes que necesitemos por separado. Como mínimo es necesario instalar todas las herramientas de la carpeta **Tools** y el **SDK Platform** de al menos una versión de Android.

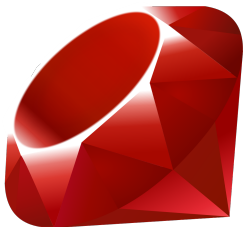
Además de proporcionar los medios para poder desarrollar aplicaciones en un dispositivo real, también incluye un emulador. Desde el emulador podemos crear distintos dispositivos virtuales desde los cuales podemos desarrollar y probar aplicaciones sin la necesidad de tener un dispositivo real.

Para programar en Android se han de escribir los programas en Java, sin embargo este no usa el JVM (Java Virtual Machine) si no que tiene su propio interprete tal como se muestra en la imagen a continuación:



Esta fue una decisión de Google, pues al usar su propia máquina virtual esta fue diseñada desde un inicio para hardware de dispositivos móviles donde existen limitaciones en cuanto a batería, rendimiento, memoria, etc.

2.4.2. Ruby

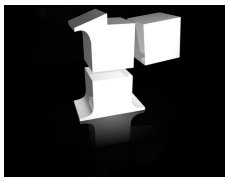


Dado que el lenguaje de implementación elegido fue Ruby [25], y este lenguaje es relativamente moderno es importante destacar algunos aspectos relevantes de su uso y funcionamiento.

En primer lugar Ruby es un lenguaje interpretado, dinámico y de tipado fuerte. Es necesario disponer de un interprete para poder ejecutar código escrito en Ruby, por suerte contamos con varios a nuestra disposición:

- **MRI:** interprete estándar, está escrito en C, actualmente hay tres versiones en uso. La versión 1.8.2 es la más antigua y estable, sin embargo es la más lenta. Posteriormente tenemos la versión 1.9.2, es sensiblemente más rápida que la 1.8.2, además de incorporar mejoras en otros aspectos. No obstante su uso todavía no está completamente generalizado. Por último tenemos la 2.0.0 es la versión más reciente.

JRuby: segundo interprete [26] más popular, está escrito en Java y aunque tiene un tiempo de arranque un poco más lento que MRI una vez en marcha es igual o más rápido que su contrapartida en C++ puesto que sus autores han puesto especial énfasis en la eficiencia. Proporcionando además una buena integración con Java, puesto que permite importar clases de Java a partir de, tanto archivos fuente .java como de archivos compilados .class o un conjunto de ellos en un .jar.

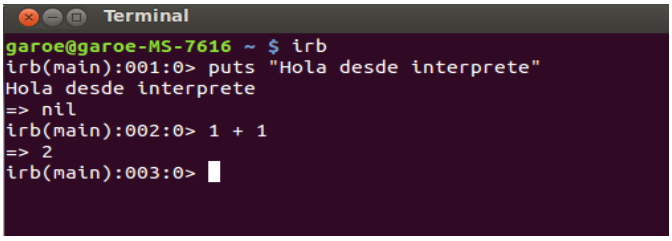


Rubinius: interprete [27] escrito en C++, sigue un diseño completamente distinto a MRI. Además solo tiene la base escrita en C++, e intentan escribir la mayor parte del código en Ruby. En la mayoría de los benchmarks es más eficiente que MRI, sin embargo no es 100% compatible con MRI.

- **Otros:** Existe una gran variedad de otros interpretes como IronRuby, GoRuby, MagLev, etc. Sin embargo debido a que no son demasiado populares no se explican en detalle.

Una vez instalados podemos ejecutar el contenido de un fichero con un programa en Ruby, normalmente terminado en **.rb** usando los siguientes comandos: ruby programa.rb, jruby programa.rb o rbx programa.rb, respectivamente para MRI, JRuby o Rubinius.

Los interpretes además permiten usar una versión interactiva, en la que se puede ir escribiendo comandos para probar código en tiempo real. Estos son **irb** y **jirb** respectivamente para MRI y Jruby. A continuación se muestra una captura en la que se ve el interprete interactivo irb y se escriben algunas instrucciones:



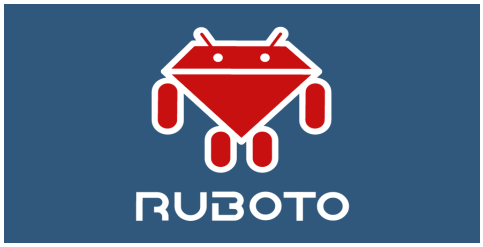
Otro concepto importante son las librerías en Ruby, estas son llamadas **gemas**. El repositorio oficial y más popular de gemas para Ruby es **RubyGems** [28], cualquiera puede subir su propia gema a este repositorio y esta será automáticamente accesible para el resto del mundo. La única restricción es que el nombre de la gema no debe estar en uso.

2.4.3. RVM

Como ya se explicó en el apartado anterior existen varios interpretes de Ruby y cada uno con varias versiones en activo. Sin embargo solo es posible tener una instalación funcionando en una máquina dada. Para resolver este problema se puede usar RVM (Ruby Version Manager) [29], con este programa podemos tener todas los interpretes instalados que queramos y en distintas versiones. Además cada gema que se instale, se instalará en un sandbox asociado al interprete y versión de este que esté activa al momento de instalar la gema. Pudiendo en cualquier momento instalar, borrar o cambiar entre todos los interpretes y versiones de ellos, que se hayan instalado.

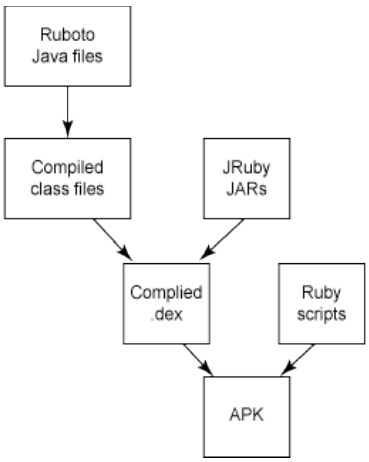
Por tanto lo más recomendable es instalar Ruby y las gemas que se necesitasen usando RVM. De esta manera nos ahorraremos muchos problemas, puesto que es común tener gemas que no son compatibles con ciertas versiones de los interpretes.

2.4.4. Ruboto

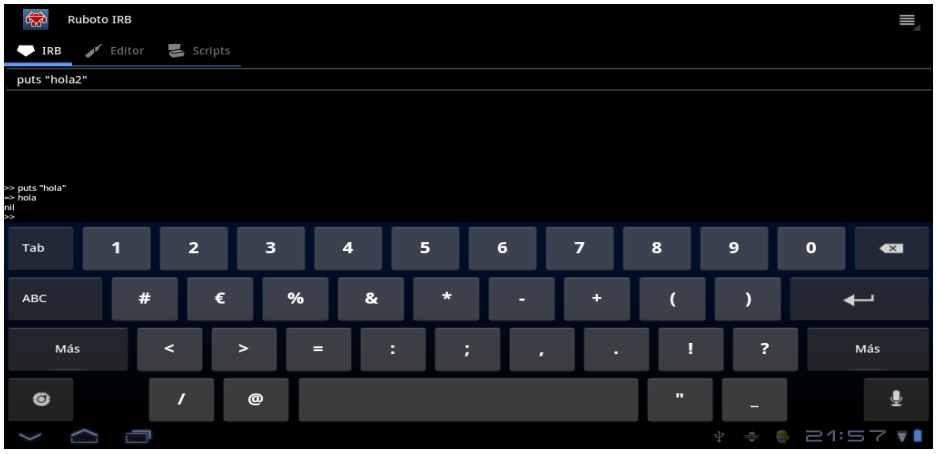


Ruboto [30] es una modificación de JRuby que ha sido adaptada para funcionar bajo el sistema operativo Android. De todos los cambios cabe resaltar que ha sido optimizado para ser más eficiente y tener un tiempo de arranque más rápido, además de proporcionar automáticamente una clase Activity que es el entryPoint para poder ejecutar código Ruby en Android.

Su funcionamiento a grosso modo es el siguiente:



2.4.4.1. Interprete interactivo



El equipo de Ruboto no solo proporciona el código ya mencionando, si no que además ha creado la aplicación Android **Ruboto-IRB**, que consiste en un equivalente al irb para PC. Si ejecutamos la aplicación podemos ir escribiendo código en Ruby y veremos el resultado al instante. No solo eso, si no que además incorpora un editor de texto y varios scripts de ejemplo. Podemos ver, modificar y ejecutar el código de dichos scripts. Además permite crear nuevos scripts que serán guardados como ficheros en el dispositivo. Cabe destacar el script **demo-irb-server.rb** ya que al ejecutarlo crea un servidor al cual nos podemos conectar en LAN introduciendo la dirección IP del dispositivo en cualquier navegador. Una vez hecho esto, veremos una página web desde la cual podemos escribir código que podemos enviar a ejecutar al dispositivo. Es más, tenemos acceso a todos los scripts de la aplicación y después de la ejecución se nos mostrará los mensajes de salida que haya generado el código que hemos enviando a ejecutar.

2.4.4.2. Generador de aplicaciones

Ruboto también proporciona varios comandos muy útiles para desarrollar aplicaciones:

- **ruboto gen app:** con este comando el programa crea una carpeta con un proyecto predefinido ruboto. Esta es la manera más fácil para empezar a programar, puesto que lo único que tenemos que hacer es modificar el fichero fuente en ruby que crea por defecto el comando.

- **ruboto setup:** con este comando se instalan todas las dependencias necesarias para poder ejecutar rubuto, tal como el android-sdk, etc. De momento es experimental y solo funciona en OS X.

Una vez dentro de la carpeta del proyecto que se ha creado con ruboto se disponen de los siguientes comandos para su uso:

- **rake install:** instala la aplicación en el dispositivo android que tengamos conectado a nuestro sistema.

- **rake reinstall:** desinstala, recompila y vuelve a instalar la aplicación.

- **rake uninstall:** desinstala la aplicación del dispositivo.

- **rake start:** inicia la aplicación en el dispositivo.

- **rake stop:** para la aplicación en el dispositivo.

- **rake restart:** reinicia la aplicación en el dispositivo.

- **rake release:** crea un archivo APK listo para ser usado para ser instalado o publicado en la Google Play .

- **rake update_scripts:** actualiza únicamente el código Ruby de una aplicación ya instalada. Esto es muy útil debido a que es mucho más rápido que compilarlo todo de nuevo.

- **rake test:** ejecuta las pruebas que hayamos definido en el directorio test de la aplicación.

- **rake bundle:** crea un archivo Bundle.jar que contendrá todas las gemas que se hayan especificado en el archivo Gemfile.apk y que serán accesibles desde el código del proyecto. Para ello usa la gema **Bundler**.

2.4.5. Bundler

Bundler [31] es una gema creada para facilitar el manejo de dependencias y versiones de las distintas gemas que use un proyecto dado. Para ello permite especificar en un archivo de texto plano llamado Gemfile, cuales son las gemas que queremos usar en nuestro proyecto, así como una serie de opciones entre las que podemos resaltar:

- **Versión:** se puede especificar tanto una versión específica, como una mínima o un rango de versiones.
- **Plataforma:** de las distintas plataformas en las que puede estar una gema, indicamos cual queremos instalar.
- **Grupos:** con esta opción se pueden indicar dependencias por grupos, por ejemplo se puede tener un grupo de desarrollo y otro de release y cada uno tiene unas dependencias distintas.

Bundler resuelve automáticamente todas las dependencias entre las distintas gemas que se le especifiquen. Y además realiza esta tarea teniendo en cuenta retrocompatibilidad, es decir, no solo instala las gemas necesarias para que funcionen las gemas que le hemos indicado, sino que además comprueba que todas sean compatibles entre si, teniendo en cuenta las versiones que le hemos indicado. Una vez están todas la dependencias hayan sido resueltas y se hayan instalado las gemas, bundler crea un fichero Gemfile.lock, en el cual indica cuales gemas instaló y además incluye la versión exacta instalada. De esta manera permite reinstalar todas las dependencias sencillamente aunque en los repositorios oficiales aparezcan nuevas versiones.

A continuación se muestra un fichero de ejemplo Gemfile (izquierda) y el fichero Gemfile.lock (derecha) que se genera tras instalar una gema.

```
source "http://rubygems.org"
gem 'chipmunk_android'
```

```
GEM
  remote: http://rubygems.org/
  specs:
    chipmunk_android (0.0.1)

PLATFORMS
  dalvik

DEPENDENCIES
  chipmunk_android
```

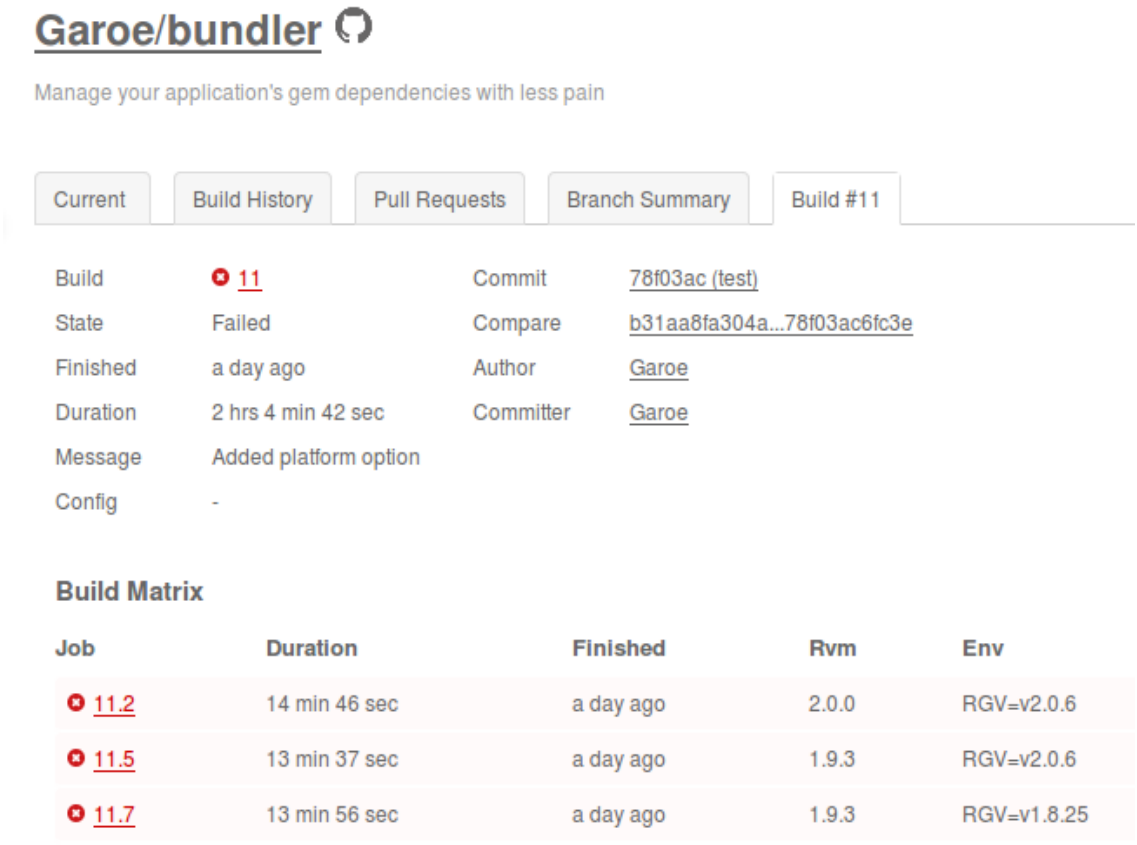
Para el testeo automático de un proyecto tan grande y complejo como Bundler se usa Rake como lanzador de pruebas y **Travis CI** como entorno en el cual se ejecutan.

2.4.6. Travis CI

Travis CI [32] es un servicio de integración continua para crear y probar proyectos que se encuentran en Github, para ello usa servidores en los cuales se usan máquinas virtuales, en los que se instala el software a probar y se realizan los tests. Es software libre y está diseñado para ser usado en proyectos de software libre. Para empezar a usarlo se ha de incluir un archivo .travis.yml en el cual se especifican opciones de construcción y pruebas.

Este software es especialmente útil debido a que, cada vez que se realiza un commit a una rama en la que se ha habilitado, se realizan automáticamente todas las pruebas. Y cuando estas terminen se avisará al desarrollador mediante un email/irc o el medio que este haya configurado. Además si alguien realiza un fork de un proyecto, y luego hace un pull request, en el aparecerá si pasó los test y un enlace a los resultados.

A continuación se muestra una imagen de un trabajo enviado a Travis-CI con los resultados de las pruebas, en este caso fallidas:



Garoe/bundler

Manage your application's gem dependencies with less pain

Current | Build History | Pull Requests | Branch Summary | Build #11

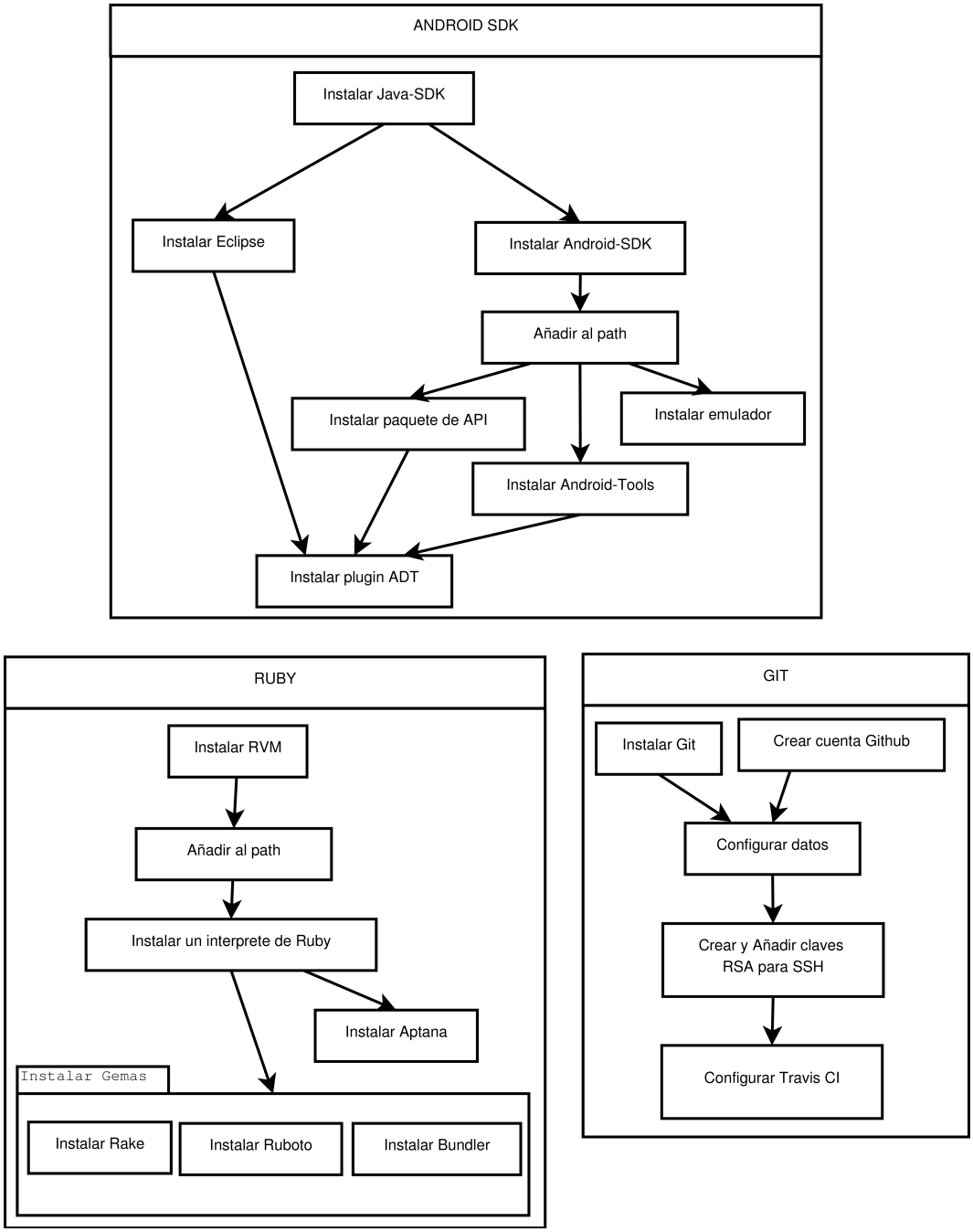
Build	11	Commit	78f03ac (test)
State	Failed	Compare	b31aa8fa304a...78f03ac6fc3e
Finished	a day ago	Author	Garoe
Duration	2 hrs 4 min 42 sec	Committer	Garoe
Message	Added platform option		
Config	-		

Build Matrix

Job	Duration	Finished	Rvm	Env
11.2	14 min 46 sec	a day ago	2.0.0	RGV=v2.0.6
11.5	13 min 37 sec	a day ago	1.9.3	RGV=v2.0.6
11.7	13 min 56 sec	a day ago	1.9.3	RGV=v1.8.25

2.5. Resumen

A continuación se muestra un diagrama que resume la instalación y configuración de todas las herramientas necesarias para tener un entorno funcional:



3. Análisis y Diseño

En este capítulo se procede a explicar los apartados de análisis de requisitos de usuarios y diseño del sistema software. Estas secciones no son tan extensas como deberían ser, teniendo en cuenta la envergadura del proyecto. Esto es debido al carácter experimental del programa y la falta de información y experiencia en el área de desarrollo.

3.1. Análisis

3.1.1. Requisitos

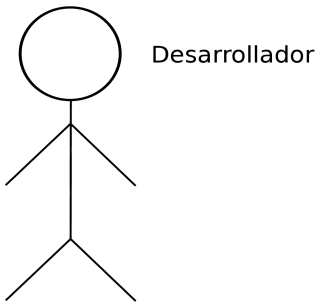
3.1.1.1. Glosario de conceptos

En primer lugar, es importante establecer un glosario de términos donde se explique claramente los conceptos del problema que nos planteamos, acompañados de una definición textual.

- **Ventana:** área donde el usuario podrá dibujar.
- **Imagen:** recurso de tipo tanto mapa de bits como vectorial, de cualquier codificación, png, jpg, etc.
- **Sonido:** recurso de tipo audio de corta duración guardado en cualquier codificación, mp3, wav, etc.
- **Canción:** recurso de tipo audio de larga duración guardado en cualquier codificación, mp3, wav, etc.
- **Espacio:** colección de cuerpo rígidos y formas, que además están sometidos a fuerzas tales como la gravedad.
- **Cuerpo rígido:** objeto puntual con masa e inercia.
- **Forma:** conjunto de vértices u otros atributos que definen una figura bidimensional.

3.1.1.2. Identificación de actores

Debido a que la aplicación es un motor para el desarrollo de juegos, solo existe un tipo de actor: el **programador** que usa el motor para crear su juego.



3.1.1.3. Identificación de procesos

En este apartado procedemos a listar las actividades que pueden llevar a cabo los usuarios de Gosu-Android y Chipmunk-Android.

Gosu-Android

- Crear ventana
- Cargar Imagen
- Dibujar Imagen
- Cargar Sonido
- Reproducir sonido
- Cargar canción
- Reproducir canción
- Pausar canción
- Parar canción
- Detectar/Leer input de teclado
- Detectar/Leer input táctil
- Dibujar caracteres (letras, números, etc)

Chipmunk-Android

- Crear espacios
- Crear cuerpos rígidos

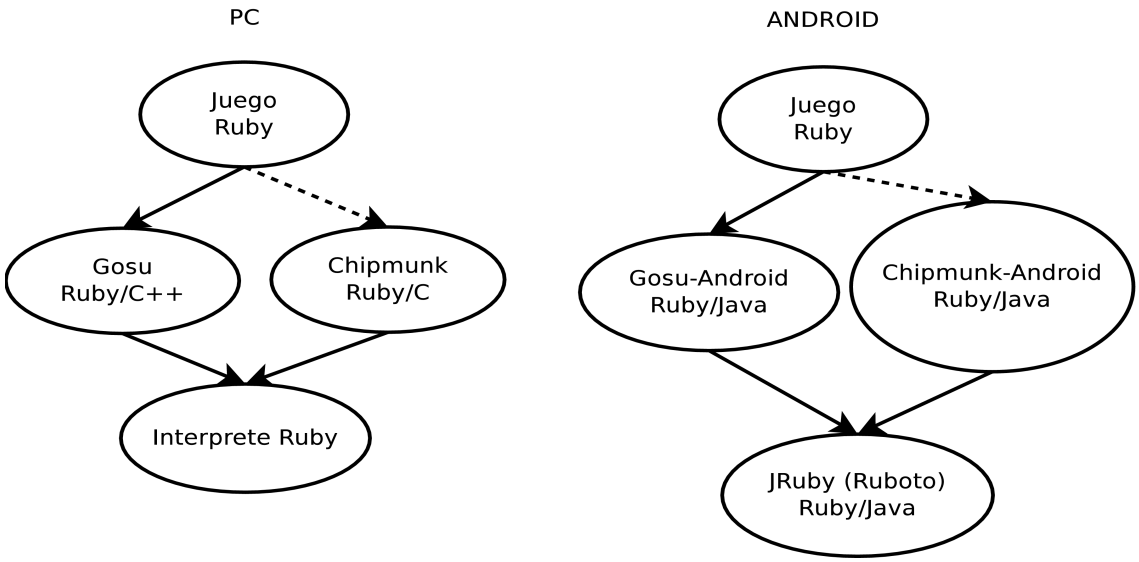
- Crear formas
- Añadir/Quitar formas a cuerpos rígidos
- Añadir/Quitar cuerpos rígidos/formas a espacios
- Ser informado de colisiones entre cuerpos

3.2. Diseño

Dado que no se tenia experiencia previa en el desarrollo de aplicaciones para móviles, ni tampoco se tenía experiencia en el uso de Ruboto, se optó por utilizar un diseño basado en **prototipos**. De esta manera, irme familiarizando con todas las plataformas e ir refinando cada vez más el resultado final.

3.2.1. Arquitectura del sistema

En primer lugar, presentamos un diagrama de como es la arquitectura del sistema que vamos a desarrollar, en contraposición a como funcionan los motores de juegos de los que tomaremos la API en PC:



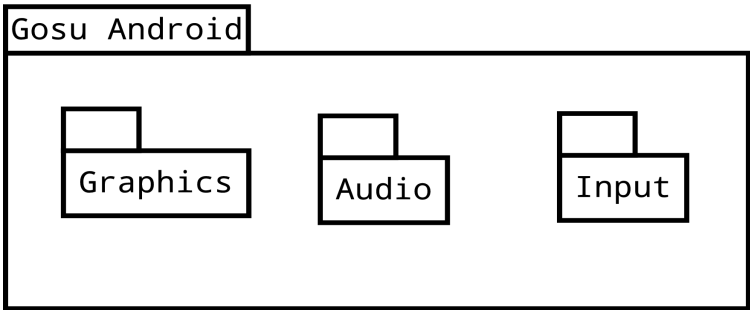
Como vemos en PC tenemos un videojuego escrito en Ruby, que usa los motores Gosu y Chipmunk, que al final acaba ejecutando el código Ruby en un interprete (cualquiera de los que presentamos previamente). Mientras que en Android tenemos el mismo videojuego en Ruby, pero en este caso usará Gosu-Android y Chipmunk-Android, mientras que el interprete será obligatoriamente JRuby modificado por Ruboto. Por supuesto, el uso de Chipmunk es opcional, tal como indica la flecha discontinua.

3.2.2. Diagrama de clases

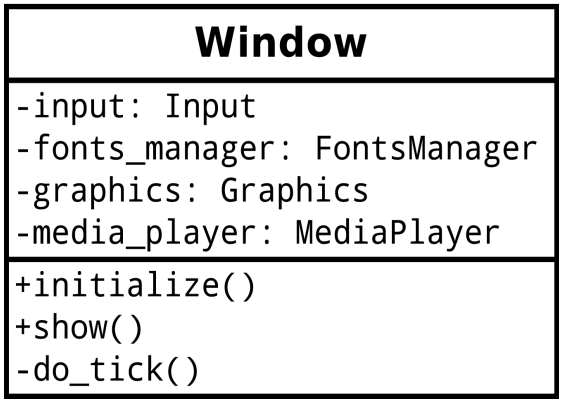
Una vez tenemos clara la arquitectura del sistema, procedemos a diseñar los diagramas de las clases que se han de implementar, comencemos en primer lugar con Gosu-Android.

Gosu-Android

En Gosu-Android distinguimos tres subsistemas: vídeo, audio e input. Por claridad, trataremos cada uno por separado:



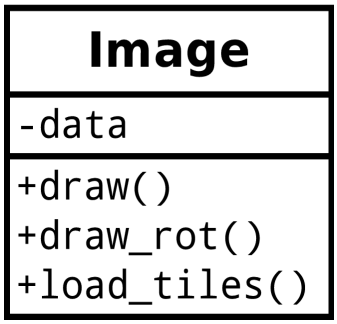
Donde la clase principal será **Window**, en ella tendremos el bucle principal y desde esta se tendrán referencias al resto de sistemas y por tanto Window será la encargada del control de más alto nivel. Tanto en el diagrama de clases siguiente como en todos los demás, solo se muestran los atributos y métodos más relevantes, siendo el número total mucho mayor.



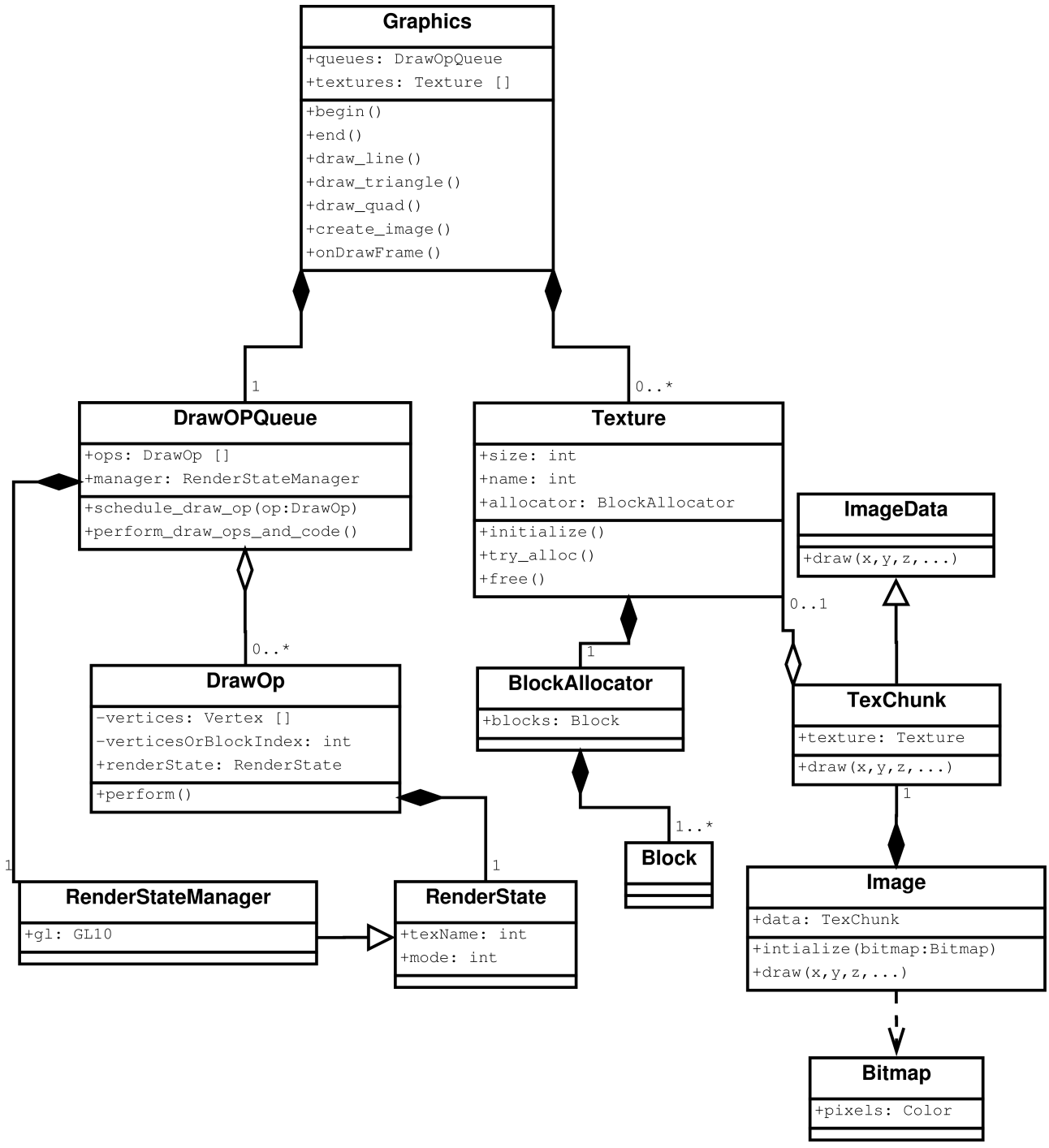
En la clase Window harán falta tener instancias de las clases controladoras de cada subsistema. Las clases serán Input, Graphics y MediaPlayer para manejar el input, el vídeo y el audio respectivamente. Además de una clase extra para escribir texto.

Graphics

En graphics la clase más importante para el usuario es la clase **Image**, puesto que es la clase con la crea y dibuja imágenes.



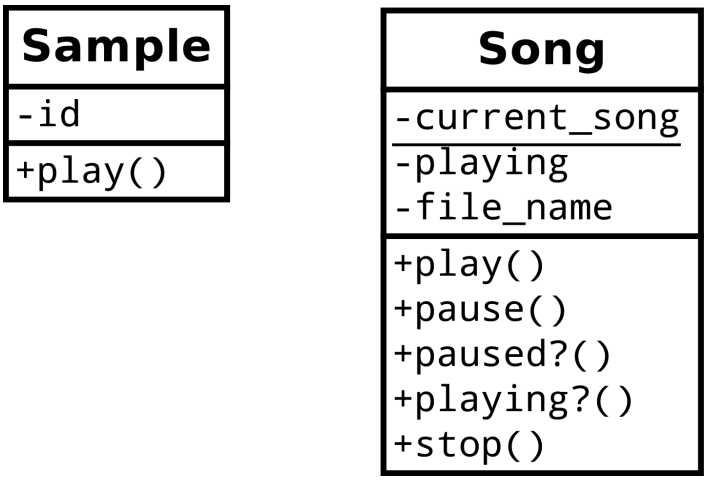
Internamente la clase **Graphics** será la encarga de manejar cualquier método que tenga que ver con el vídeo. Dado que queremos un esquema sencillo y desacoplado, el **Render** solo entenderá de un tipo de operaciones de dibujo, que será la clase **DrawOP**, habrá una cola **DrawOpQueue** que contiene operaciones simples de dibujo **DrawOp**.



DrawOp tiene en el atributo RenderState si la operación es sencilla, es decir sin textura, o con textura. Además de incluir otras transformaciones varias y distintos modos de renderizado.

Audio

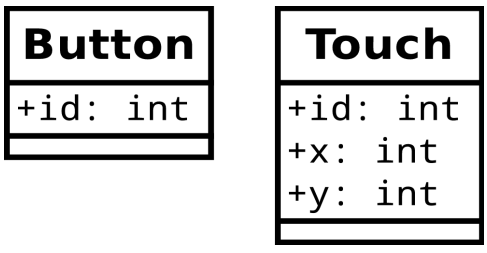
En audio tenemos dos clases principales de cara al usuario, la clase **Sample** y la clase **Song**. Sample, es un sonido de corta duración y que puede ser reproducido repetidas veces, por ejemplo un efecto como disparo de una bala, una explosión, sonido de salto, etc. Mientras que la clase Song, es usada para sonidos largos, normalmente canciones, solo puede haber una en reproducción en un momento dado.



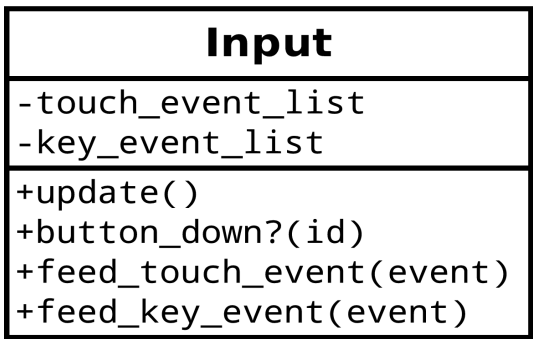
Debido a que la clase Sample es muy sencilla, basta tener un identificador del archivo asociado a la instancia de la clase y un método play para reproducirlo. Mientras que con clase Song, al solo permitir que se reproduzca una instancia cada vez, debe tener más variables internas y métodos que ayuden con la lógica del sistema.

Input

Se distinguen dos clases principales para el usuario, **Button** y **Touch**. Button, representa una tecla pulsada. Mientras que Touch, representa un toque en la pantalla del dispositivo táctil, por tanto tiene coordenadas X e Y, así como un identificador único.



Internamente todo es manejado por la clase **Input**.

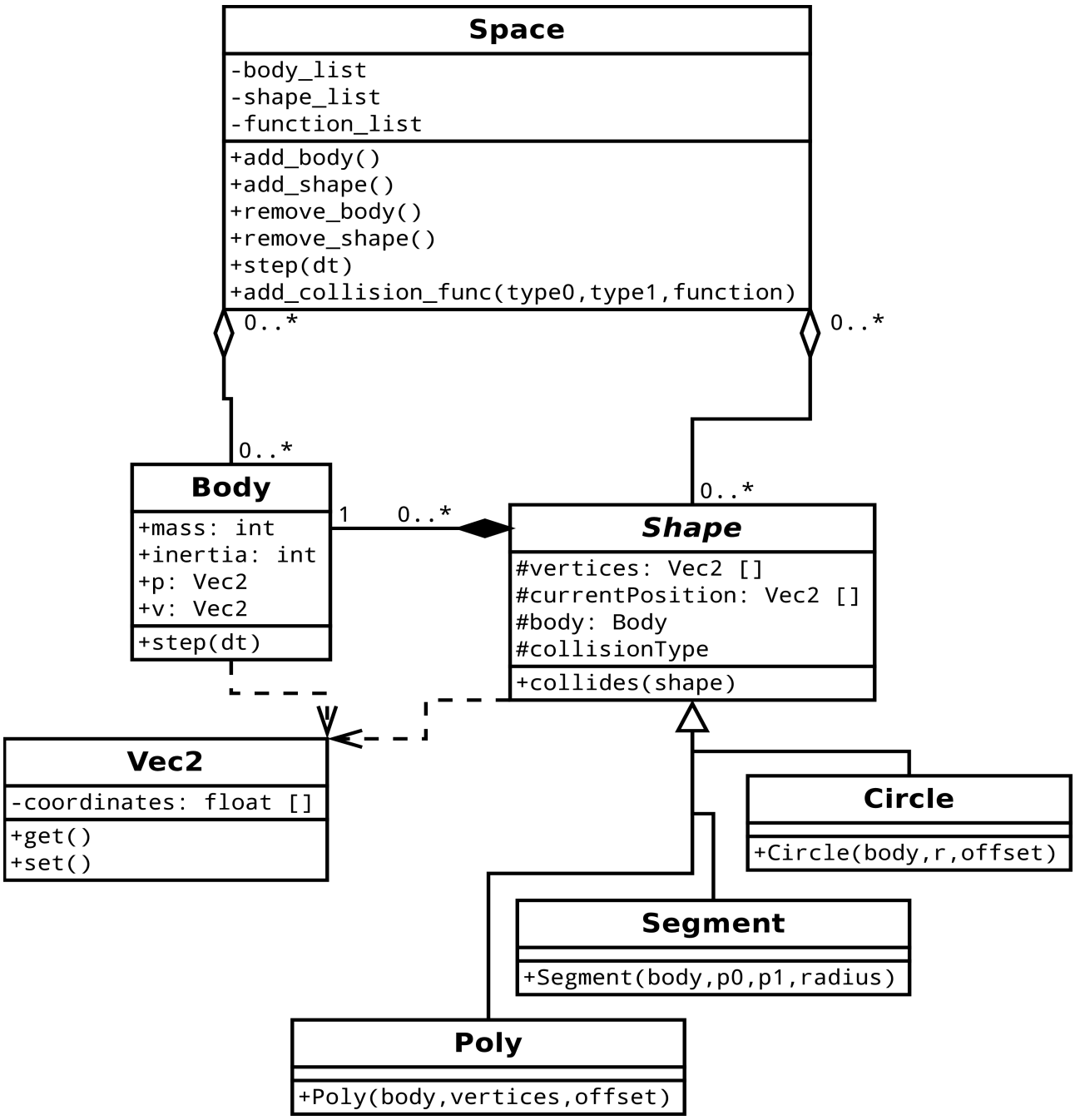


Tenemos dos listas: una para eventos táctiles y otra para eventos de teclado. Serán necesarios métodos para introducir elementos en las listas, así como métodos para actualizar el estado. Por otra parte al usuario se le informará de los distintos tipos de toques, es decir presionar, arrastrar y levantar.

Chipmunk Android

Chipmunk Android contiene las siguientes clases:

- **Vec2:** Vector de dos coordenadas.
- **Body:** Un punto, al que se le definen, masa, velocidad y momentum.
- **Shape:** Define una forma, que se añade a un objeto, sirve para definir colisiones entre Bodies.
- **Space:** Un espacio al que se le añaden, Bodies y Shapes, tiene atributos que define como se comportaran los objetos, tal como gravedad. Los distintos objetos solo interactuaran entre si, si están en el mismo Space.



Un Space se compone de varios Bodies y Shapes que interactúan entre si. Por tanto, debe tener métodos para agregar y eliminar instancias de estos objetos. Por otra parte, un Body puede tener uno o mas Shapes. En la practica esto significa que se pueden definir distintos tipos de colisiones para el mismo objeto, con un circulo, un cuadrado, etc. Además, la clase Shape es abstracta y define las interfaces que todo objeto de tipo Shape debe tener. En principio, han de ser métodos de colisión con el resto de Shapes. También es útil para agrupar el código en común de todas las subclases de tipo Shape.

4. Desarrollo

Una vez instalado todo el software necesario, se decidió que lo primero sería crear el código de Gosu-Android, de esta manera se podría observar resultados gráficos muy rápidamente. En primer lugar crear una ventana sin nada, luego dibujar formas sencillas (líneas, triángulos, cuadrados), a continuación dibujar imágenes. Posteriormente incluir audio y por último input. Al terminar las clases básicas de Gosu-Android, se procedió a implementar Chipmunk-Android, debido a limitaciones temporales, del motor de físicas solo se pudieron implementar las clases más básicas y los métodos más comunes.

Todo el código desarrollado, así como la documentación se encuentran disponibles en: https://github.com/Garoe/gosu-android y en https://github.com/Garoe/chipmunk_android

4.1. Gosu-Android

4.1.1. Versión 0.1→ IRB

Android provee de dos opciones a la hora de dibujar objetos en pantalla. Una API nativa de Android o usar OpenGL. Después de probar ambas y consultar foros donde si discutía cual era mejor. Se decidió usar OpenGL puesto que es más rápida, es un estándar por lo cual siempre es más fácil de portar y mantener código, además OpenGL proporciona muchas más funcionalidad con respecto a la nativa de Android.

OpenGL ES 1.0 vs OpenGL ES 2.0

Android usa un subconjunto de OpenGL llamado OpenGL ES → OpenGL for Embedded Systems. Este subconjunto está especialmente diseñado para dispositivos móviles, sin embargo no ofrece todas las características de la versión completa. Por otra parte, ofrece mejoras tales como la ventaja de no tener que aislar las llamadas a métodos de OpenGL entre los bloques glBegin y glEnd.

Android también ofrece dos versiones a elegir por el programador, la 1.0 y la 2.0. La primera versión es soportada por dispositivos con Android 1.0 o superior, mientras que la segunda es para Android 2.2 o superior. Las principal diferencia radica en que la versión 2.0 incluye el manejo de Shaders, mientras que la 1.0 no. Debido a problemas con el uso de shaders en Ruby, así como para lograr una mayor compatibilidad, se optó por usar la 1.0.

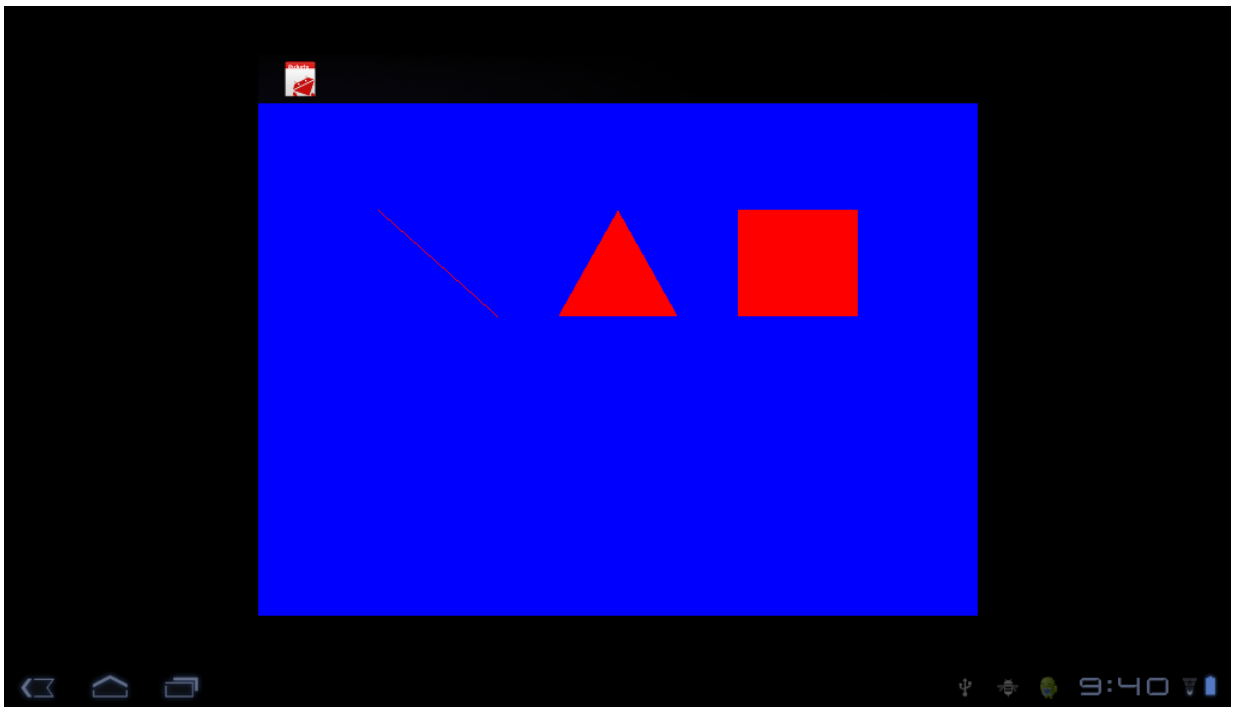
Al momento de escribir esta memoria, acaban de lanzar OpenGL ES 3.0 para Android 4.3, cuyas características incluyen: mejoras en el renderizado, nueva versión del lenguaje de shaders, mejoras varías con texturas, etc.

Ventana vacía

Debido a la simplicidad y rapidez para observar resultados, el primer prototipo del motor se desarrolló usando la aplicación Ruboto-IRB, junto con el script demo-irb-server.rb. Tomando como referencia el tutorial de OpenGL para Android y varios de los scripts de ejemplo que proporciona ruboto, se logró mostrar una ventana creada con OpenGL desde Ruby.

Formas básicas

A continuación se incluyó la posibilidad de dibujar líneas, triángulos y cuadrados, en distintos colores. Así como permitir cambiar el color del fondo del área de dibujo.



Input

Android distingue entre dos tipos de input, pulsar una tecla: tanto de teclado en pantalla como teclado físico, o toques en la pantalla. En esta versión solo se pudo implementar para teclas, para capturar las pulsaciones de botones se ha de establecer el listener en la clase Activity:

```
def add_key_event_listener
  # Get the class of the object.
  @activity.class.class_eval do
    def onKeyDown(keyCode, event)
      if @input.feed_key_event_down(keyCode)
        return true
      else
        return super keyCode, event
      end
    end

    def onKeyUp(keyCode, event)
      if @input.feed_key_event_up(keyCode)
        return true
      else
        return super keyCode, event
```

```
    end
  end
```

Esta parte del código fue especialmente complicada debido a que la clase Activity es definida por el usuario de Gosu-Android. Sin embargo, el motor puede acceder a la instancia de dicha clase y con ella aprovechando que Ruby es un lenguaje dinámico se puede acceder a la clase y añadirle los métodos necesarios para poder capturar los eventos.

También es importante resaltar que se debe devolver true cuando se ha manejado el evento y false en caso contrario. Tal como se observa en el código, si se maneja se devuelve true y si no se llama al método de la clase padre y se devuelve su valor. En caso de no hacer esto, podría haber problemas para seguir recibiendo eventos de teclado.

Problemas principales resueltos e información aprendida

Esta primera implementación del código fue una de las partes más costosas en cuanto a tiempo de todo el proyecto. Ya que no existían conocimientos previos de ninguna de las herramientas usadas.

Uno de los conceptos más importantes fue como importar una clase de Java para poder usarla en Ruby, puesto que se necesitan manejar las clases de Android. Para importar una clase se usa la siguiente línea de código:

```
java_import "android.opengl.GLSurfaceView"
```

Después de que haya sido ejecutada, se ha creado en el contexto actual la clase que se ha importado. Debido a que hubo problemas de solapamiento de clases, por ejemplo si creamos una clase que se llama igual que una importada, se decidió que todas las clases de Java se importarían dentro de un módulo para tener un espacio de nombres bajo control:

```
module JavaImports
  #Opengl dependencies
  java_import "javax.microedition.khronos.egl.EGL10"
  java_import "javax.microedition.khronos.egl.EGLConfig"
  java_import "javax.microedition.khronos.opengles.GL10"
end
```

Una parte bastante compleja fue el uso de listeners en el código Ruby para eventos del sistema en Java. Para resolverlo se siguió un tutorial de la wiki de ruboto[33], sin embargo no siempre funcionaba tal como debería. A continuación se muestran varias manera de establecer callbacks de Ruby a Java.

Cuando solo se va a usar el listener para una función:

```
button.setOnClickListener do |view|
  # Do something
end

button.on_click_listener = proc do |view|
  # Do something
end
```

Cuando se requiere una clase listener con varias funciones:

```
class MyCustomView < android.view.View
  def onDraw(canvas)
    super
    # Do Somethng
  end

  def onDestroy()
    super
    #Do Somethng
  end
end
```

Se aconseja usar siempre los proc debido a que usar el bloque sin más a veces resultaba en que el código nunca se llamaba.

Al tener que usar OpenGL ES se descubrió que tenia varias limitaciones, como por ejemplo el hecho de que no se pueden dibujar quads directamente. En ese caso la solución consistió en dibujar dos triángulos adyacentes:

```
  @gl.glDrawArrays(JavaImports::GL10::GL_TRIANGLE_STRIP, 0, 4)
```

Por otra parte, en OpenGL se usan las llamadas mediante Arrays debido a que son más eficientes que pasar los datos uno a uno en un bucle:

```
@gl.glColorPointer(4, JavaImports::GL10::GL_FLOAT, 0, color_buffer)
@gl.glVertexPointer(3, JavaImports::GL10::GL_FLOAT, 0, vertex_buffer)
@gl.glTexCoordPointer(2, JavaImports::GL10::GL_FLOAT, 0, texture_buffer)
```

OpenGL ES también tiene otra limitación bastante importante con respecto a su versión estándar. Mientras que en la estándar se incluye la librería y se hacen las llamadas en cualquier lugar del código, en ES todas las llamadas se han de hacer usando la **variable de contexto gl**, esta variable se recibe como parámetro en ciertas funciones:

```
def onDrawFrame(gl)
  @gl = gl
end
```

Tal como se observa la solución fue guardarla y actualizarla para poder usarla desde otros contextos.

4.1.2. Versión 0.2→ Aplicación Ruboto

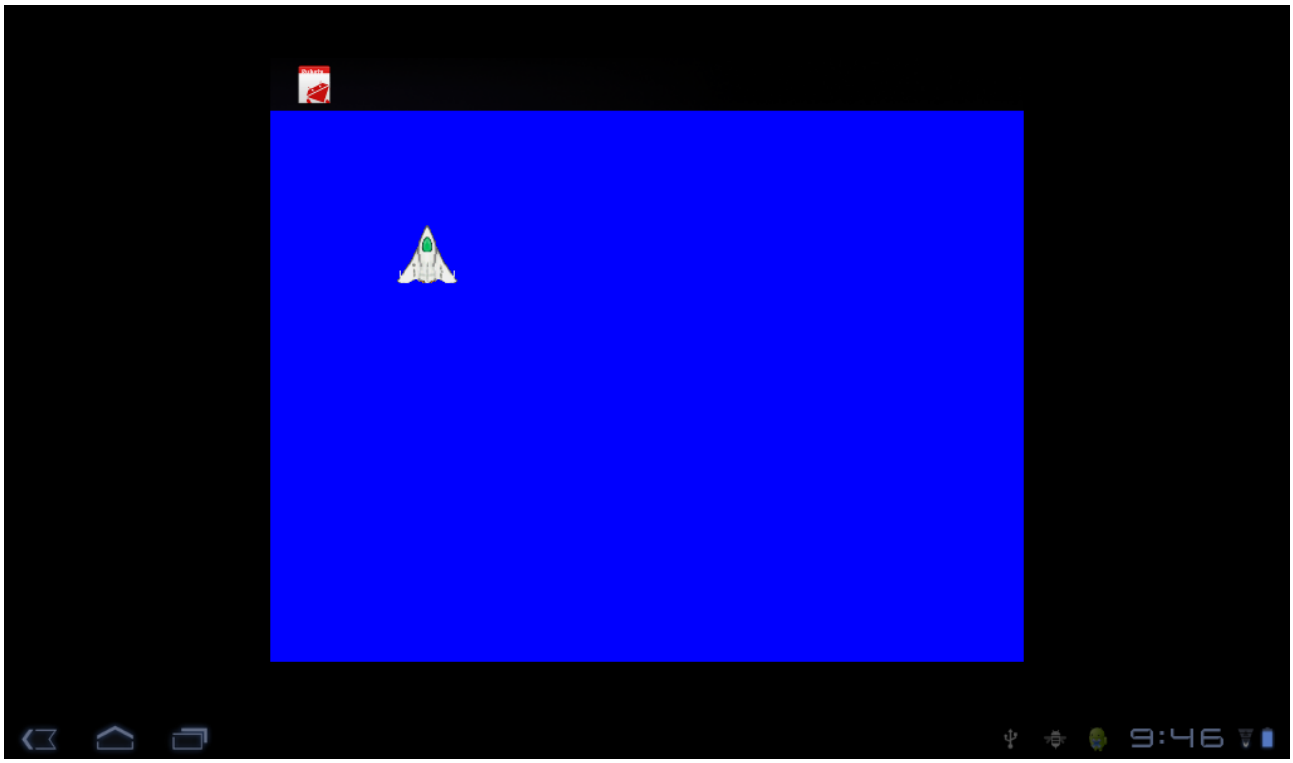
Debido a cuestiones de implementación era necesario crear una subclase Ruby de la clase Java GLSurfaceView, para ello Ruboto proporciona el comando ruboto gen class BaseClass –name NewClass. Sin embargo este comando solo sirve para aplicaciones creadas con ruboto, no se puede usar directamente con el IRB. Por tanto en este punto se paso de usar el IRB a usar el generador de aplicaciones.

En concreto, la clase GLSurfaceView es la responsable de las vistas en OpenGL. A esta clase fue necesario añadirle atributos extras, para el manejo de eventos y para poder establecer opciones de configuración. Por ejemplo, para capturar los toques se debe añadir un listener en dicha clase, tal como se muestra a continuación:

```
def onTouchEvent(event)
  super
  @input.feed_touch_event(event)
  return true
end
```

Imágenes

Una vez completadas las formas básicas, era hora de permitir dibujar imágenes. Para ello, el procedimiento consiste en dibujar un quad y luego mapear la textura (imagen) en dicha figura. Por simplicidad y coherencia se crea el quad del tamaño en píxeles de la imagen que se quiera dibujar.



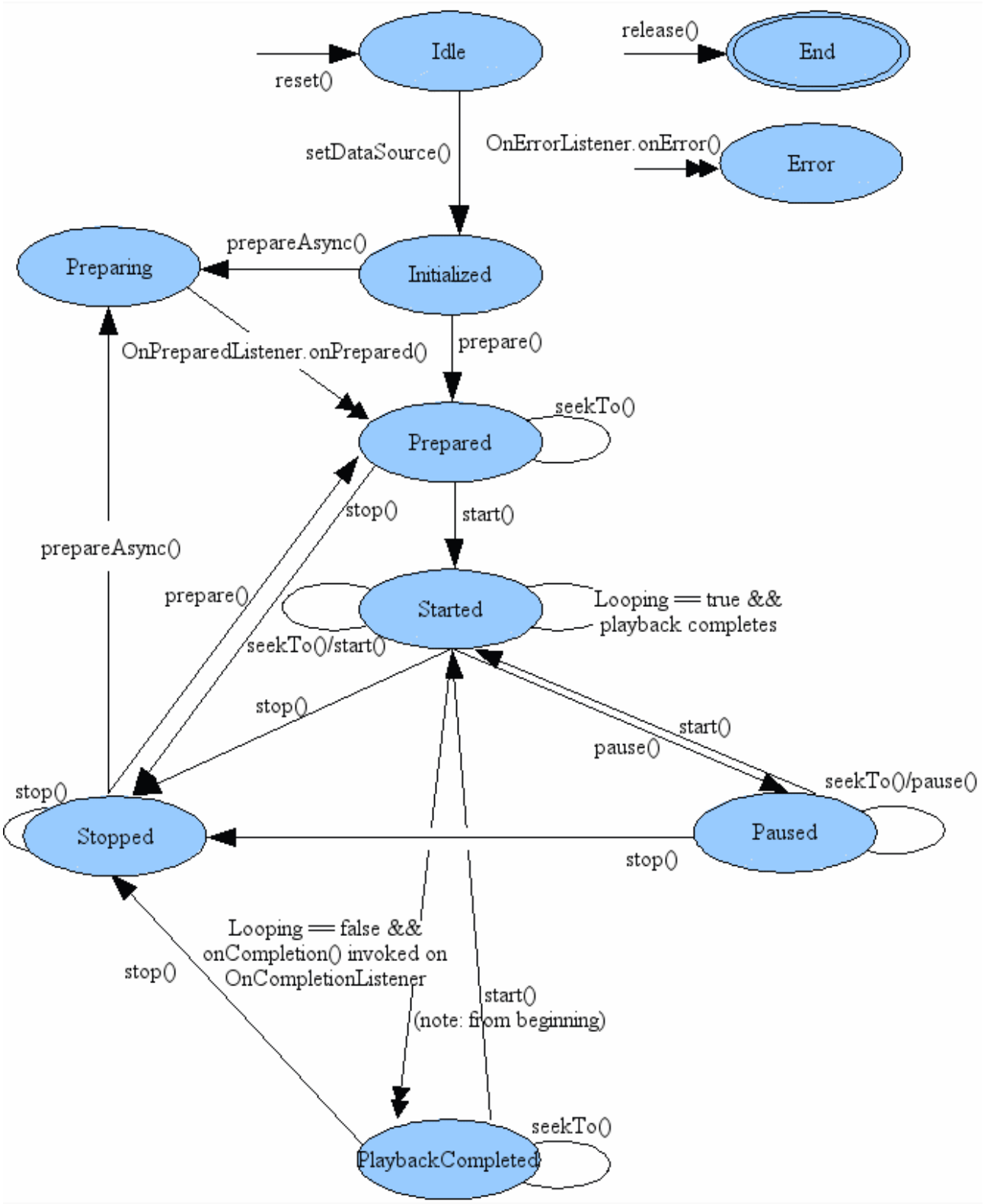
Sonido

Para reproducir sonido, Android pone a disposición del desarrollador dos mecanismos.

El primero es **SoundPool**, un contenedor de recursos de audio en memoria, en el que se pueden reproducir varios de estos recursos a la vez. Usando esta API se implementó la clase **Sample**, cabe resaltar lo sencillo que es de usar el código, observando las partes más relevantes tenemos:

```
#Crear un nuevo SoundPool
@@pool = JavaImports::SoundPool.new(MAX_SAMPLES,
JavaImports::AudioManager::STREAM_MUSIC, 0)
#Cargar un nuevo recurso
@id = @@pool.load(filename, 1)
#Reproducirlo
@stream_id = @@pool.play(@id, volume, volume, 1, 0, 1.0)
```

Como segunda opción existe el **MediaPlayer**, una clase usada para controlar la reproducción de streams de audio o vídeo. Esta clase funciona internamente como una máquina de estados, y es muy importante tener en cuenta en cual estado está y por tanto cuales son los métodos que se pueden utilizar, pues en caso de usar un método en el estado equivocado saltará una excepción. El diagrama de estados es el siguiente:



Gráfica de estados de la clase MediaPlayer → http://developer.android.com/images/mediaplayer_state_diagram.gif

Otro punto importante a resaltar es el hecho de que la clase MediaPlayer ofrece métodos **síncronos** y **asíncronos** para distintas tareas, como pueden ser cargar un recurso de audio. Por tanto, si se usa el método síncrono no retornará del método hasta que este haya completado su tarea. Mientras que si se usa el asíncrono, hay que establecer un listener para que nos avise cuando termina. En esta primera implementación se usó el método síncrono. También hay que tener muy presente en cual estado se encuentra la instancia de la clase MediaPlayer que se está usando, pues esto determina cuales son los métodos a los que se nos está permitido llamar. Esto cobra especial importancia cuando se cargan varios recursos de audio, puesto que cada recurso comparte la instancia del MediaPlayer.

En un principio no se sabia de la existencia de Chipmunk, por lo que en este momento se implementó un pequeño motor de físicas integrado en el código de Gosu-Android. Sin embargo, la API de la parte de físicas no estaba implementada en la versión de PC, por lo tanto esto creaba incompatibilidades entre las dos versiones. Esto llevo a buscar alternativas, que concluyeron con el descubrimiento de Chipmunk. Por lo que todo el código desarrollado para el motor interno fue descartado.

Problemas principales resueltos e información aprendida

Un concepto interesante en Ruby es que se pueden asignar definiciones de clases a variables, es decir:

```
GosuSurfaceView = TouchSurfaceView
class GosuSurfaceView
  def atributes
     @new_attr = 1
  end
end
```

Con este código se crea la variable GosuSurfaceView, que es en realidad la clase TouchSurfaceView y por tanto se le está añadiendo el atributo new_attr a la clase TouchSurfaceView. Esto fue necesario dado que se necesitaba añadir ciertos atributos a esta clase y no se podía hacer directamente.

Por otra parte en Ruby tenemos la limitación de que solo podemos tener un constructor por clase, por lo que hubo que recurrir a usar constructores con argumentos de tipo y número variable.

Otro punto importante, es que en Android solo el hilo que crea la interfaz de usuario tiene permisos para cambiarla, sin embargo los callbacks los ejecuta un hilo distinto al de la interfaz. Para poder cambiar opciones de la interfaz es necesario usar el siguiente código:

```
p = Proc.new do
  @window.setLayout(@width, @height)
  @activity.setTitle @caption
end
@activity.runOnUiThread(p)
```

Se crea un **Proc** con el código que modifica los parámetros y luego se manda a ejecutar dicho proc en el hilo que creo la interfaz usando el método **runOnUiThread**.

Por otra parte es muy importante usar la configuración adecuada al cargar imágenes en Android, pues este provee de dos decodificadores y estos admiten varias opciones de input. El desconocimiento de cual era la configuración adecuada provocó en primeras versiones problemas tales como imágenes que se cargaban con tamaños cambiados, distintos colores o transparencias que se mostraban con un color.

4.1.3. Versión 0.3 → Optimización

Al terminar el código se observó que tanto el tiempo de arranque y más especialmente el tiempo de ejecución era muy lento. El primero de unos 40 segundos mientras que el segundo del orden de unos 10 a 15 fps. Por tanto había que optimizar el código para conseguir mejorar ambos parámetros.

Para descubrir donde estaba el cuello de botella, en primer lugar se pensó en usar alguna de las gemas que existen para Ruby que proporcionan herramientas de profiling. Sin embargo, se descubrió que todas tenían alguna porción de código en C++. Por tanto, era imposible ejecutarlas bajo Android. Por lo que no quedó más remedio que recurrir a timers explícitos y consultar logs para ver resultados.

Después de un testeo intensivo se descubrió que la parte gráfica, en concreto el dibujo de texto era el principal responsable. Esto era debido a que se creaba el texto en cada frame. Por lo que se optó por crear un archivo con todos los caracteres ASCII que se cargaba al inicializarse la aplicación.

Aunque esto produjo el incremento deseado en la cantidad de fps, produjo un incremento en el tiempo de arranque de la aplicación.

Al programar esta versión, el equipo Ruboto lanzó una nueva versión que incluía entre otras características la posibilidad de crear subclases de Java en Ruby de manera dinámica. Para ello se usaba el método ruboto_generate, por ejemplo:

```
ruboto_generate(android.opengl.GLSurfaceView => "TouchSurfaceView")
```

Con esta llamada estamos creando la subclase TouchSurfaceView en Ruby a partir de la clase Java android.opengl.GlSurfaceView. Por lo que hubo que adaptar el código para poder aprovecharnos de esta utilidad.

El proceso de optimización fue iterativo, una vez hecha las mejoras previas se realizaron otras optimizaciones menores, tal como usar el método asíncrono del MediaPlayer.

Problemas principales resueltos e información aprendida

Cuando se estaba programando este código salió una versión nueva del Java SE Development Kit (JDK) y esta actualización se instaló inadvertidamente. El código dejó de compilar, por lo que la lección es clara, mantener siempre la misma versión de las librerías a lo largo de todo el desarrollo de la aplicación.

4.1.4. Versión 0.4 → Java

Teniendo en cuenta los problemas de eficiencia a los que se enfrentaba el motor, se optó por reescribir las partes más problemáticas del motor en Java.

Usando los datos de profiling obtenidos anteriormente se decidió pasar a Java las siguientes clases: **Bitmap**, **ClipRect**, **Color, DrawOp**, **DrawOpQueue**, **RenderState**, **RenderStateManager** y **Vertex** .

Al reescribir dicho código en Java se logró alcanzar unos 30 fps y un tiempo de inicio menor a 10 segundos. Sin embargo, se empezaron a notar parones en los fps de manera más o menos regular. Esto es debido al colector de basura de Java, puesto que la cantidad de memoria que tenía que liberar era importante. En un primer momento se intento modificar los parámetros del G.C. para evitar que se ejecutara en lo posible. Sin embargo, estos esfuerzos fueron infructuosos, ya que en la implementación de la Java Virtual Machine para P.C. si permite modificar estos parámetros. Mientras que para Android esto no es posible. Por lo que se optó por darle el menor trabajo posible, para ello se intentó evitar en la medida de lo posible pedir y liberar memoria en cada frame. Esto se logró en parte mediante la clase **DrawOpPool**, con esta clase pre-reservamos memoria para unos 100 objetos **DrawOp**, que luego usará el usuario de forma transparente. Además se realizó hacer una limpieza de **variables locales** en favor de **variables de clase** o **de instancia**. Puesto que el ámbito de las variables locales se pierde en cada frame. Estas han de ser creadas de nuevo en cada frame con el consiguiente trabajo extra para el colector de basura.

Reescribir el código en Java no solo afectó a la performance, sino que debido a características intrínsecas de cada lenguaje el código quedo en ciertos sitios simplificado y en otros se complicó.

En los puntos negativos cabe resaltar los bucles, el acceso a variables de clases (se han de hacer métodos de acceso) y en general el aumento de líneas de código. Ya que en Ruby los bucles se simplifican mucho con el uso de bloques y con una etiqueta nos ahorramos el tener que escribir métodos de acceso a variables de clases, por ejemplo con la clase Bitmap, se muestra a la izquierda la implementación en Ruby y a la derecha en Java:

```ruby
i = 0
self[:vertices].each do |vertex|
  index.push vertex.x
  index.push vertex.y

  case i
  when 0
    texture.push left, top
  when 1
    texture.push right, top
  when 2
    texture.push left, bottom
  when 3
    texture.push right, bottom
    i = -1
  end
  i += 1
end
```

```java
int i = 0;
int k = 0;
for(int j = 0; j < verticesOrBlockIndex; j++){
  index[k] = vertices[j].x;
  index[k+1] = vertices[j].y;

  switch(i){
  case 0:
    texture[k] = left;
    texture[k+1] = top;
    break;
  case 1:
    texture[k] = right;
    texture[k+1] = top;
    break;
  case 2:
    texture[k] = left;
    texture[k+1] = bottom;
    break;
  case 3:
    texture[k] = right;
    texture[k+1] = bottom;
    i = -1;
  }
  k += 2;
  i++;
}
```

Sin embargo, hubo puntos positivos de pasar el código a Java, tales como los constructores de clase con mismo nombre pero distinto parámetros, o poder aprovechar las características nativas de lenguaje. Ya que en Ruby si se define un método con el mismo nombre que otro, aunque tenga distinto número y/o tipo de argumentos, el primero es sobrescrito. Además, en la clase Color interesaba usar el overflow en ciertos cálculos, sin embargo en Ruby este no se produce puesto que transforma la variable automáticamente a otra que pueda contener el nuevo número y hubo que emularlo por software:

```
def force_overflow(i)
  if i < -2147483648
      i & 0xffffffff
  elsif i > 2147483647
      -(-(i) & 0xffffffff)
  else
      i
  end
end

def initialize(*args)
  case args.length
  when 0

  #Constructor for literals of the
form 0xaarrggbb.
  when 1
    initialize((args[0] >> 24) &
0xff, (args[0] >> 16) & 0xff,
           (args[0] >> 8) & 0xff,
(args[0] >> 0) & 0xff)

  #Constructor for rgb
  when 3
    initialize(0xff, args[0], args[1],
args[2])

  #Constructor for argb
  when 4
      @rep = force_overflow((args[0] <<

ALPHA_OFFSET) | (args[1] << RED_OFFSET)
| (args[2] << GREEN_OFFSET) | (args[3]
<< BLUE_OFFSET))
  else
    raise ArgumentError
  end
end
```

```
    public Color(){
     rep = 0;
    }

    public Color( long argb ){
     rep = RGBAtoColor( (int)((argb >>
24) & 0xff), (int)((argb >> 16) &
0xff),
                  (int)((argb >>  8) &
0xff), (int)((argb) & 0xff) );
    }

    public Color( int red, int green,
int blue ){
        rep = RGBAtoColor(0xff, red,
green, blue);
    }

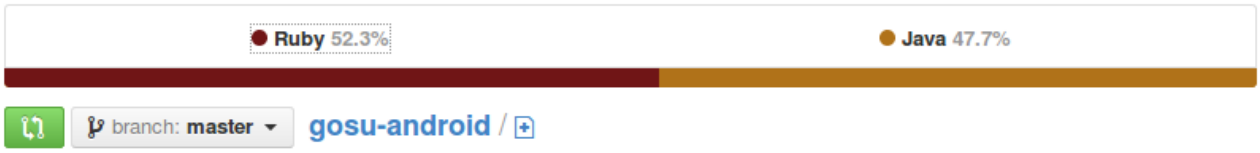
    public Color( int alpha, int red,
int green, int blue ){
        rep = RGBAtoColor(alpha, red,
green, blue);
    }
```

Por otra parte, mientras se estaba implementando esta versión se publicó una nueva versión de Ruboto que de nuevo tenía cambios con respecto a como se crean subclases Ruby de clases en Java. En concreto ya no era necesario llamar a ningún método específico sino que bastaba con usar la sintaxis de Ruby para definir subclases:

```
class GosuSurfaceView < JavaImports::GLSurfaceView
```

Y de esta manera tan sencilla se podían crear ahora dinámicamente y sin ningún problema subclases.

El porcentaje final de código escrito en cada lenguaje es el siguiente:

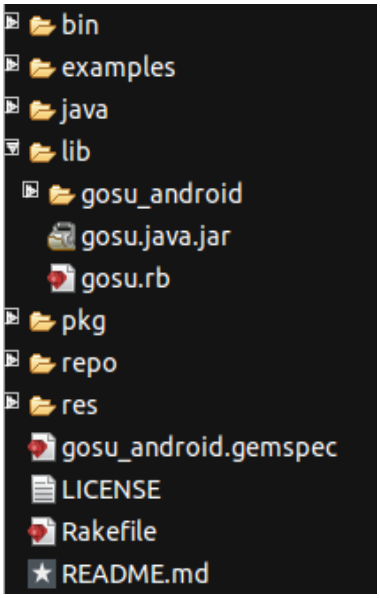


Problemas principales resueltos e información aprendida

Para integrar el código en Java con el proyecto en Ruby no basta simplemente con reescribir los archivos en java. Estos archivos se deben compilar y comprimir en un archivo jar. Este archivo debe ser importado tal como se importaban las clases nativas de Android. Este proceso está automatizado usando los comandos rake. Para mas información consultar el fichero rake pues en el están los comandos para compilar en java.

4.1.5. Versión 0.5 → Profesionalidad

Para la versión final se refactorizó el código para cumplir con los estándares que usan en RubyGems, se añadió el archivo gosu_android.gemspec con toda la información relevante para poder subir el código como una Gema.



Bin

Esta es una carpeta opcional, en ella se encuentra un ejecutable que se puede usar opcionalemente para facilitar la instalación de la gema.

```
#!/usr/bin/env ruby

begin
  require 'gosu_android/commands/base'
rescue RuntimeError
  puts $!.message
  exit 1
end

# Run Base, which will handle actual commands
Gosu::Commands::Base.main
```

Se trata de un script que simplemente carga el contenido del fichero base.rb situado en lib/gosu_android/commands, los comandos que se pueden ejecutar son los siguientes:

```
Usage: gosu_android OPTION
Adds or deletes dependencies for Gosu in a Ruboto project.
If no argument is specified it displays this help.
```

```
 -a, --add adds the files to the project
 -d, --delete deletes the files from the project
 -v, --version shows current version
 -h, --help display this help and exit

The options -a and -d can not be given together.

Exit status:
 0 if everything went ok,
 1 if there was some error.
```

El binario está implementado siguiendo el estándar de bash, con -a no solo añade los archivos del motor, si no también todos los recursos de audio y vídeo, mientras que con -d borra todos los archivos que incluyó con -a.

Examples

Esta es otra carpeta opcional en donde se sitúan todos los archivos de ejemplo que se quieran proporcionar junto con la gema.

Java

Se trata de otra carpeta opcional, en esta se encuentran todos los archivos que se han escrito usando Java.

Lib

La organización recomendada por RubyGems consiste en tener el código principal de la gema en una carpeta cuyo nombre coincida con el de la gema, en nuestro caso gosu_android dentro de una carpeta llamada lib. Además de un archivo principal, que cargará el resto de la librería llamado también como la gema. Tal como se observa en la imagen, el archivo no se llama gosu_android.rb sino gosu.rb. Esto es debido a que como entre los objetivos del motor está replicar la API de su contrapartida en PC, al nombrarlo de esta manera se evita que al incluir el motor, el usuario tenga que escribir código diferenciado para PC y para Android. Para evitar escribir un instalador se añade en la carpeta lib un archivo .jar con el código java ya compilado, siendo el código de este .jar el que se ejecutará.

Pkg

Esta carpeta se crea únicamente en local, no es subida al repositorio Github. En ella se generarán los archivos .gem que luego se subirán a RubyGems, para que puedan ser instalados por el público. O simplemente para ser instalados en local y hacer pruebas.

Repo

Se trata de otra carpeta que solo está en local, contiene un repositorio local de gemas. Fue usada extensivamente para comprobar que se estaba creando adecuadamente la gema, antes de subirla en el repositorio público.

Res

En esta carpeta están situados todos los recursos de audio y vídeo que usa el motor y los juegos de ejemplo.

Gosu_android.gemspec

En este archivo se especifica toda la información relevante de la Gema:

```
require 'date'
require 'rake'
lib_path = File.expand_path('lib', File.dirname(__FILE__))
$:.unshift(lib_path) unless $:.include?(lib_path)
require 'gosu_android/version'
require 'gosu_android/description'

Gem::Specification.new do |s|
  s.name = %q{gosu_android}
  s.version = Gosu::VERSION
  s.date = Date.today.strftime '%Y-%m-%d'
  s.authors = ['Garoe Dorta']
  s.email = %q{garoedp@gmail.com}
  s.summary = %q{A Gosu implementation for Android.}
  s.homepage = %q{https://github.com/neochuky/gosu-android/}
  s.description = Gosu::DESCRIPTION
  s.license = 'MIT'
  s.files = FileList['[A-Z]*', 'examples/{*,.*}', 'bin/*', 'lib/**/*',
'res/*/*'].to_a
  s.executables = %w(gosu_android)
  s.default_executable = 'gosu_android'
end
```

Entre otros valores, se establece, el nombre, la versión, fecha, autor, cuales archivos se incluirán, una descripción, la licencia que tiene la gema, etc.

Rakefile

Este archivo es usado para automatizar tareas útiles durante la compilación, publicación de la gema, pruebas, etc.

```
desc 'Generate the gosu.java.jar'
task :jar => [CLASSES_DIR] + FileList[JAVA_SOURCES] do
  sh "javac -source 1.6 -target 1.6
-bootclasspath \"#{ENV['ANDROID_HOME']}/platforms/android-10/android.jar\" -d
#{CLASSES_DIR} #{JAVA_SOURCES}"
  sh "jar cf lib/gosu.java.jar -C #{CLASSES_DIR} gosu"
end


desc 'Build the gem'
task :gem => :jar do
  sh 'gem build gosu_android.gemspec'
  sh 'mv gosu_android-*.gem pkg/'
end
```

La tarea **rake jar** compila todos los archivos java y substituye el el antiguo fichero gosu.java.jar con uno nuevo. Mientras que la tarea **rake gem** crea un nuevo archivo .gem en la carpeta pkg a partir de las especificaciones indicadas en el archivo .gemspec.

README.md

Este archivo contiene la información que se verá en la página de inicio del proyecto github. Usa el lenguaje de maquetado de github para el formato de la página.

```
Gosu-Android
============
A [Gosu](http://www.libgosu.org/) implementation for Android devices.

Installation
-----------

Install [ruboto](https://github.com/ruboto/ruboto/) and create a project.

### Recomended
```

```
With this installation method you will have a clean enviroment (bundler) to make
your gosu games.

* Create a file named `Gemfile.apk` in your ruboto project and add the lines:

```ruby
source "http://rubygems.org"
gem 'gosu_android'
```
```

Con este código se consigue el siguiente resultado en la página web:

Gosu-Android

A Gosu implementation for Android devices.

Installation

Install ruboto and create a project.

Recomended

With this installation method you will have a clean enviroment (bundler) to make your gosu games.

- Create a file named `Gemfile.apk` in your ruboto project and add the lines:

```
source "http://rubygems.org"
gem 'gosu_android'
```

- Create a folder inside `res` called `drawable-nodpi` and copy this file in it. On linux you can do it easily with: `mkdir res/drawable-nodpi`

  `wget https://raw.github.com/neochuky/gosu-android/master/res/drawable-nodpi/character_atlas8.png`

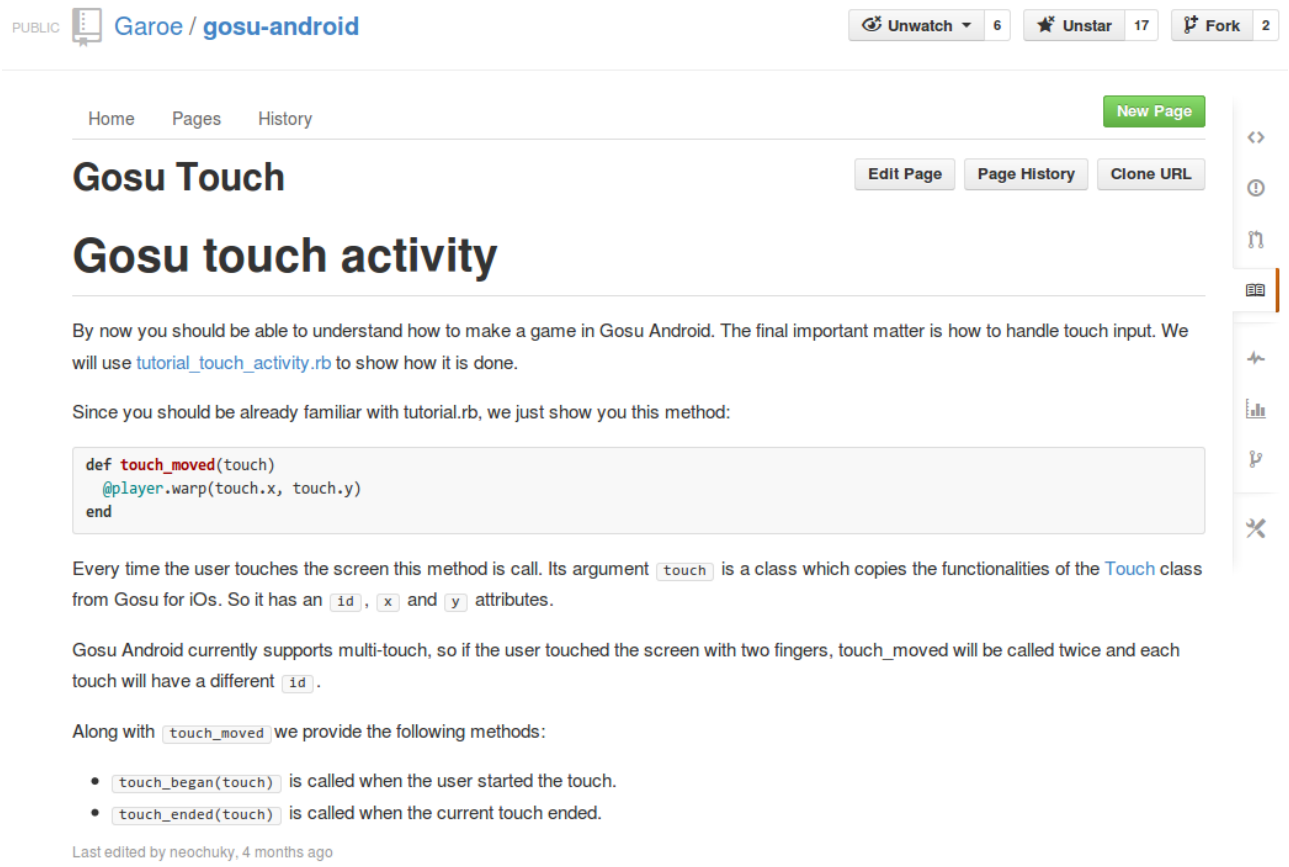
  `mv character_atlas8.png res/drawable-nodpi`

Alternative

With this installation method you will have a dirty enviroment (simple file copy) to try the samples.

- `gem install gosu_android`
- As with ruboto, place yourself in the root directory of your app.
- Then execute `gosu_android -a` or `gosu_android --add` to automatically copy every gosu_android file to your ruboto project. It will also copy all the media files that are use in gosu_android examples.

Para ayudar al usuario del motor que lo usa por primera vez, se añadió una wiki que explica los conceptos clave a la hora de usar el motor, siguiendo varios ejemplos en orden ascendente de dificultad:



PUBLIC Garoe / gosu-android

Unwatch 6 Unstar 17 Fork 2

Home Pages History New Page

Gosu Touch

Edit Page Page History Clone URL

Gosu touch activity

By now you should be able to understand how to make a game in Gosu Android. The final important matter is how to handle touch input. We will use tutorial_touch_activity.rb to show how it is done.

Since you should be already familiar with tutorial.rb, we just show you this method:

```
def touch_moved(touch)
  @player.warp(touch.x, touch.y)
end
```

Every time the user touches the screen this method is call. Its argument `touch` is a class which copies the functionalities of the Touch class from Gosu for iOs. So it has an `id`, `x` and `y` attributes.

Gosu Android currently supports multi-touch, so if the user touched the screen with two fingers, touch_moved will be called twice and each touch will have a different `id`.

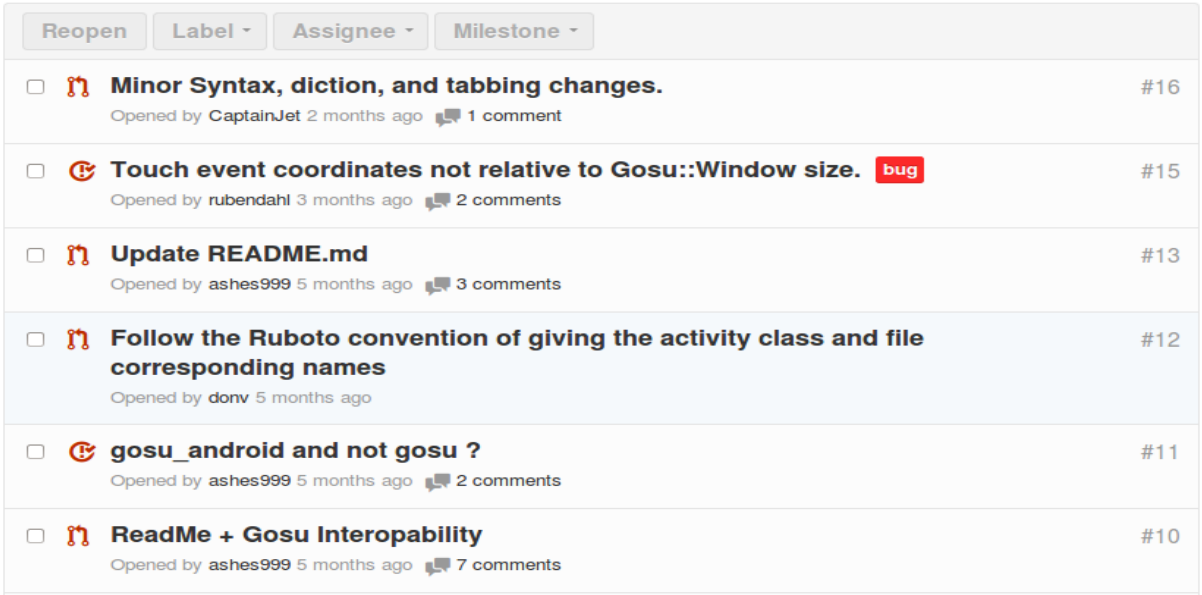
Along with `touch_moved` we provide the following methods:

- `touch_began(touch)` is called when the user started the touch.
- `touch_ended(touch)` is called when the current touch ended.

Last edited by neochuky, 4 months ago

La wiki completa se puede consultar en al anexo que se encuentra al final de este documento.

También se corrigieron varios issues abiertos por usuarios que descargaron el motor. A continuación se pueden ver varios issues resueltos:



Reopen Label Assignee Milestone

Minor Syntax, diction, and tabbing changes. Opened by CaptainJet 2 months ago 1 comment		#16
Touch event coordinates not relative to Gosu::Window size. bug Opened by rubendahl 3 months ago 2 comments		#15
Update README.md Opened by ashes999 5 months ago 3 comments		#13
Follow the Ruboto convention of giving the activity class and file corresponding names Opened by donv 5 months ago		#12
gosu_android and not gosu ? Opened by ashes999 5 months ago 2 comments		#11
ReadMe + Gosu Interopability Opened by ashes999 5 months ago 7 comments		#10

Problemas principales resueltos e información aprendida

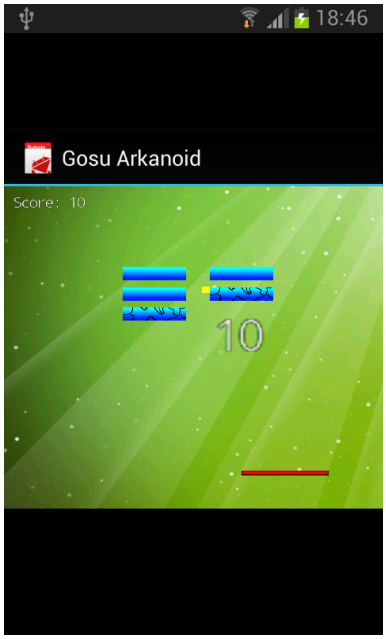
Para poder probar la gema antes de subirla a los servidores de RubyGems el método usado es simular un servidor en una carpeta del sistema. Por tanto se crea la gema y se crea el nuevo repositorio. Posteriormente se configura dicho repositorio para ser usado por defecto en el sistema y se procede a instalar la gema desde el y comprobar si funciona correctamente.

4.1.5.1. Juiciness

Los últimos cambios que se realizaron en los ejemplos del motor, se realizaron siguiendo el concepto de Juiciness. Este concepto surge en la convención Nordic Game Indie Night 2012, en ella el desarrollador Petri Purho empleó este termino para designar al conjunto de elementos que ayudan al feedback auditivo o visual del jugador.

Tal como se explica en la charla, es posible mejorar mucho un videojuego sin cambiar su mecánica, en su lugar se añaden animaciones, nuevos sonidos o modifican las ya existentes.

Teniendo esto en cuenta se añadió una animación de rebote, animación de puntuación al romper un bloque y nuevo sonido de vida perdida en el ejemplo Arkanoid2.



En estas imágenes se muestra el antes y el después del juego de ejemplo Arkanoid2. Tal como se observa, al añadir unas cuantas texturas la calidad aparente de la aplicación mejora considerablemente.

4.1.6. Versión en desarrollo

Aunque queda fuera de los objetivos del proyecto, debido al interés mostrado por la comunidad se toma la decisión de continuar con el desarrollo del motor. La versión 0.5 es la versión estable, que se puede descargar desde RubyGems. Pero en Github se encuentra también la versión en desarrollo. Que al momento de escribir esta memoria, tiene como objetivos la integración con Travis-CI, la corrección de nuevos bugs que han sido reportados y la implementación de nuevas características.

4.2. Chipmunk Android

Con toda la experiencia adquirida en desarrollar Gosu-Android se procedió a desarrollar Chipmunk-Android. Por tanto, desde un principio se decidió crear todas las clases internamente en Java, mientras que se exportan como una gema en Ruby. La estructura de ficheros también fue creada desde un principio para cumplir con los estándares de RubyGems.

Sin embargo, debido a las limitaciones temporales inherentes a un proyecto académico, no pudieron sino implementarse las clases básicas. En concreto están la clase Space, así como la clase Body y varios tipos de Shapes. Por tanto la funcionalidad a la que el usuario tiene acceso consiste en crearse bodies con atributos de posición y velocidad. A los cuales se les pueden añadir shapes de tipo segmento, circulo o cuadrado. Que tienen métodos de detección de colisiones entre ellos.

Cabe resaltar el código para agregar métodos cuando se produce una colisión, su uso es el siguiente:

```
@space.add_collision_func(:ball, :block) do |ball_shape, block_shape|
  #Codigo definido por el usuario
end
```

Como se puede observar, se llama al método con dos parámetros que indican el tipo de Shape que se van a colisionar. Y un tercer parámetro que es un método, en donde se le pasarán los objetos que han colisionado.

Para implementar esto se usa el código que se presenta a continuación:

```
public void add_collision_func( Object type0, Object type1,
                                ChipmunkRunnable function){
 function_list.add(function);
 add_collision_func_internal(type0, type1);
}
```

Dado que los tipos de los Shapes se definen mediante etiquetas en Ruby y este tipo no tiene traducción directa a Java, se toman los tipos como objetos base en Java: **Object**. Mientras que la

función a la que se ha de llamar es de tipo **ChipmunkRunnable**, que es una subclase de **Runnable** de Java:

```
public interface ChipmunkRunnable extends Runnable {
  public void run( Shape shape0, Shape shape1 );
}
```

Esto es necesario debido a que la clase Runnable tiene el método run sin parámetros y en nuestro caso necesitamos que tenga como parámetros los Shapes que han colisionado.

También merece la pena destacar el código del bucle principal de Chipmunk, pues es bastante sencillo:

```
    //Detect collisions
    int i = 0;
    //Get a pair of types we must check
    for( Object[] collision_pair: collisions_to_do ){
      //Get a shape of each type and check collision
      for( Shape shape0: objects_by_collision_type.get(collision_pair[0])){
        for( Shape shape1: objects_by_collision_type.get(collision_pair[1])){
          if(shape0 != shape1 && shape0.collides( shape1 )){
            function_list.get(i).run(shape0, shape1);
          }
        }
      }
      i++;
    }

    //Move objects
    for(Body body: body_list){
      body.step(dt);
    }
```

En **collisions_to_do** tenemos una **lista** con de pares de tipos de Shapes, que pueden colisionar. Mientras que **objects_by_collision_type** es un **hash** cuyas **claves** son tipos de Shapes y el **valor** es una lista con todos los Shapes que corresponden a dicho tipo. Por tanto, se tienen fácilmente todas las combinaciones de tipos para los que hay que comprobar si se producen colisiones.

El porcentaje de código escrito en Java vs Ruby es el siguiente:

4.3. Interacción con la comunidad

La comunidad de software libre mostró interés por el desarrollo de este proyecto. Debido a esto se estableció una relación de simbiosis con varios desarrolladores. Por lo que, la comunidad contribuyó al proyecto y además se implementó código ajeno al motor a petición de los responsables de Ruboto.

4.3.1. Código de la comunidad

A continuación se detalla por usuario los aportes al motor:

Donv

Entre otros, este usuario creó un equivalente a la clase **GLSurfaceView** en Java, esto resolvía errores que se producían al usar Gosu-Android en Bluestacks, debido a que BlueStacks emula Android 2.3 y se quedaba sin memoria, pero al crear la clase en Java se resuelve este problema.

CaptainJet

Corrección de erratas y recodificación de los recursos de vídeo, para que las imágenes ocupen menos espacio en disco.

Ashes999

Corrección de erratas.

4.3.2. Código para la comunidad

Debido a que Ruboto es un desarrollo muy reciente y algo inestable. No tiene todavía mucha popularidad, debido a esto nos encontramos con que Gosu-Android es la primera gema que se desarrolla para uso exclusivo en Android. Es decir, puesto que depende de librerías Android, no se puede ejecutar en ninguna otra plataforma, y por tanto es una gema que solo se ha de poder instalar en Android. Al ser la primera gema de este tipo, se acordó con los programadores de Ruboto, que lo mejor sería incluir una nueva plataforma para el parámetro platform en el archivo gemfile.gemspec, y así poder resolver adecuadamente donde se puede instalar y donde no se permitirá.

Esto conlleva dos modificaciones:

- Incluir el código necesario en el proyecto rubygems para añadir la nueva plataforma.
- Incluir el código necesario para incluir la opción platform al instalar una gema con **Bundler**.

De esta manera podemos indicar que, aunque estamos en un PC vamos a instalar una gema para Android en nuestro proyecto a través de Ruboto. Para ello no solo hubo que incluir la nueva plataforma Android en el código de bundler, si no también incluir la opción platform. Así como añadir los tests necesarios debido a que se están añadiendo nuevas características.

La primera parte de la modificación, fue resuelta por Donv. Mientras que la segunda parte era responsabilidad nuestra.

En primer lugar se estudió el código de Bundler con detalle para entender como funcionaba. Una vez se tuvo una idea básica, se fueron realizando pequeñas modificaciones hasta que se logró instalar la gema de ejemplo example_gem[34] en un proyecto Ruboto. Una vez hecho esto, se procedió a crear los tests necesarios para el testeo automático que se realiza en Bundler. Para ello hubo que aprender como funciona Travis-CI. Al momento de escribir esta memoria se ha hecho un pull request a Bundler con las modificaciones aunque este todavía no ha sido aceptado.

Se muestra a continuación las partes más relevantes del código que se añadió a Bundler. En primer lugar, añadir la nueva opción en el código que define las llamadas posibles:

```
    method_option "platform", :type => :string, :banner =>
      "Force the use of the specified platform instead of runtime detected
platform"


    if options[:platform]
        Bundler.settings[:platform] = options[:platform]
        unless Dependency.gem_platform(options[:platform].to_sym)
          raise InvalidOption, "Unknown platform, available platforms are " \
                               "#{(Dependency::PLATFORM_MAP.keys.collect {|p|
                                                          p.to_s}).inspect}"
        end
      end
```

Nuevo método para comprobar con la plataforma escogida, en lugar de con la del sistema:

```
    #Match platform used when --platform is used
    def self.match_platform_option(p)
      platform = Dependency.gem_platform(Bundler.settings[:platform].to_sym)
      Gem::Platform::RUBY == p or
      p.nil? or p == platform or
      Gem::Platform.new(p) == platform
    end
```

Por último el código para tests de la nueva opción:

```
it "works with gems that have different dependencies" do

  gemfile <<-G
    source "file://#{gem_repo1}"

    gem "nokogiri"
  G

  bundle "install --platform jruby"
  should_be_installed "nokogiri 1.4.2 JAVA", "weakling 0.0.3"
end
```

5. Conclusiones y Trabajo futuro

A continuación se detallan las conclusiones y el trabajo futuro.

5.1. Conclusiones

Se ha logrado implementar el motor y que funcione adecuadamente, debido al carácter experimental del proyecto y las limitaciones impuestas por los dispositivos móviles, esto era una incógnita al comienzo del desarrollo. Además, estas restricciones se acentúan al no estar implementado con código nativo en Java, sino interpretado a través de Jruby.

Al ejecutar un interprete de Ruby (Jruby) sobre otro interprete Dalvik para Java, se generan graves problemas de eficiencia.

El colector de basura de java es problemático, debido a que se ejecuta cuando el sistema operativo lo considera conveniente e interrumpe el juego durante algunos milisegundos. Además el control que tenemos sobre el, desde Ruby es prácticamente nulo.

La comunidad participó activamente, al momento de escribir esta memoria el motor sobrepasa las dos mil descargas. A través de github varios usuarios han reportado bugs. Por otra parte, varios han contribuido al proyecto con mejoras de código. El número de descargas previo es del empaquetado desde RubyGems, pues Github no informa de cuantas descargas se realizan desde su plataforma. En cuanto al repositorio, Gosu-Android tiene 3 forks en activo, 16 stars y 5 watchs (las stars son usuarios interesados en el repositorio y los watchs son usuarios que quiere estar al tanto de cualquier cambio que haya ocurrido en el repositorio).

Se ha logrado comprender como funcionan los grandes proyectos de software libre con integración continua (Travis-CI) de pruebas automáticas.

Poco después de terminar con Chipmunk-Android, los desarrolladores del motor Chipmunk sacaron al mercado un plugin para Unity, que permite integrarlo en cualquier proyecto que se realice usando este SDK. Este plugin permite su uso para plataformas Android, por lo que continuar con el desarrollo de Chipmunk-Android se torna innecesario. Sin embargo, el código que se desarrolló no fue en balde, ya que la experiencia adquirida es muy valiosa.

Aunque el motor funciona, sigue estando muy comprometido en cuanto a eficiencia, especialmente si se compara con otros. Sin embargo se ha de tener en cuenta que muchos programadores usan Ruby como medio para probar ideas y diseños, que una vez han sido validadas en Ruby se implementan en otros lenguajes, por lo que los problemas que se han encontrado pueden pasar a un segundo plano. Y por tanto, se está proporcionando una herramienta que cubre necesidades que ninguna otra cubre.

5.2. Trabajo futuro

Entre las mejoras que todavía quedan por realizar en Gosu-Android se pueden destacar las siguientes:

- Terminar de reescribir todo el código a Java. En caso necesario pasar a C++ para una mejor eficiencia.
- Terminar de implementar la API de Gosu, puesto que no está completa.
- Terminar la integración con Travis-CI, además de crear buenos casos de pruebas.
- Arreglar los bugs que han sido reportados y no han sido corregidos aún.
- Añadir más ejemplos.
- Integrar el código con Gosu y que se realice la instalación de la parte correspondiente según la plataforma de destino.
- Incluir soporte para ejecutar OpenGL 2.0 y 3.0 si el dispositivo final lo soporta.

6. Bibliografía

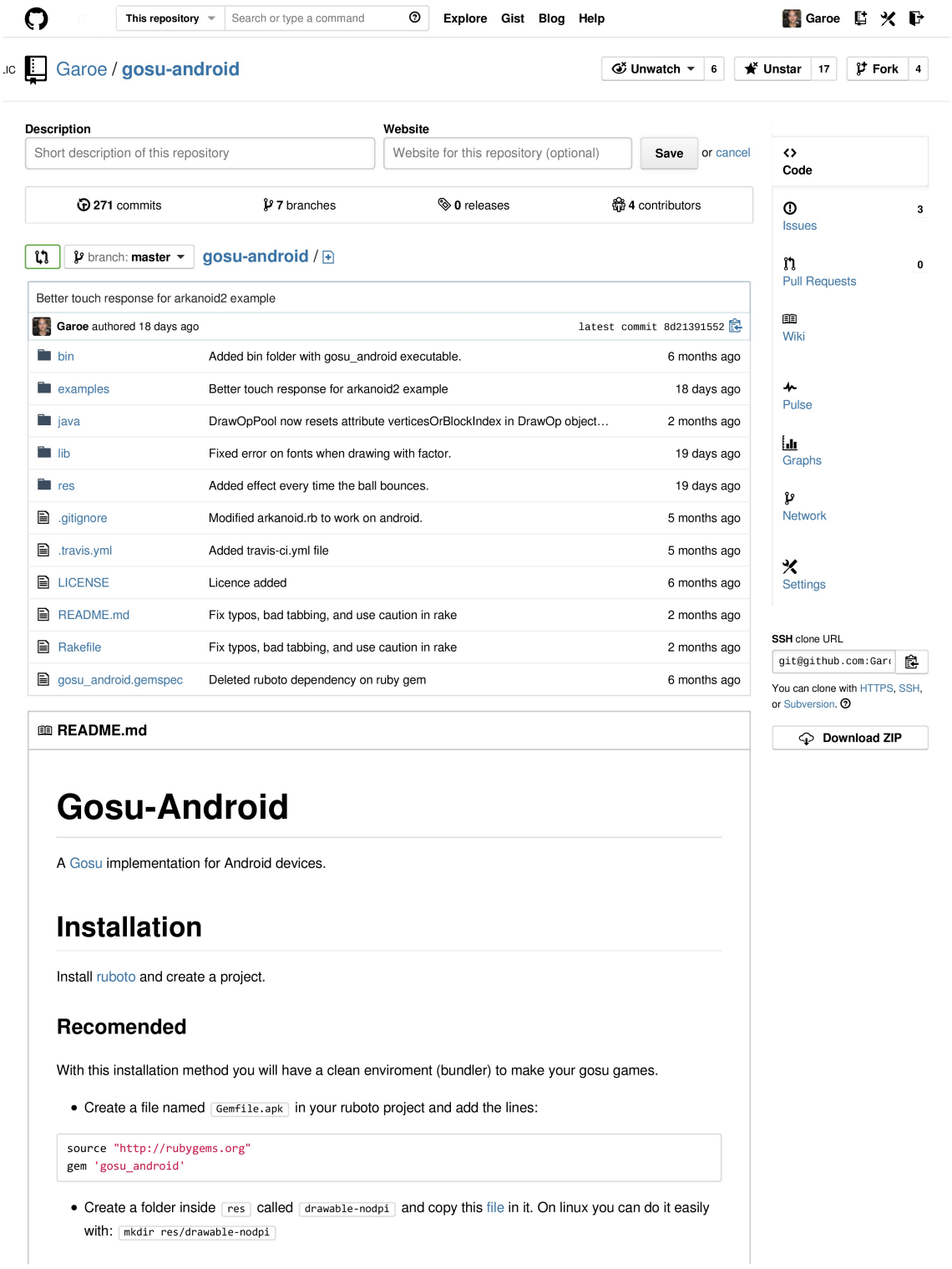
[0] http://en.wikipedia.org/wiki/Cathode_ray_tube_a_musement_device
[1] http://es.blackberry.com/
[2] http://www.windowsphone.com/es-es
[3] http://www.apple.com/iphone/
[4] http://www.android.com/
[5] http://mon-ouie.github.io/projects/ray.html
[6] http://g3d-ruby.rubyforge.org/
[7] http://www.andengine.org
[8] http://www.libgosu.org/
[9] http://ippa.se/chingu
[10] https://love2d.org/
[11] http://chipmunk-physics.net/
[12] http://www.havok.com/products/physics
[13] www.unity3d.com
[14] http://irrlicht.sourceforge.net/
[15] https://www.panda3d.org/
[16] https://groups.google.com/forum/#topic/ruboto/qhj2orCZH4c
[17] http://git-scm.com/
[18] https://github.com/
[19] http://www.eclipse.org/
[20] http://www.aptana.com/
[21] http://www.geany.org/
[22] https://projects.gnome.org/gedit/
[23] http://www.bluestacks.com/
[24] http://developer.android.com/sdk/index.html
[25] https://www.ruby-lang.org/es/
[26] http://jruby.org/
[27] http://rubini.us/
[28] http://rubygems.org/
[29] https://rvm.io/
[30] http://ruboto.org/
[31] http://bundler.io/
[32] https://travis-ci.org/
[33] https://github.com/ruboto/ruboto/wiki/Generating-classes-for-callbacks
[34] https://github.com/ruboto/example_gem

7. Anexo

7.1. Wiki

A continuación se muestra el contenido completo de la wiki que se hizo para Gosu Android. Esta wiki es junto con la wiki de Gosu el manual de usuario para el motor.

Garoe/gosu-android https://github.com/Garoe/gosu-android



```
wget https://raw.github.com/neochuky/gosu-android/master/res/drawable-nodpi/character_atlas8.png
mv character_atlas8.png res/drawable-nodpi
```

Alternative

With this installation method you will have a dirty enviroment (simple file copy) to try the samples.

- `gem install gosu_android`
- As with ruboto, place yourself in the root directory of your app.
- Then execute `gosu_android -a` or `gosu_android --add` to automatically copy every gosu_android file to your ruboto project. It will also copy all the media files that are use in gosu_android examples.

General Information

- This is still an early effort, so there are a number of features that had not yet been added.
- There are some known bugs that I hope to fix soon.
- In its current status there are some small changes to Gosu Window initialization, check examples with many comments in the wiki.

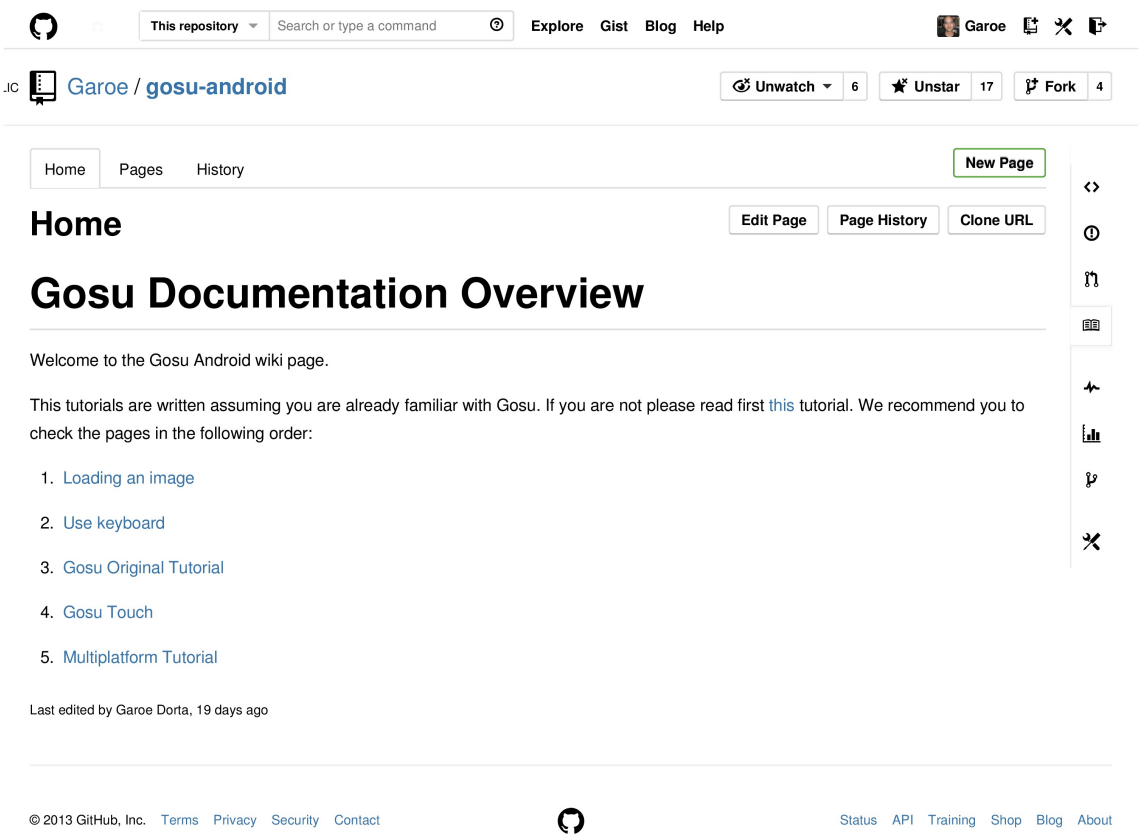
Troubleshooting

- When using several audio files double check that all have the same codification or you could get:
  `E/MediaPlayer(16127): Unable to to create media player`
  - http://forums.pragprog.com/forums/152/topics/9144

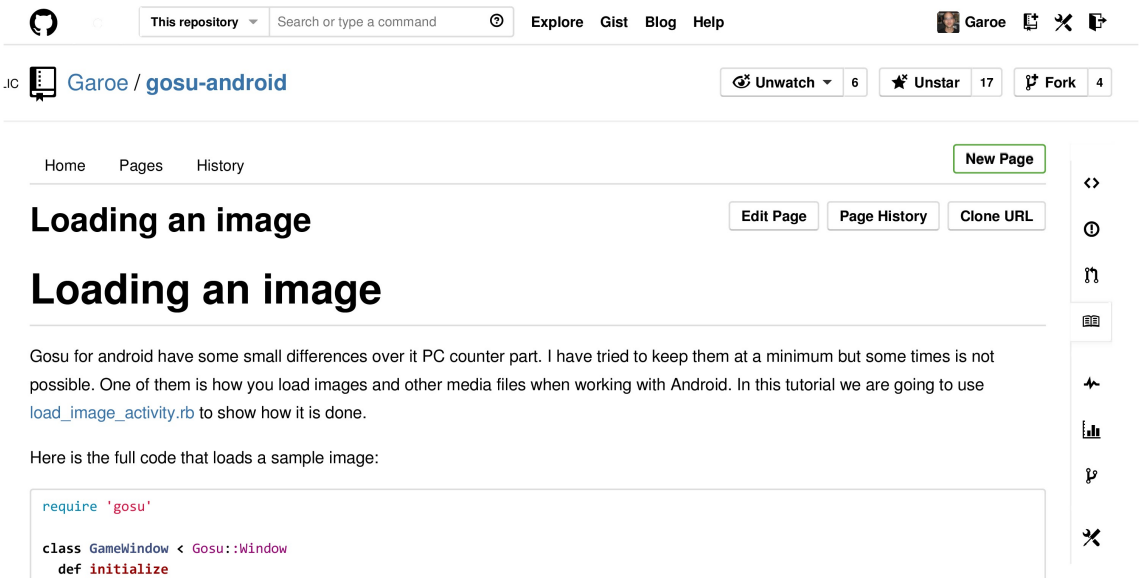
© 2013 GitHub, Inc. Terms Privacy Security Contact

Status API Training Shop Blog About

Home · Garoe/gosu-android Wiki https://github.com/Garoe/gosu-android



Loading an image · Garoe/gosu-android Wiki https://github.com/Garoe/gosu-android/wiki/Loading-an-image



Loading an image

Gosu for android have some small differences over it PC counter part. I have tried to keep them at a minimum but some times is not possible. One of them is how you load images and other media files when working with Android. In this tutorial we are going to use load_image_activity.rb to show how it is done.

Here is the full code that loads a sample image:

```
require 'gosu'

class GameWindow < Gosu::Window
  def initialize
    #Creates a window of 600 by 400, not fullscreen, at 30 fps
    super 600, 480, false, 30
    #Window title
    self.caption = "Gosu Load Image"
    #Load image from drawable folder
    @image = Gosu::Image.new(self, Ruboto::R::drawable::ship)
  end

  def draw
    #Image top left corner will be on top left corner of the screen
    @image.draw(0, 0, 0)
  end
end

#Initialize gosu for android
class LoadImageActivity
  def on_create(bundle)
    super(bundle)
    Gosu::AndroidInitializer.instance.start(self)
    rescue Exception => e
      puts "#{ e } (#{ e.class } #{e.message} #{e.backtrace.inspect} )!"
  end

  def on_ready
    window = GameWindow.new
    window.show
    rescue Exception => e
      puts "#{ e } (#{ e.class } #{e.message} #{e.backtrace.inspect} )!"
  end
end
```

Everything should look familiar to a Gosu user but line 7,

```
@image = Gosu::Image.new(self, Ruboto::R::drawable::ship)
```

which we will explain later. And of course the class `LoadImageActivity`, this class is necessary to start an android application, but all that matters are the lines that load the `Gosu::Window`

```
  window = GameWindow.new
  window.show
```

Before explaining line 7 lest first learn some notions about android resources. Android wants all media files to be store in a folder named

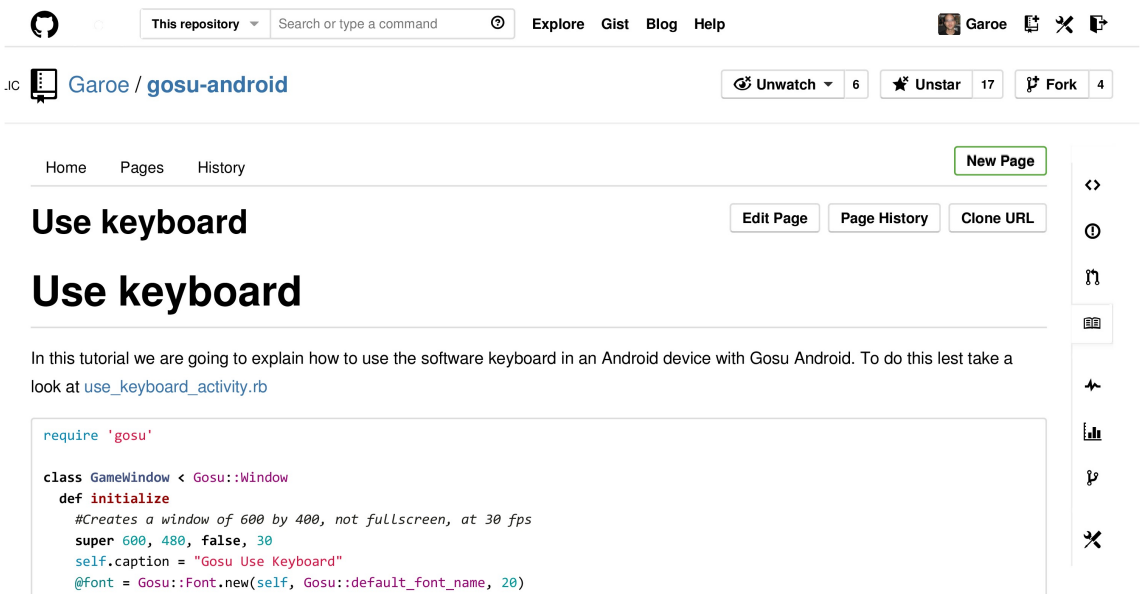
`res` in your project. Not only inside `res` but on a different folder inside `res` according to its type. For instance audio files go in the `raw` folder inside `res`. While the pictures go in the `drawable` folder. There are some terminations on drawable like: hdpi, nodpi, etc. This indicates images for large displays, small displays, and so on.

So the line `Ruboto::R::drawable::ship` means access the file `ship` which is located is some drawable folder.

Finally there are some considerations that you should bear in mind when using images in Gosu Android. Firstly, all file names must be in lower case.
Secondly, images for animations must be place in `drawable-nodpi`. While images for static display can be place in whichever your prefer.

Last edited by Garoe, 3 months ago

© 2013 GitHub, Inc. Terms Privacy Security Contact Status API Training Shop Blog About



Use keyboard

In this tutorial we are going to explain how to use the software keyboard in an Android device with Gosu Android. To do this lest take a look at use_keyboard_activity.rb

```
require 'gosu'

class GameWindow < Gosu::Window
  def initialize
    #Creates a window of 600 by 400, not fullscreen, at 30 fps
    super 600, 480, false, 30
    self.caption = "Gosu Use Keyboard"
    @font = Gosu::Font.new(self, Gosu::default_font_name, 20)
    #Bring up the keyboard
    self.show_soft_keyboard
  end

  def button_down(id)
    #Save the key value
    @key = id
  end

  def draw
    #Show which key the user pressed
    @font.draw("Press a key to see its code: #{@key}", 10, 10, 3, 1.0, 1.0, 0xffffff00)
  end
end

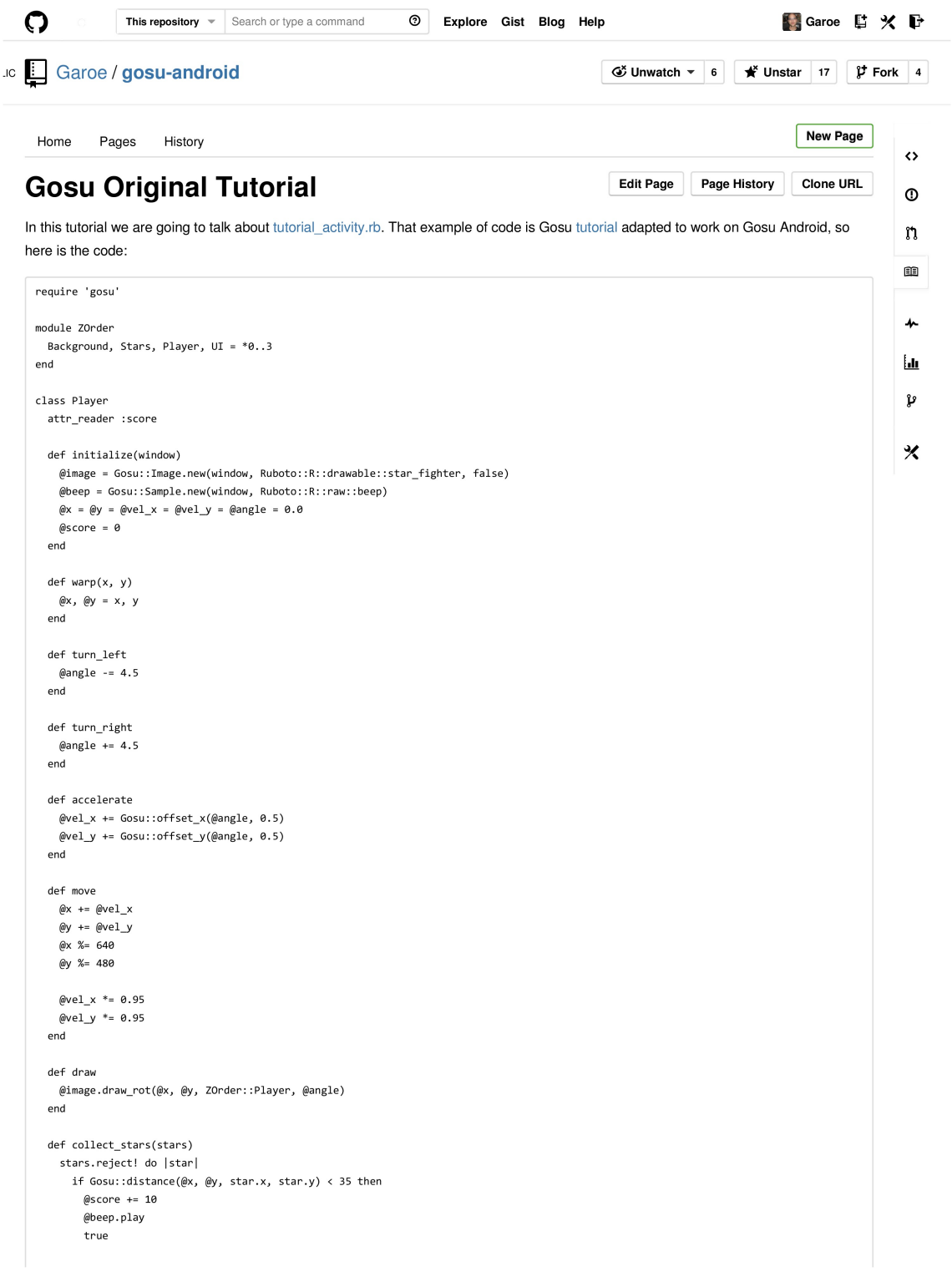
#Initialize gosu for android
class UseKeyboardActivity
  def on_create(bundle)
    super(bundle)
    Gosu::AndroidInitializer.instance.start(self)
    rescue Exception => e
      puts "#{ e } (#{ e.class } #{e.message} #{e.backtrace.inspect} )!"
  end

  def on_ready
    window = GameWindow.new
    window.show
    rescue Exception => e
      puts "#{ e } (#{ e.class } #{e.message} #{e.backtrace.inspect} )!"
  end
end
```

We see here a simple activity that creates a Window. On the Window initialization we got the line `self.show_soft_keyboard` this will bring up the software keyboard on an Android device that lacks a physical one.

Once you are done typing we provide the method `self.hide_soft_keyboard` this method will make the software keyboard disappear.

Last edited by Garoe, 3 months ago


This repository
Search or type a command
Explore
Gist
Blog
Help
Garoe
Garoe / gosu-android
Unwatch 6
Unstar 17
Fork 4
Home
Pages
History
New Page
Gosu Original Tutorial
Edit Page
Page History
Clone URL
In this tutorial we are going to talk about tutorial_activity.rb. That example of code is Gosu tutorial adapted to work on Gosu Android, so here is the code:
require 'gosu'
module ZOrder
Background, Stars, Player, UI = *0..3
end
class Player
attr_reader :score
def initialize(window)
@image = Gosu::Image.new(window, Ruboto::R::drawable::star_fighter, false)
@beep = Gosu::Sample.new(window, Ruboto::R::raw::beep)
@x = @y = @vel_x = @vel_y = @angle = 0.0
@score = 0
end
def warp(x, y)
@x, @y = x, y
end
def turn_left
@angle -= 4.5
end
def turn_right
@angle += 4.5
end
def accelerate
@vel_x += Gosu::offset_x(@angle, 0.5)
@vel_y += Gosu::offset_y(@angle, 0.5)
end
def move
@x += @vel_x
@y += @vel_y
@x %= 640
@y %= 480
@vel_x *= 0.95
@vel_y *= 0.95
end
def draw
@image.draw_rot(@x, @y, ZOrder::Player, @angle)
end
def collect_stars(stars)
stars.reject! do |star|
if Gosu::distance(@x, @y, star.x, star.y) < 35 then
@score += 10
@beep.play
true

```
      else
        false
      end
    end
  end
end

class Star
  attr_reader :x, :y

  def initialize(animation)
    @animation = animation
    @color = Gosu::Color.new(0xff000000)
    @color.red = rand(256 - 40) + 40
    @color.green = rand(256 - 40) + 40
    @color.blue = rand(256 - 40) + 40
    @x = rand * 640
    @y = rand * 480
  end

  def draw
    img = @animation[Gosu::milliseconds / 100 % @animation.size]
    img.draw(@x - img.width / 2.0, @y - img.height / 2.0,
        ZOrder::Stars, 1, 1, @color, :add)
  end
end

class GameWindow < Gosu::Window
  def initialize
    super(640, 480, false)
    self.caption = "Gosu Tutorial Game"

    @background_image = Gosu::Image.new(self, Ruboto::R::drawable::space, true)

    @player = Player.new(self)
    @player.warp(320, 240)

    @star_anim = Gosu::Image::load_tiles(self, Ruboto::R::drawable::star, 25, 25, false)
    @stars = Array.new

    @font = Gosu::Font.new(self, Gosu::default_font_name, 20)
  end

  def update
    if button_down? Gosu::KbA then
      @player.turn_left
    end
    if button_down? Gosu::KbD then
      @player.turn_right
    end
    if button_down? Gosu::KbW then
      @player.accelerate
    end
    @player.move
    @player.collect_stars(@stars)

    if rand(100) < 4 and @stars.size < 25 then
      @stars.push(Star.new(@star_anim))
    end
  end

  def draw
    @background_image.draw(0, 0, ZOrder::Background)
    @player.draw
    @stars.each { |star| star.draw }
    @font.draw("Score: #{@player.score}", 10, 10, ZOrder::UI, 1.0, 1.0, 0xffffff00)
  end

  def button_down(id)
```

```
    if id == Gosu::KbEscape then
      close
    end
  end
end

class TutorialActivity
  def on_create(bundle)
    super(bundle)
    Gosu::AndroidInitializer.instance.start(self)
    rescue Exception => e
      puts "#{ e } (#{ e.class } #{e.message} #{e.backtrace.inspect} )!"
  end

  def on_ready
    window = GameWindow.new
    window.show
    window.show_soft_keyboard
    rescue Exception => e
      puts "#{ e } (#{ e.class } #{e.message} #{e.backtrace.inspect} )!"
  end
end
```

To begin with there are the differences is loading media files like:

```
  @image = Gosu::Image.new(window, Ruboto::R::drawable::star_fighter, false)
  @beep = Gosu::Sample.new(window, Ruboto::R::raw::beep)
```

This is treated in Loading an image wiki page so please read it if you have not already.

Also there is a huge code where the `Gosu::Window` is created:

```
class TutorialActivity
  def on_create(bundle)
    super(bundle)
    Gosu::AndroidInitializer.instance.start(self)
    rescue Exception => e
      puts "#{ e } (#{ e.class } #{e.message} #{e.backtrace.inspect} )!"
  end

  def on_ready
    window = GameWindow.new
    window.show
    window.show_soft_keyboard
    rescue Exception => e
      puts "#{ e } (#{ e.class } #{e.message} #{e.backtrace.inspect} )!"
  end
end
```

Android requires a main Activity to execute code and even here he must comply. So we create a class which name must correspond to name in the package argument of ruboto gen app, in our case: `ruboto gen app --package org.tutorial --target android-10`.

Gosu Android needs some internal stuff to be initialize before creating a window and that is what `Gosu::AndroidInitializer.instance.start(self)` does. When the initializer is ready it will call `on_ready` where our `Window` will be created.

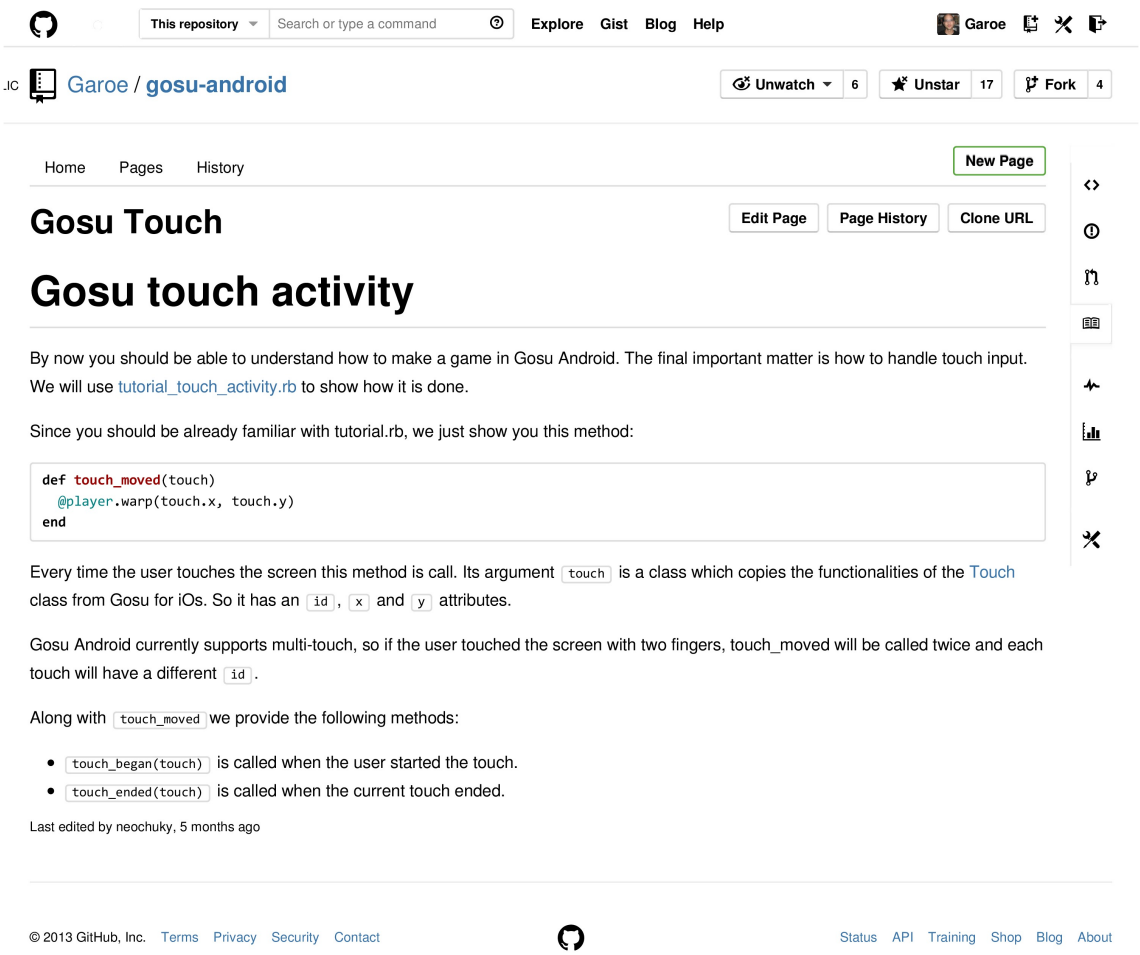
Lastly we call `window.show_soft_keyboard` which is discussed in Use keyboard.

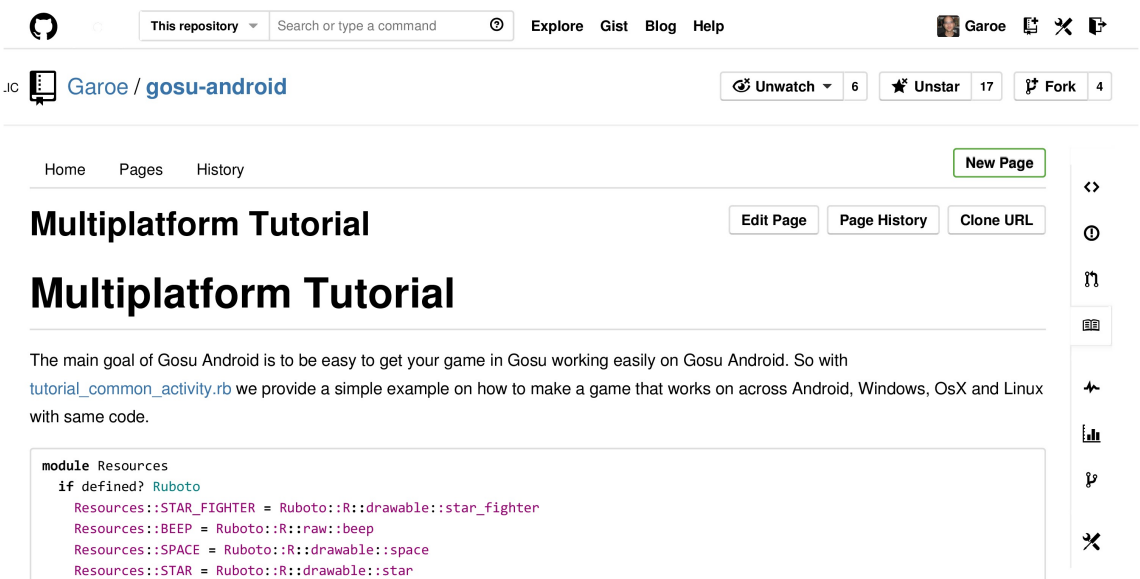
Last edited by neochuky, 5 months ago

© 2013 GitHub, Inc. Terms Privacy Security Contact

Status API Training Shop Blog About

Gosu Touch · Garoe/gosu-android Wiki https://github.com/Garoe/gosu-android/wiki/Gosu-Touch





Multiplatform Tutorial

The main goal of Gosu Android is to be easy to get your game in Gosu working easily on Gosu Android. So with tutorial_common_activity.rb we provide a simple example on how to make a game that works on across Android, Windows, OsX and Linux with same code.

```
module Resources
  if defined? Ruboto
    Resources::STAR_FIGHTER = Ruboto::R::drawable::star_fighter
    Resources::BEEP = Ruboto::R::raw::beep
    Resources::SPACE = Ruboto::R::drawable::space
    Resources::STAR = Ruboto::R::drawable::star
  else
    Resources::STAR_FIGHTER = "media/Starfighter.bmp"
    Resources::BEEP = "media/Beep.wav"
    Resources::SPACE = "media/Space.png"
    Resources::STAR = "media/Star.png"
  end
end
```

The first thing is to define the adequate routes to the media files according to the current platform. So for example when loading an image we do `@image = Gosu::Image.new(window, Resources::STAR_FIGHTER, false)`

For the PC we got the input control for the keyboard:

```
def update
  if button_down? Gosu::KbA then
    @player.turn_left
  end
  if button_down? Gosu::KbD then
    @player.turn_right
  end
  if button_down? Gosu::KbW then
    @player.accelerate
  end
  @player.move
  @player.collect_stars(@stars)

  if rand(100) < 4 and @stars.size < 25 then
    @stars.push(Star.new(@star_anim))
  end
end
```

While for Android we got the touch input:

```
def touch_moved(touch)
  @player.warp(touch.x, touch.y)
end
```

We know that the touch method will never be called on PC.

Finally we define how the window is created for each platform:

```
if not defined? Ruboto
  window = GameWindow.new
  window.show
else
  class TutorialCommonActivity
    def on_create(bundle)
      super(bundle)
      Gosu::AndroidInitializer.instance.start(self)
      rescue Exception => e
        puts "#{ e } (#{ e.class } #{e.message} #{e.backtrace.inspect} )!"
    end

    def on_ready
      window = GameWindow.new
      window.show
      rescue Exception => e
        puts "#{ e } (#{ e.class } #{e.message} #{e.backtrace.inspect} )!"
    end
  end
end
```

Last edited by neochuky, 5 months ago

© 2013 GitHub, Inc. Terms Privacy Security Contact

Status API Training Shop Blog About