PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

ESCUELA DE POSGRADO

KILLA, UN LENGUAJE DE PROGRAMACION BASADO EN LUA Y JAVASCRIPT

Tesis para optar por el Título de Magíster en Ciencias de la Computación

Laurens Isaac Rodríguez Oscanoa

ASESOR: Ph.D Jose Antonio Pow-Sang Portillo

JURADO

PRESIDENTE:	**Héctor Andrés Melgar Sasieta**
SEGUNDO MIEMBRO:	**Jose Antonio Pow-Sang Portillo**
TERCER MIEMBRO:	**Johan Paul Baldeón Medrano**

Lima, Noviembre de 2012

Resumen

Este tesis trata acerca del diseño e implementación de un nuevo lenguaje llamado Killa el cual esta basado en la maquina virtual de Lua pero tiene una sintaxis basada en C y con soporte de características avanzadas de lenguajes modernos como JavaScript que soportan funciones de primer orden y closures. Se ha hecho un lenguaje más robusto y apropiado para desarrollar videojuegos medianos y grandes que Lua o JavaScript.

Killa proviene de la palabra quechua luna, lo cual es un tributo a Lua que a su vez significa luna en portugués, Killa es un descendiente directo de Lua y fue diseñado desde el principio para compartir la misma filosofía de Lua: ser ligero en cuanto a consumo de recursos, ser rápido en cuanto a velocidad de ejecución, ser fácil de incrustar en sistemas externos y ser libre y abierto para poder ser usado y modificado libremente sin ningún tipo de restricciones.

Killa hace todo lo que Lua puede hacer pero a su vez agrega mejoras importantes para evitar errores de programación comunes gracias a sus características nuevas como declaración de variables obligatoria y minimización del espacio global del programa haciendo la definición de símbolos privada por defecto. En programas escritos en Killa es posible usar variables globales pero a diferencia de otros lenguajes se puede saber exactamente donde están y cuales son con una simple búsqueda en los archivos del programa.

Killa también agrega mejoras en cuanto a la productividad y legibilidad de código al permitir construcciones familiares y comunes en muchos otros lenguajes populares como son operadores de asignación compuestos, expresiones ternarias y operadores binarios. Al usar índices de array que empiezan en cero en vez de índices que empiezan en uno como hace Lua, Killa evita muchos errores de cambio de contexto que comenten los programadores que tienen que usar Lua al mismo tiempo que otros lenguajes populares que usan índices de base cero. Un programador con experiencia en lenguajes populares como C++, Java, JavaScript o C# se sentirá en un entorno familiar y conocido cuando programe en Killa.

El propósito de Killa es el de ser usado en la industria de videojuegos. La implementación presentada al final de esta tesis es completamente funcional, Killa es bastante sólido pues está amparado por los casi 20 años de desarrollo que Lua trae consigo y del cual ha sido creado. Además Killa ha sido probado satisfactoriamente incrustándolo en motores de videojuegos de código abierto ampliamente conocidos y modernos como Cocos2d-x y Love2D.

Killa Corre en cualquier plataforma y sistema operativo. Se muestran ejemplos funcionando en sistemas de escritorio y dispositivos móviles.

Tabla de contenidos

Listado de Figuras

Listado de Tablas

Introducción

En la ultima década, el progreso en el desarrollo de lenguajes dinámicos ha tenido un buen avance debido a la percepción de que la productividad de los programadores se ve aumentada cuando se los libera de tareas de bajo nivel como preocuparse por crear bloques de memoria, estimar el tamaño de dichos bloques correctamente y liberarlos cuando ya no sean necesarios o decidir entre los diversos tipos de datos que el lenguaje permite usar y manejar correctamente las conversiones entre dichos tipos.

Los partidarios de los lenguajes dinámicos argumentan que los lenguajes estáticos son demasiado rígidos y que la elasticidad de los lenguajes dinámicos los hacen más adecuados para el desarrollo de prototipos de sistemas con requerimientos cambiantes o desconocidos o que interactúan con otros sistemas que actúan de modo impredecible [1].

Es por eso que lenguajes de programación como Perl, Python, Ruby y JavaScript han ido ganando popularidad y su uso es cada vez mayor. El crecimiento en el uso de los lenguajes dinámicos puede apreciarse en la siguiente grafica extraída de TIOBE [2]:

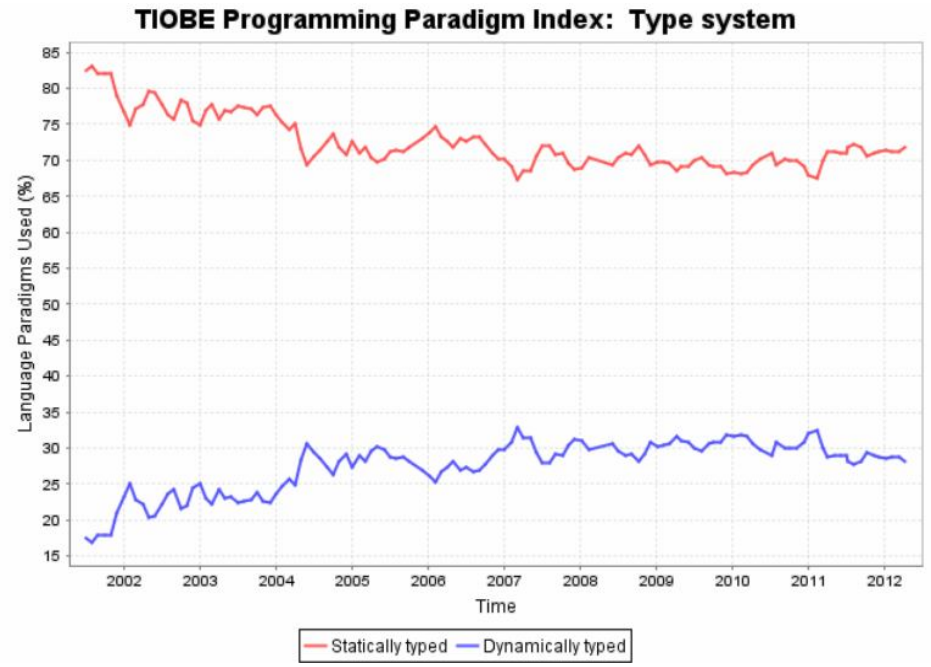

FIGURA 0.1: Uso de lenguajes estáticos y dinámicos en los últimos 10 años

Puede verse que los lenguajes dinámicos han ganado una aceptación de casi 30% en el 2012 frente a un 20% en el 2002.

Por otro lado existen muchas tareas críticas que requieren máxima eficiencia y acceso a los recursos del sistema operativo de la manera más directa posible, esa es la razón por la que aún existen programas escritos en lenguajes estáticos como C/C++. Sin embargo, a medida que dichos programas evolucionan y se vuelven mas complejos se hace mucho más difícil poder agregarles funcionalidad por su inherente complejidad y por los largos tiempos de desarrollo, compilación, pruebas y depuración que este tipo de programas requiere.

Esto ha forzado a buscar formas de poder agregar funcionalidad a dichos programas de alto rendimiento. Una solución es a través del uso de lenguajes de programación script para incrustación, lo que significa añadir a dichos programas monolíticos un intérprete de scripts con el cual se pueda acceder a parte de su funcionalidad y realizar tareas sin la necesidad de recompilar el programa, sino simplemente agregando scripts que son interpretados en tiempo de ejecución. De esa manera es posible unir lo mejor de ambos mundos, la velocidad y acceso total al sistema operativo que brinda C/C++ junto con la versatilidad y facilidad de desarrollo que ofrecen lenguajes dinámicos [3].

Lua [4] es uno de esos lenguajes de script para incrustación, el cual gracias a ser pequeño, potente y de licencia liberal MIT se ha consolidado como el lenguaje para incrustación por defecto de la industria, siendo su uso casi estándar y obligatorio en la industria de videojuegos moderna, juegos tan masivos como el World of Warcraft con mas de 10 millones de usuarios lo usa para poder personalizar su interfaz de usuario [5].

Esta tesis plantea la creación de un lenguaje para incrustación llamado Killa, el cual usa la maquina virtual de Lua y comparte la misma filosofía de Lua, ser pequeño en cuanto a uso de recursos consumidos, eficiente en cuanto a velocidad de compilación y ejecución, fácil de ser incrustado en sistemas complejos que necesiten soporte de script pero usando una sintaxis mas parecida a JavaScript, corrigiendo deficiencias de ambos lenguajes en el camino y brindando mejor soporte para el desarrollo de programas robustos y de tamaño medianamente grande.

1. Capítulo I: Definición del problema

El desarrollo de aplicaciones grandes y complejas (como es el caso de motores de videojuegos) en lenguajes estáticos como C++ es lento y complejo a medida de que el código fuente crece. En la industria de videojuegos no es raro tener motores con millones de líneas de código, decenas de dependencias en librerías externas usadas y mucho tiempo de compilación.

Como ejemplo considérese el motor Torque 3D que ha sido recientemente abierto bajo licencia MIT, a pesar de ser relativamente pequeño si se lo compara con otros motores de carácter comercial y mas usados (como es el Unreal Engine) aun así tiene mas de medio millón de líneas de código fuente. Sin contar el código del juego mismo. Se ve pues que es muy difícil realizar juegos completos modificando la base de código del motor mismo escrito en C++.

Es debido a esta razón que casi todos estos motores permiten extensibilidad a través de una capa de scripts que son cargados e interpretados en tiempo de ejecución. Por ejemplo:

- Torque 3D usa TorqueScript
- Word of Warcraft usa Lua
- Unity usa JavaScript, C# y Boo
- Unreal Engine usa UnrealScript

Una pregunta valida es porque la mayoría de motores de videojuegos no usan lenguajes de script más conocidos y con mayor numero de usuarios y documentación. El principal problema es que casi todos estos lenguajes consumen muchos recursos pues han sido creados para ser lenguajes de tipo general. Otro problema importante es que son difíciles de integrar dentro de los respectivos motores. Adicionalmente tienen bajo rendimiento, es decir son lentos, lo que es un grave problema para el desarrollo de videojuegos donde se espera el mayor rendimiento posible con el menor consumo de recursos. Ejemplo de estos lenguajes son:

- Python
- Perl
- Ruby
- JavaScript.

Esto ha obligado a mucho estudios de desarrollo de videojuegos a crear y usar lenguajes de script poco conocidos y esotéricos, como son el caso de:

- TorqueScript
- UnrealScript
- C-Script
- AngelScript,
- Squirrel.

De los lenguajes de script incrustables disponibles actualmente, sólo uno ofrece ventajas tanto en licencia liberal, velocidad de ejecución, tamaño en memoria y consumo de recursos además de documentación disponible: Lua.

Aun así Lua fue creado como un lenguaje script de carácter general y para el desarrollo de videojuegos con motores basados en C++ tiene desventajas que se explican con más detalle en esta tesis.

1.1 Lua

Lua es un lenguaje de programación de extensiones extensible de acuerdo a sus autores: Roberto Ierusalimschy, Luiz Henrique de Figueiredo y Waldemar Celes de la Universidad Católica de Río de Janeiro, Brasil.

El desarrollo de Lua ("luna" en portugués) comenzó en 1993 como un derivado de un lenguaje inicial llamado Sol cuando los autores pertenecían al Grupo de Tecnología de Gráficos por Computadora (TECGRAF) [6].
La versión actual y estable de Lua es la versión 5.2 que fue publicada en diciembre del 2012 y que tiene una licencia de código abierto y permisivo MIT. Esta versión agregó soporte de saltos incondicionales y una librería de manejo de operaciones de bits. Killa esta basado en esta versión.

El crecimiento del porcentaje de uso de Lua durante los últimos 10 años, puede apreciarse en el siguiente grafico:
http://www.tiobe.com/index.php/paperinfo/tpci/Lua.html

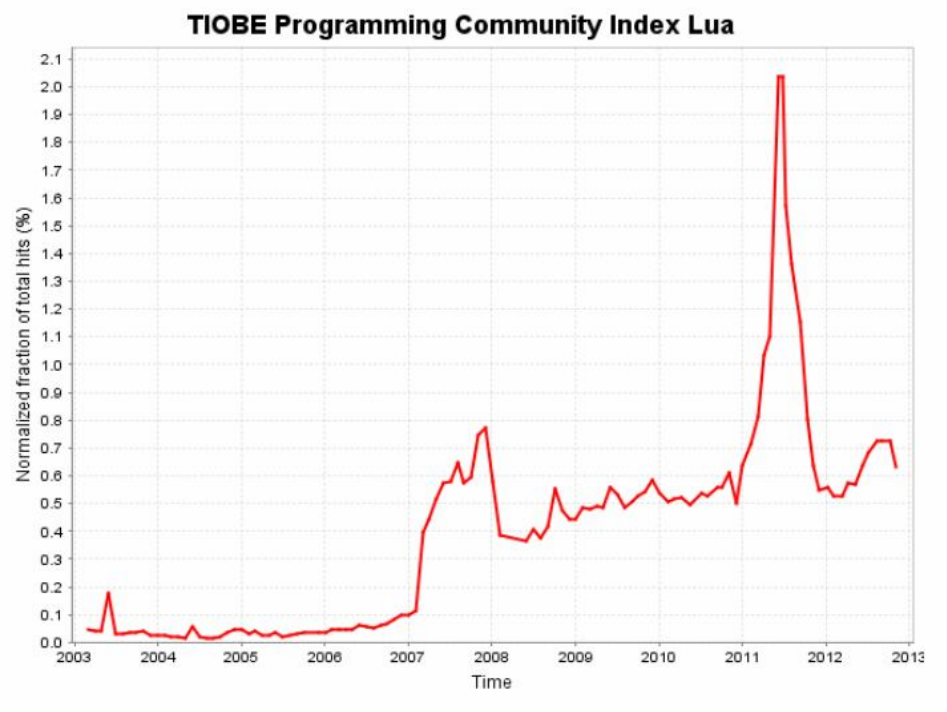

FIGURA 1.1: Uso de Lua en los últimos 10 años

A noviembre del 2012, Lua ocupa el puesto 18 entre los lenguajes más populares que lista TIOBE, habiendo llegado a ocupar el puesto 11 en mayo del 2011.

1.2 JavaScript

JavaScript [7] es un lenguaje de scripts basado en prototipos, con un soporte débil de tipos y que permite el uso de funciones de primer orden. Fue inicialmente desarrollado por Brendan Eich como parte del navegador Netscape Navigator 2.0 y publicado en 1995 bajo el nombre inicial de LiveScript.

JavaScript ha saltado a la fama gracias al predominio del uso de Internet y los navegadores. Es un lenguaje que actualmente tiene muchísimo desarrollo en cuanto a optimizaciones en velocidad y consumo de recursos debido a la feroz competencia existente entre navegadores como Google Chrome, Mozilla Firefox, Microsoft Internet Explorer y Apple Safari por lograr velocidades de carga rápidas y mejorar la interactividad de las páginas de Internet.

El crecimiento del porcentaje de uso de JavaScript durante el periodo 2002 y 2012, puede apreciarse en el siguiente grafico sacado de TIOBE:
http://www.tiobe.com/index.php/paperinfo/tpci/JavaScript.html

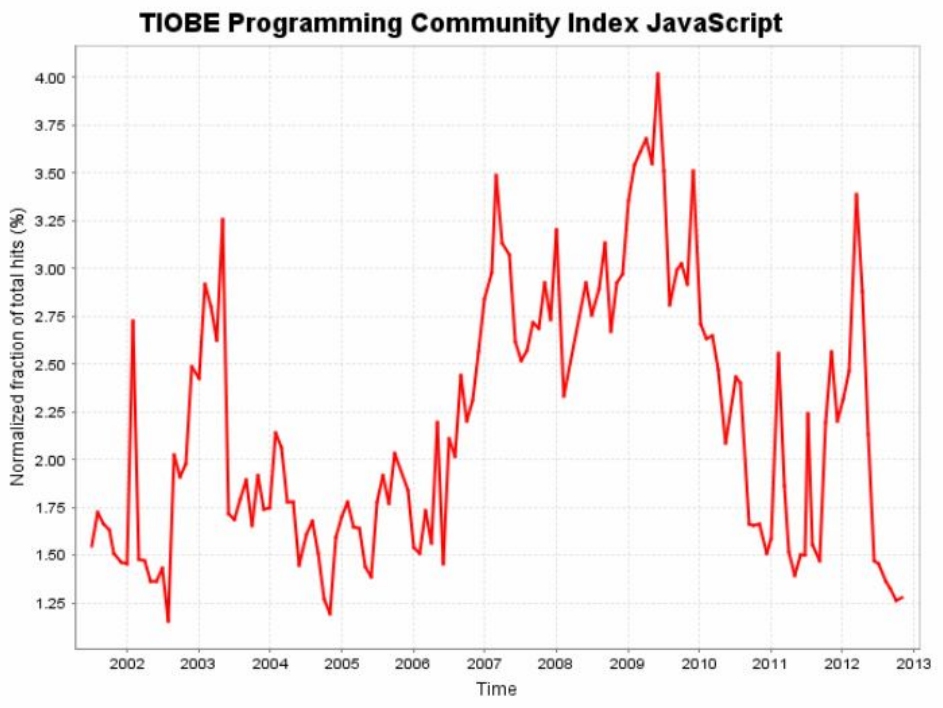

FIGURA 1.2: Uso de JavaScript en los últimos 10 años

A noviembre del 2012, JavaScript ocupa el puesto 11 entre los lenguajes más populares que lista TIOBE.

1.3 Deficiencias de Lua

Lua, a pesar de tener muchas virtudes, sufre también de deficiencias importantes, primeramente su sintaxis no es familiar para un programador con experiencia en lenguajes basados en la sintaxis de C como son C++, C#, Java y JavaScript los cuales son un gran porcentaje de los interesados en usarlo, y aun cuando su sintaxis es relativamente simple de aprender y dominar son los detalles que diferencian Lua de los lenguajes mencionados anteriormente lo que lo hacen incómodo de usar para un programador que tiene también que programar cotidianamente en esos otros lenguajes además de Lua. Si se pudiera minimizar las diferencias en la sintaxis entre Lua y los lenguajes derivados de C y hacer el cambio de contexto que significa pasar de un lenguaje a otro se podría lograr un ahorro de esfuerzo mental considerable además de reducir el número de errores y tiempo de desarrollo.

El otro punto importante es que Lua ha comenzado a sufrir los dolores del peso de su propio éxito y popularidad, actualmente no es raro que se diseñen programas completos en C/C++ sin mayor funcionalidad directa, pues han sido diseñados para ser controlados a través de scripts Lua, los cuales por este motivo son numerosos y complejos, pero Lua, por su origen humilde, no ofrece muchas facilidades para crear programas robustos de tamaño medianamente grande.

1.3.1 Sintaxis no basada en C

Lua usa bloques de código con palabras clave como do, then y end, y es similar en esto a lenguajes como Pascal o Ruby, sin embargo programadores en C++, Java o lenguajes derivados están acostumbrados a usar llaves { } para delimitar bloques de código. Como ejemplo de la sintaxis de Lua a continuación indicamos como calcular el promedio de una lista de valores en Lua:

```
function getAverage(array)
      local sum = 0
      local length = #array
      if length < 1 then
            return nil
      end
      for k = 1, length do
            sum = sum + array[k]
      end
      return sum / length
end

print(getAverage({10, 20, 40, 50}))
```

Desde un punto de vista estadístico y usando TIOBE [2] como un indicador del porcentaje de uso de los lenguajes más populares vemos que entre los 20 lenguajes más populares a Junio del 2012 el porcentaje de uso de lenguajes con sintaxis basada en la sintaxis de C (como son: C++, Java, Objective-C, C#, PHP, Perl, JavaScript y C mismo) es de casi 69%. De esa lista los que lenguajes que usan una sintaxis no basada en C es de solo un 20%. Además, a diferencia de los lenguajes basados en C que mantienen sintaxis bastante similar, varios de estos lenguajes son sintácticamente muy diferentes entre si como es el caso de Lua versus Lisp y PL/SQL. Esto se muestra en la tabla 1.1

TABLA 1.1 Porcentaje de uso de lenguajes con sintaxis basada en C

	LENGUAJE	USO	Basado en C	Diferente a C
1	C	17.73%	0.17725	0
2	Java	16.27%	0.16265	0
3	C++	9.36%	0.09358	0
4	Objective-C	9.09%	0.09094	0
5	C#	7.03%	0.07026	0
6	(Visual) Basic	6.05%	0	0.06047
7	PHP	5.29%	0.05287	0
8	Python	3.85%	0	0.03848
9	Perl	2.22%	0.02221	0
10	Ruby	1.68%	0	0.01683
11	JavaScript	1.47%	0.01474	0
12	Visual Basic .NET	1.22%	0	0.01216
13	Delphi/Object Pascal	1.15%	0	0.0115
14	Lisp	0.99%	0	0.00986
15	Logo	0.86%	0	0.0086
16	Pascal	0.84%	0	0.00844
17	Transact-SQL	0.71%	0	0.00705
18	Ada	0.68%	0	0.00681
19	PL/SQL	0.64%	0	0.00637
20	Lua	0.64%	0	0.00635
			68.45%	**19.29%**

En la figura siguiente se grafica la tabla anterior mostrando los lenguajes con sintaxis derivada de C en tonalidades azules y el resto en otras tonalidades de color.

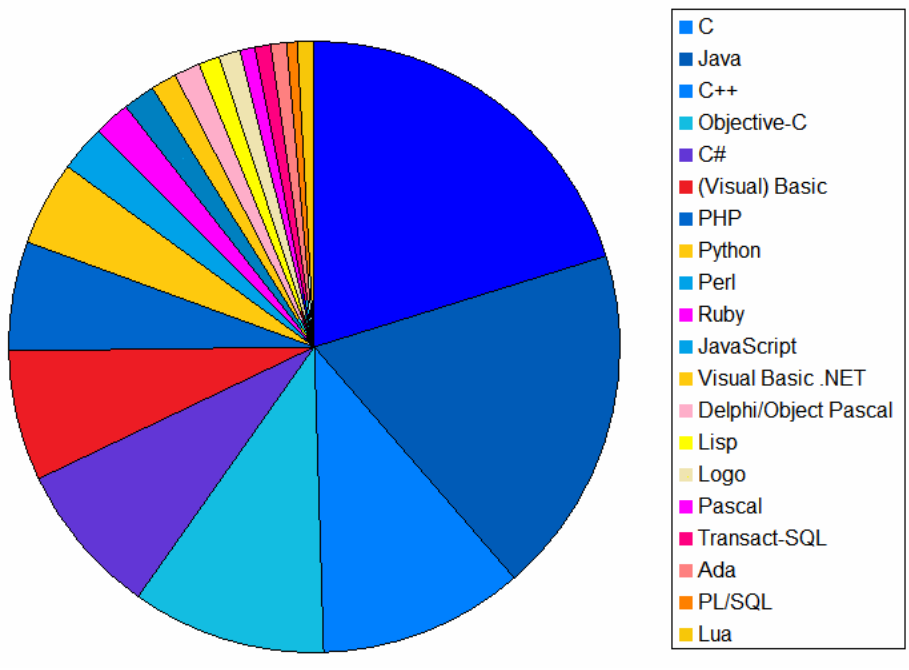

FIGURA 1.3: Lenguajes con sintaxis derivada de C más usados

1.3.2 Arrays con índice en base 1

Lua soporta arrays como un tipo especial de tablas pero maneja los índices de estos empezando en uno, lo cual es similar a FORTRAN, Cobol y Pascal pero diferente de lenguajes derivados de C que usan índices con base en cero.
Existen opiniones de personajes importantes en las Ciencias de la Computación como la de Edsger W. Dijkstra [8] que defienden el uso de índices con base en cero. La opción de usar índices de array con base en uno es explicada por los autores de Lua con el argumento de que es más fácil de entender por los usuarios sin experiencia en programación y casuales, esta decisión, aunque entendible, tiene la desventaja práctica de hacer el uso de Lua problemático para programadores con mayor experiencia y conocimiento en los lenguajes más usados en la industria que tienen que poner especial cuidado en no cometer errores de índice por descuido al trasladar sus programas a Lua.
En Lua *es posible* escribir código con índices en base cero, pero el operador de longitud de array (#) no se podrá usar pues no considera el elemento de índice cero como parte del array, además de que la declaración de arrays debe hacerse de un modo especial, considérese el mismo ejemplo anterior usando base cero:

```
function getAverageZeroIndex(array, length)
      local sum = 0
      if length < 1 then
            return nil
      end
      for k = 0, length - 1 do
            sum = sum + array[k]
      end
      return sum / length
end
print(getAverageZeroIndex({[0]=10, 20, 40, 50}, 4))
```

TABLA 1.2 Porcentaje de uso de lenguajes con índices base cero

	LENGUAJE	USO	Índice 0	Índice 1
1	C	17.73%	0.17725	0
2	Java	16.27%	0.16265	0
3	C++	9.36%	0.09358	0
4	Objective-C	9.09%	0.09094	0
5	C#	7.03%	0.07026	0
6	(Visual) Basic	6.05%	0	0.06047
7	PHP	5.29%	0.05287	0
8	Python	3.85%	0.03848	0
9	Perl	2.22%	0.02221	0
10	Ruby	1.68%	0.01683	0
11	JavaScript	1.47%	0.01474	0
12	Visual Basic .NET	1.22%	0	0.01216
13	Delphi/Object Pascal	1.15%	0	0.0115
14	Lisp	0.99%	0.00986	0
15	Logo	0.86%	0	0.0086
16	Pascal	0.84%	0	0.00844
18	Ada	0.68%	0	0.00681
19	PL/SQL	0.64%	0	0.00637
20	Lua	0.64%	0	0.00635
			74.97%	**12.07%**

La Tabla 1.2 vuelve a mostrar el porcentaje de uso de los 20 lenguajes más usados según TIOBE [2] pero en base a si usan índices de array con base cero o uno. Se ha eliminado Transact-SQL pues no soporta arrays. Se puede ver que la diferencia es aun mayor: 75% usa base cero versus 12% que usa base uno. Esto es debido a que lenguajes populares y similares a Lua como son Ruby y Python usan arrays con índices base cero.

1.3.3 Errores silenciosos debido al uso de variables globales

De acuerdo a la literatura académica y al consenso de la industria de software el uso indiscriminado de variables globales es dañino para la creación de programas robustos y fáciles de mantener.
Esto es detallado en amplitud en el informe "Global variable considered harmful" por W. Wolf y M. Shaw [19] donde se propone incluso la eliminación completa de las variables no locales. En [20] Sward justificando su re-ingeniería de programas que usan variables globales en programas Ada de uso militar, justifica su decisión acertando que el uso de variables globales es peligroso debido a los efectos colaterales que pueden suceder cuando un modulo de un programa modifica inadvertidamente una variable global usada por otros módulos los cuales no tienen luego manera de saber que están usando ahora un valor incorrecto.

Lua (y JavaScript) usan variables globales por defecto, peor aun, cuando una variable no ha sido declarada anteriormente se crea una nueva variable global con valor nulo, esto se produce silenciosamente, sin ningún mensaje de error lo cual introduce errores en el programa cuando el programador inadvertidamente escribe incorrectamente el nombre de una variable.

1.3.4 Falta de operadores de asignación compuestos y operador ternario

Lua carece de los operadores de asignación compuesto (`+=`), (`-=`), etc. Y del operador ternario `(condition)? onTrue : onFalse` que son comunes en lenguajes derivados de C, los cuales simplifican bastante el código repetitivo.

1.4 Deficiencias de JavaScript

Internet ha hecho de JavaScript un lenguaje muy usado, además JavaScript tiene una sintaxis derivada de C y algo parecida a Java lo que es una ventaja si lo comparamos con la sintaxis de Lua, sin embargo posee deficiencias importantes en su sintaxis debido principalmente al poco tiempo invertido en su desarrollo inicial y a la enorme cantidad de código que lo usa actualmente, lo que hace prácticamente imposible corregir sus defectos sin volver inoperante una gran parte de los sitios web que lo usan para dar interactividad a sus paginas web.

1.4.1 Difícil de lograr un consenso para mejorar el lenguaje JavaScript

La sintaxis de JavaScript es actualmente controlada por el consorcio ECMAScript el cual esta integrado por organizaciones como Microsoft, Google, Mozilla, Apple y otros los cuales velan por el soporte y desarrollo del lenguaje en los navegadores de mayor uso. Cambios de sintaxis sin la aprobación de todos los miembros son imposibles siendo así que la propuesta agresiva para modernizar el lenguaje conocida como ECMAScript versión 4 fue rechazada por compañías como Google, Microsoft y Yahoo que prefirieron un cambio menos drástico del lenguaje.

1.5 Estado del arte

A continuación se presentan trabajos similares al presentado en esta tesis.

1.5.1 Bright

El único intento similar a esta tesis encontrado en la literatura fue el desarrollo de *Bright* [9], el cual fue presentado en el *Lua Workshop 2008*. Lamentablemente el proyecto fue de carácter cerrado y nunca llegó a hacerse público. Sin embargo algunas de las razones importantes para crear *Bright* fueron resumidas en dicho informe. Los autores de *Bright* realizaron cambios a la semántica como: indización de vectores con base en cero (Lua usa base 1), creación de un tipo indefinido (*undefined*) y hacer que NULL sea un valor válido de búsqueda para las tablas, también agregaron operadores de manejo de bits. El efecto conseguido con dichos cambios según el informe fue una menor resistencia a la adopción del lenguaje en la organización que la creó. El hecho de hacer que los índices de array comiencen en cero redujo el número de errores para los programadores que tienen experiencia programando en C, la creación de un tipo indefinido para detectar variables no inicializadas globales hizo que los errores de programación debido a errores de escritura fueran encontrados fácilmente.

1.5.2 Comunidad Lua

Aunque han habido propuestas para modificar la sintaxis de Lua para hacerla más similar a C/C++ o JavaScript, dichos intentos no han recibido mayor acogida de parte de los creadores ni de la comunidad de Lua, lo cual es entendible pues muchos miembros de esta comunidad consideran a Lua como un lenguaje independiente y completo por si mismo y no necesitan programar en lenguajes derivados de C al mismo tiempo, su razón de usar Lua es precisamente evitar usar otros lenguajes mas complicados.

Otra consideración importante es que la sintaxis de Lua fue creado para atraer y ser fácil de usar por un público sin experiencia previa en programación y basándonos en su aceptación entre dichos programadores casuales podríamos presumir que cumple con su objetivo de ser fácil de aprender. No tendría sentido parchar ahora la sintaxis del lenguaje para hacerlo mas similar a JavaScript o C/C++ pues terminaría en un estado indefinido sin llegar a ser jamás ninguno de ellos pero dejando de ser definitivamente Lua. Esto debido a que ya existe una considerable cantidad de código y documentación de Lua en la industria actual que forzaría que dichos cambios no sean muy agresivos para evitar tener que reescribir mucho código y documentación existente.

Killa surgió debido a la posibilidad que se tuvo de crear un nuevo lenguaje que mezcle lo mejor de ambos mundos (Lua y JavaScript) incluyendo en el proceso mejoras para poder desarrollar programas con menos errores. De Lua se usó su maquina virtual optimizada y su API de incrustación lo que permite incrustar Killa igual que Lua en cualquier programa que soporte Lua. De JavaScript se usó la sintaxis (que a su vez esta basada en C, C++ y Java) lo que permite una mayor familiaridad para programadores que usan dichos lenguajes.

1.6 Objetivos

El objetivo general de esta tesis es crear un lenguaje de programación script que permita el desarrollo de videojuegos de una manera sencilla y segura. Este lenguaje debe ser libre de usar en diferentes entornos y plataformas tanto por programadores aficionados como por profesionales de la industria de videojuegos.

Los objetivos específicos de este trabajo son:

1. El lenguaje debe ser fácil de incrustar en motores de videojuegos existentes escritos en C++.
2. La sintaxis del lenguaje debe estar basada en C. Pero debe incorporar características avanzadas de lenguajes como JavaScript.
3. El lenguaje debe ser de calidad industrial, es decir que debe haber sido probado extensivamente y no tener errores.
4. El interprete del lenguaje debe soportar múltiplex plataformas:
    - Windows
    - Mac
    - Linux
    - iOS
    - Android
5. El lenguaje debe permitir evitar errores de programación comunes en lenguajes como Lua y JavaScript.
6. El lenguaje debe tener licencia liberal para poder ser usado en proyectos comerciales cerrados.

1.7 Justificación

La justificación de este trabajo parte del hecho de trabajar en la industria del desarrollo de videojuegos por bastante tiempo y no encontrar un lenguaje adecuado para el desarrollo de nuestros títulos, considerando que la mejor opción libre existente (Lua) tiene las limitaciones presentadas anteriormente y aunque existan lenguajes script similares a lo buscado (AngelScript, Squirrel) no han sido probados y usados extensivamente como Lua.

2. Capitulo II: El lenguaje de programación Killa

La palabra Killa proviene del lenguaje Quechua y significa “luna”. Fue nombrado asi en tributo al lenguaje Lua cuyo nombre significa lo mismo en portugués.

Killa es un lenguaje de programación que permite programar usando múltiples paradigmas como son programación procedural o programación orientada a objetos usando el concepto de meta-tablas heredado de Lua.

Killa se desarrolló en base al código fuente de Lua versión 5.2 y usa la máquina virtual de Lua para su ejecución. Es debido a ello que aunque su sintaxis por fuera es diferente, mantiene por dentro un alto grado de compatibilidad con la estructura de programas escritos en Lua.

Muchos de los conceptos básicos de Lua están presentes en Killa pero con diferente presentación. Killa usa el inglés para definir su sintaxis. Algunos de los términos empleados en esta tesis no tienen una traducción directa en español y se dejarán en inglés, así mismo los ejemplos de código y comentarios dentro del código serán hechos todos en inglés.

2.1 Definiciones

Los *identificadores* en Killa pueden ser cualquier secuencia de letras, dígitos o guiones de subrayado (_) mientras no comiencen con un dígito o sean *palabras reservadas* por el lenguaje.

La siguiente es la lista de *palabras reservadas* del lenguaje:

break	**do**	**each**	**else**
false	**for**	**function**	**goto**
if	**in**	**null**	**private**
public	**return**	**to**	**true**
var	**while**		

FIGURA 2.1: Palabras reservadas de Killa

Killa es un lenguaje que diferencia minúsculas de mayúsculas.
Los siguientes identificadores son usados como variables especiales:

arg	**global**	**this**	

FIGURA 2.2: Identificadores especiales de Killa

El símbolo `global` es usado para acceder a las variables globales.
El símbolo `arg` para acceder a los argumentos de una función
El símbolo `this` es el identificador de objeto en métodos de objeto

La siguiente es la tabla de operadores y símbolos adicionales empleados por el lenguaje:

+	+=	-	-=	*	*=	/	/=
%	%=	**	==	!=	>	>=	<
<=	=	(	)	{	}	[	]
;	:	,	.	&&	||	!	//
&	^	|	<<	>>	~		
/*	*/	..	$	..=			

FIGURA 2.3: Símbolos y operadores en Killa

Además de las palabras reservadas que no pueden ser usadas como identificadores, Killa tiene una lista de palabras reservadas para futuras versiones del lenguaje:

case	match	class	const	continue
default	extends	finally	implements	import
interface	new	override	protected	super
switch	throw	try	undefined	

FIGURA 2.4: Palabras reservadas para futuras versiones de Killa

El uso de los términos anteriores dará lugar a un error de no implementación en tiempo de compilación, previniendo su uso accidental.

Las *cadenas literales* son delimitadas por comillas: `"one string"` o apóstrofes `'another string'`.

Una *constante numérica* puede ser escrita con una parte decimal opcional y una parte exponencial opcional. Killa soporta valores hexadecimales que empiecen con: `0x`, opcionalmente permite separar dígitos con guiones de subrayado para facilidad de lectura, ejemplos validos:

```
3   3.0   3.1416   314.16e-2   0.31416E1   0x56   1_000_000
```

Los comentarios en Killa empiezan con el símbolo `//` para comentarios de línea o se demarcan con `/* ... */` para comentarios que abarcan múltiples líneas:

```
// This is a single line comment.

/* This is a
   multiline comment */
```

2.2 Tipos y valores

Killa es un lenguaje de *tipos dinámicos*. Esto significa que las variables no tienen tipos, solo los valores los tienen.

Todos lo valores son de *primera clase*. Esto significa que todos los valores pueden ser almacenados en variables, pasados como argumentos a otras funciones, y devueltos como resultados. Internamente hay 8 tipos básicos:

`null`, `boolean`, `number`, `string`, `function`, `userdata`, `thread` y `table`.

`null` es un tipo especial con un solo valor (`null`). No es válido tratar de usar este valor para operaciones aritméticas o de concatenación y si se intenta llamar a una variable de valor nulo como función sucederá un error en tiempo de ejecución. `null` es considerado falso en sentencias condicionales y es válido asignar el valor de `null` a las variables para que el manejador de memoria pueda liberar el espacio consumido por sus contenidos previos.

`boolean` es un tipo con solo dos valores (verdadero o falso): `true` o `false`. Son valores de tipo lógico que se usan dentro de expresiones condicionales. Además de `false` el valor de `null` también hace una condición falsa, cualquier otro valor es verdadero. Esto significa que se debe tener cuidado especial con el numero cero y la cadena vacía que son considerados como valores verdaderos en Killa a diferencia de C/C++ o JavaScript.

`number` es el tipo que representa los valores numéricos. Internamente esta representado como un numero de tipo flotante de 64-bits (siguiendo el estándar IEEE-754). Para efectos de usar Killa en dispositivos sin soporte de números flotantes se puede redefinir este tipo para usar enteros o también números flotantes de menor precisión.

`string` es el tipo usado para representar cadenas de caracteres.
Internamente Killa almacena las cadenas como un buffer de caracteres de 8-bits pero no usa como delimitador de final de cadena el carácter nulo o vacio (‘\0’). Esto permite usar las cadenas de Killa para almacenar cualquier tipo de información binaria (como cadenas Unicode o data de imágenes).

`function` representa un bloque de código ejecutable.
En Killa es posible almacenar funciones en una varialb y pasar funciones como parámetro a otras funciones. Gracias a ello es posible usar Killa como un lenguaje de tipo funcional.

`userdata` es un tipo usado para guardar valores arbitrarios de C en variables. Este tipo corresponde a una dirección de un bloque de memoria del sistema y solo puede ser asignado o ser comparado con otros valores `userdata`. Estos valores no pueden ser creados ni modificados en Killa solo a través del API C.

`thread` representa hilos independientes de ejecución denominados *corutinas*. Las corutinas de Killa no son hilos a nivel de sistema operativo sino una facilidad del lenguaje para crear tareas en paralelo, detenerlos en cualquier momento y volver a resumir su ejecución mas adelante o terminarlos.
Este mecanismo permite por ejemplo crear sistemas de inteligencia artificial donde el comportamiento de cada entidad es controlado por una corutina y no se consume recursos significativos del sistema operativo a diferencia de si se usaran hilos nativos del sistema.

`table` una tabla es un array asociativo, es decir un array que puede ser indexado no solo con números enteros sino con cualquier otro valor (excepto `null`). Esta es la única estructura de datos que el lenguaje soporta pero puede usarse para representarse arrays, conjuntos, árboles y objetos. Al igual que los índices, los valores guardados por un objeto pueden ser de cualquier tipo, lo que implica que pueden ser funciones cual se usa para añadir *métodos* a una tabla permitiendo una programación orientada a objetos rudimentaria.
Internamente las tablas están optimizadas para máximo rendimiento si se usan como arrays continuos de memoria pues cada tabla tiene una parte continua (array) y una parte asociativa lo que permite un rápido acceso en el caso de tablas con elementos continuos usados como arrays.

Los valores de tipo `table`, `function`, `thread`, `string` y `userdata`, son almacenados *por referencia*, es decir que las variables que las contienen solo almacenan la dirección donde se guardan internamente. Esto hace que operaciones como asignación, paso por parámetro y los valores devueltos por una función no necesiten realizar copia alguna de estos datos.

En caso que se necesite saber el tipo de una variable la función global `type` devuelve una cadena con el tipo del valor contenido por la variable:

```
var a = "student"
type(a) // returns: "string"
```

Los tipos de datos son convertidos a cadena en caso se encuentren dentro de una operación de concatenación. Las cadenas son convertidas a números si se encuentran dentro de una operación aritmética. También se puede convertir una cadena a número usando la función `tonumber()`.

```
var age = 25
var s = "Age: " .. age;      // s == "Age: 25"
var q = 4 + tonumber("3.1")  // q == 7.1
var t = 4 + "31"             // t == 35
```

Se ha pensado agregar declaración de tipos opcionales a Killa en una futura versión, esto permitirá agregar comprobación de tipos en ciertos casos:

```
var age:int = 25              // type int
var man:Entity = new Entity() // type object
age = man                     // this would fail
```

2.3 Declaración de variables

Antes de poder usar variables es necesario declararlas. Hay tres tipos de variables: globales, locales y campos de tablas.

```
Var ::= [private] var Name [;]
Var ::= [private] var Name '= ' exp. [;]
Var ::=  public var Name '= ' exp. [;]
```

Las variables locales se definen usando `var` o `private var`.
El término `private` es opcional y se sobreentiende:

```
private var health = 100   // local variable
var counter = 1            // local variable
```

Las variables locales tienen ámbito de definición estático y pueden ser accedidas libremente por funciones definidas dentro de su mismo ámbito de definición.
Se entiende como ámbito de definición al bloque de código que contiene la declaración de la variable, sea este el archivo del programa o la función contenedora o el bloque de código contenedor.
Si una variable local no ha sido asignada tiene el valor por defecto `null`.

```
var name  // name is null
```

Las variables globales se definen usando `public var`
Es necesario asignar un valor a una variable global al momento de crearla.

```
public var VERSION = "0.1" // global variable
```

Las variables de tipo global pueden ser accedidas desde cualquier parte del programa, pero deben ser accedidas a través del objeto `global`:

```
public var GRAVITY = 9.8            // global variable

var speed = global.GRAVITY * time   // using the global
speed = GRAVITY * time              // error: GRAVITY is global
```

El hecho de requerir el uso del objeto global para poder referenciar variables globales permite encontrar fácilmente dentro del código del programa el uso de dichas variables. Esta es una característica única del diseño de Killa y se sustenta en la idea de limitar el uso de variables globales al máximo y hacer visible su uso dentro del código pues se considera que el abuso de variables globales es una característica de programas de mala calidad.

A diferencia de Lua o JavaScript las variables deben ser declaradas antes de poder ser usadas, no existe una declaración automática de variables globales. Tratar de usar una variable no declarada dará lugar a un error en tiempo de compilación. Esto permite detectar errores comunes que suceden por fallas de correcta escritura de los nombres de las variables en esos lenguajes.

Los campos de tablas son accedidos a través corchetes:

Var ::= *PrefixExp* '[' *Exp* ']'

Pero también es posible acceder a ellos usando la forma equivalente:

Var ::= *PrefixExp* '.' *Exp*

Ejemplo:

```
var programmer = { name: "", age: 25 }
programmer["name"] = "Laurens"
programmer.age = 31
```

Es posible declarar múltiples variables locales en la misma sentencia:

Var ::= [**private] var** *Name* **{**',' *Name***}** '= ' *Exp* **{**',' *Exp***}** [;]

Ejemplo:

```
var x, y = 5, 10  // multiple declaration and assignment
print(x)          // output: 5
print(y)          // output: 10
```

Por cuestión de diseño no es posible hacer lo mismo con las variables globales, Killa es más exigente con la declaración de variables globales y fuerza a declarar cada una en una sentencia separada.

```
public var x, y = 5, 10       // error
```

2.4 Sentencias de ejecución

2.4.1 Script

Es la unidad de ejecución del lenguaje.

```
Script ::= { Stat  [';'] }
```

Es simplemente una secuencia de sentencias de ejecución (*statements* en inglés) separadas opcionalmente por puntos y comas. Un *script* puede definir variables locales, recibir argumentos y retornar valores, puede ser almacenado en un archivo o en una cadena dentro del programa anfitrión. Para ejecutar un *script*, Killa primero lo precompila en instrucciones para una máquina virtual y luego ejecuta el código compilado usando el interprete de la máquina virtual. Los *scripts* también pueden ser precompilados y almacenados en formato binario.

2.4.2 Bloque

Un bloque es una secuencia de sentencias de ejecución delimitada por llaves. Sintácticamente es lo mismo que un *script* pero al estar delimitado por llaves se puede considerar como una sola sentencia.

```
Block ::= '{' Script '}'
Stat  ::= Block
```

2.4.3 Declaraciones de variables y asignaciones

Ya se vio anteriormente la sintaxis de la declaración de variables.
En cuanto a asignaciones, Killa permite realizar múltiples asignaciones en una sola sentencia. La sintaxis para asignaciones define una lista de variables en el lado izquierdo y una lista de expresiones en el lado derecho. Los elementos en ambas listas están separados por comas.

```
Stat    ::= VarList '=' ExpList [';']
VarList ::= Var { ',' Var }
ExpList ::= Exp { ',' Exp }
```

Si la lista de variables y la lista de expresiones no coinciden en tamaño se completaran con valores nulos o se descartaran. La sentencia de asignación primero evalúa todas las expresiones del lado derecho y recién entonces realiza

las asignaciones.

```
x , y = y , x   // Swap values.
```

Killa soporta asignaciones compuestas, es decir operar y asignar a la misma vez con los operadores: `+=`, `-=`, `*=`, `/=` y `%=`.

```
x += 1          // Equivalent to x = x + 1
y %= 2          // Equivalent to y = y % 2
```

La asignación compuesta `..=` es nuevo en Killa y sirve para concatenar cadenas:

```
var s1 = 'testing '
s1 ..= "concatenation"
print(s1) // prints: "testing concatenation"
```

2.4.4 Estructuras de control

Las estructuras de control existentes son `while`, `do` y `if`:

```
Stat ::= while '(' Exp ')' Block
Stat ::= do  Block  while '(' Exp ')' ';'
Stat ::= if '(' Exp ')' Block { else if '(' Exp ')' Block }
         [ else Block ]
```

La expresión condicional de una estructura de control puede retornar cualquier valor. Pero sólo `false` y `null` son considerados falsos. El número cero y la cadena vacía son considerados verdaderos.

La instrucción `return` es usada para retornar valores de una función o script. Se puede devolver más de un valor y por eso la sintaxis es:

```
Stat ::= return [ ExpList ] [';']
```

La instrucción `break` es usada para terminar la ejecución de una instrucción de bucle `while`, `do` o `for`. Saltándose a la siguiente instrucción después del bucle.

```
Stat ::= break [';']
```

La instrucción `goto` es usada para interrumpir la ejecución normal de un programa y saltar a una etiqueta de salto siempre y cuando la etiqueta de salto este dentro de la función o bloque contenedor de la llamada a salto.

```
Stat ::= goto Label [';']
```

Estos bloques de control son similares a los de C++ o JavaScript, como un ejemplo, se muestra el uso de estos bloques para imprimir una lista con los números primos menores que 100:

```
// Eratosthene's sieve
function primes(limit) {
    var sieve = []
    var ret = []
    var i = 2
    while (i < limit) {
        while (sieve[i]) {
            i += 1
        }
        if (i < limit) {
            table.insert(ret, i)
            for (var j = i * i;  j <= limit; j += i) {
                sieve[j] = true
            }
            i += 1
        }
    }
    return ret
}
print(table.concat(primes(100), ","))
```

Salida:

```
2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97
```

2.4.5 Bucle for

La instrucción `for` tiene la forma:

Stat ::= **for** '(' *ExpInit* ';' *ExpCond* ';' *ExpInc* ')' *Block*

La expresión de inicialización (*ExpInit*) sirve para inicializar variables de control y es ejecutada antes de iniciar el bucle.
La expresión condicional (*ExpCond*) sirve para continuar o salir del bucle y es ejecutada en cada repetición al inicio del bucle.
La expresión de incremento (*ExpInc*) sirve para modificar las variables de control y es ejecutada al final de cada repetición del bucle.
Las variables definidas en la expresión de inicialización son solo válidas dentro de la instrucción `for` y su bloque asociado.
El cuerpo del bucle es el bloque (*Block*) que se repite en cada iteración.
Este tipo de bucle es idéntico al bucle `for` encontrado en lenguajes como C++ o JavaScript.

Como un ejemplo se muestra el uso de este bucle par imprimir las primeras 10 potencias de 2:

```
function printPowers(base, count) {
    for (var k = 0; k < count; k += 1) {
        print(base.."^"..k.." = "..(base ** k))
    }
}
printPowers(2, 10)
```

Salida:

```
2^0 = 1
2^1 = 2
2^2 = 4
2^3 = 8
2^4 = 16
2^5 = 32
2^6 = 64
2^7 = 128
2^8 = 256
2^9 = 512
```

2.4.6 Bucle for each

La instrucción `for each` tiene 2 formas:

Stat ::= **for each** '(' **var** *Name* '=' *ExpBegin* **to** *ExpEnd* [',' *ExpInc*] ')' *Block*

Esta forma crea una variable (*Name*) la cual toma el valor inicial de (*ExpBegin*) e itera hasta alcanzar el valor final (*ExpEnd*). Opcionalmente es posible definir el incremento de cada iteración (que por defecto es 1) con la expresión (*ExpInc*). Todas las expresiones son evaluadas al inicio del bucle y se consideran constantes en el transcurso del bloque. Esto es similar al `for` numérico de Lua.

Como ejemplo mostramos como podemos imprimir los elementos de un array:

```
var t = [100,200,300]
for each (var k = 0 to 2) {
    print(k ..") "..t[k])
}
```

La segunda forma es:

Stat ::= **for each** '(' **var** *ExpList* **in** *Iterator* ')' *Block*

En esta forma la lista de expresiones (*ExpList*) es iterada con todos los valores devueltos por la función iteradora (*Iterator*). Este forma de bucle es muy útil para recorrer la lista de campos de una tabla usando las funciones iteradoras `pairs` y `ipairs` o tambien iteradotes personalizados.

Como ejemplo mostramos como imprimir los elementos del mismo array del ejemplo anterior usando la función de la librería `ipairs` que devuelve un iterador a las llaves y valores de la tabla:

```
var t = [100,200,300]
for each (var k,v in ipairs(t)) {
    print(k ..") "..v)
}
```

Como ejemplo del uso de tiradores personalizados mostramos como implementar un iterador similar a ipairs manualmente:

```
function list_iterator(t) {
    var i = -1
    var n = $t
    return function () {
        i += 1
        if (i < n) { return t[i] }
    }
}

var t = [10, 20, 30, 40]
for each (var element in list_iterator(t)) {
    print(element)
}
```

Salida:

```
10
20
30
40
```

2.4.7 Llamadas de función

Una llamada a función puede ser ejecutada como una sentencia de ejecución:

Stat ::= FunctionCall [';']

En este caso todos los valores devueltos son descartados.

2.5 Expresiones

Las expresiones en Killa son las siguientes:

Exp ::= *PrefixExp*
Exp ::= **null** | **false** | **true**
Exp ::= *Number*
Exp ::= *String*
Exp ::= *FunctionDef*
Exp ::= *TableConstructor*
Exp ::= '...'
Exp ::= *Exp BinOperator Exp*
Exp ::= *UnaryOperator Exp*
Exp ::= *TernaryExpression*
PrefixExp ::= *Variable* | *FunctionCall* | '(' Exp ')'

Las expresiones de argumentos variables, denotado por los tres puntos ('...'), solo pueden ser usadas cuando están directamente dentro de una función de argumentos variables.

Los operadores binarios comprenden los operadores aritméticos, operadores relacionales, y operadores lógicos.
Los operadores unarios comprenden el signo negativo, el signo de negación lógico y el símbolo de longitud de tabla.

Llamadas a funciones y expresiones de argumentos variables pueden resultar en múltiples valores. Si una expresión es usada como una sentencia (solo es posible con llamadas a funciones), entonces todos los valores de la lista de retorno son descartados. Si una expresión es usada como el último (o el único) elemento de una lista de expresiones, entonces no se modifica la lista (a menos que la llamada este encerrada entre paréntesis). En todos los otros contextos, Killa ajusta la lista de resultados a un elemento, descartando todos los valores excepto el primero.

Aquí unos ejemplos:

```
f();               // all returning values discarted
g(f(), x);         // f() is adjusted to 1 result
g(x, f());         // g gets x plus all results from f()
a, b, c = f(), x;  // f() is adjusted to 1 result (c undefined)
a, b = ...;        // a gets the first vararg parameter, b gets
                   // the second (both a and b can get undefined
                   // if there is no corresponding variable
                   // argument)

a, b, c = x, f();  // f() is adjusted to 2 results
a, b, c = f();     // f() is adjusted to 3 results
return f();        // returns all results from f()
return ...;        // returns all received vararg parameters
return x, y, f();  // returns x, y, and all results from f()
{f()};             // creates a list with all results from f()
{...};             // creates a list with all vararg parameters
{f(), null};       // f() is adjusted to 1 result
```

Cualquier expresión encerrada entre paréntesis resulta siempre en un único valor. De esta manera (`f(x, y, z)`) es siempre un único valor, aun si `f` devuelve múltiples valores. El valor de (`f(x, y, z)`) es el primer valor retornado por `f` o `null` si `f` no retorna algún valor.

2.5.1 Operadores aritméticos

Killa soporta los operadores aritméticos habituales:
(+) adición, (-) sustracción, (*) multiplicación, (/) división, (%) módulo y (**) exponenciación. Además de sus respectivas asignaciones compuestas: (+=), (-=), (*=), (/=) y (%=).

A diferencia de JavaScript y C/C++, Killa no soporta los operadores de incremento y decremento (--) y (++) esto es debido al diseño del lenguaje que considera que el uso de esos operadores posibilitan la introducción de errores sutiles en el código cuando son usados excesivamente. Haciendo mas lenta la verificación y revisión de programas.

La exponenciación es válida para cualquier exponente numérico.
Por ejemplo, `x ** (-0.5)` calcula la inversa de la raíz cuadrada de `x`.
El módulo es definido como:

```
a % b == a - math.floor(a / b) * b;
```

Es decir, como el residuo de la división que redondea el cociente hacia menos infinito.

2.5.2 Conversión automática de tipos

Killa soporta conversión automática de tipos en tiempo de ejecución. Cualquier operación aritmética realizada en cadenas trata de convertir la cadena en número, igualmente en las operaciones de concatenación los números son convertidos a cadenas automáticamente. Para mayor control sobre las conversiones se debe usar la función de la librería `string.format`

2.5.3 Operadores relacionales

Los operadores relacionales en Killa son:

```
==    !=    <     >     <=    >=
```

Estos operadores resultan siempre evaluados a valores de verdadero o falso.

El operador de igualdad (`==`) compara primero el tipo de sus operandos, si los tipos son diferentes, entonces el resultado es falso. De lo contrario, los valores de los operandos son comparados.

Números y cadenas son comparados de la manera habitual.
Objetos (`table`, `userdata`, `thread` y `function`) son comparados por referencia: dos objetos son considerados iguales si y solo si son el mismo objeto.

Cada vez que se crea un nuevo objeto, este nuevo objeto es diferente de cualquier otro objeto ya existente.

Las reglas de conversión no se aplican para comparaciones de igualdad.
Por lo tanto, `"0" == 0` se evalúa como falso, además que `t[0]` y `t["0"]` denotan diferentes campos de un objeto.

El operador (`!=`) es la negación lógica del operador de igualdad (`==`).

2.5.4 Operadores lógicos

Los operadores lógicos en Killa son: `&&` (Y lógico), `||` (O lógico) y `!` (negación lógica). Al igual que las estructuras de control, todos los operadores lógicos consideran `false` y `null` como valores de falso (`false`) y todo lo demás como verdadero (`true`).

El operador (`&&`) retorna su primer argumento si este valor es `false` o `null`, de lo contrario retorna su segundo argumento, ejemplos:

```
null && 10          // -> null
false && error()    // -> false
false && null       // -> false
0 && 20             // -> 20
```

El operador (`||`) retorna su primer argumento si este valor es diferente de `false` o `null`, de lo contrario retorna su segundo argumento, ejemplos:

```
10 || 20            // -> 10
10 || error()       // -> 10
null || "a"         // -> "a"
```

```
false || null      // -> null
```

Ambos operadores evalúan el segundo operando solo si es necesario.

2.5.5 Concatenación

El operador de concatenación de cadenas en Killa es el operador (..) En Killa el operador de concatenación convierte números a cadenas y el operador de suma convierte cadenas a números eliminando la ambigüedad. A diferencia de JavaScript no existe un único operador para concatenación y suma aritmética, eliminando de esta modo cierta ambigüedad.

2.5.6 Operador de tamaño

El operador de tamaño en Killa es ($). Se usa este símbolo debido a que el símbolo usado en Lua (#) será usado posteriormente para pre-procesamiento.
Este operador devuelve la longitud de una cadena y el tamaño de un array si es usado como array:

```
print($"0123456789")    // Output: 10
print(${1,2,3,4,5,6})   // Output: 6
print(${key:1,2,3,4,5}) // Output: 4 (is a table not an array)
```

2.5.7 Operadores binarios

Killa soporta los mismos operadores binarios de C. Operan sobre valores numericos convirtiendolso internamente en valores enteros sin signo de 32 bits

```
~   &   ^   |   <<   >>
```

2.5.8 Orden de precedencia

El orden de precedencia de operadores en Killa sigue la siguiente tabla, de menor a mayor prioridad:

```
**
! ~ - $
* / %
+ -
>> <<
& ^ |
..
== != <= < > >=
&& ||
```

Como es habitual, se pueden usar paréntesis para cambiar el orden de precedencia de una expresión. Los operadores de concatenación (..) y exponenciación (**) son asociativos por la derecha. El resto de operadores binarios son todos asociativos por la izquierda.

2.5.9 Constructores de tablas

La sintaxis para construir tablas asociativas esta definida por:

```
TableConstructor ::= '{' [FieldList] '}'
FieldList ::= Field {FieldSep Field} [FieldSep]
Field ::= '[' Exp ']' ':' Exp | Name ':' Exp
FieldSep ::= ',' | ';'
```

Cada campo de la forma `[exp1]:exp2` agrega a la tabla un campo con índice `exp1` y valor `exp2`. Un campo de la forma `name:exp` es equivalente a: `table["name"] = exp`.

La sintaxis para construir tablas de array es definida por:

```
TableConstructor ::= '[' [FieldList] ']'
FieldList ::= Exp {FieldSep Exp} [FieldSep]
FieldSep ::= ','
```

Cada expresión `exp` definida es equivalente a: `[k] = exp`, donde `k` es un valor numérico que empieza desde 0 y se va incrementando con cada elemento.

En Killa, a diferencia de Lua, no se permite mezclar tablas de tipo asociativo con tablas de tipo array al menos en tiempo de definición. Futuras versiones de Killa diferenciaran completamente estos tipos de datos

```
var mixed = {1, 3, name: "foo", ['new']: 234}  // error in Killa
```

Si el último campo en la lista tiene la forma exp y la expresión es una llamada a función o una expresión de argumentos variables, entonces todos los valores retornados por esta expresión ingresan a la lista consecutivamente. Para evitar esto, se debe encerrar la llamada a función o la expresión de argumentos variables entre paréntesis.

La lista de campos puede tener un separador de campo sobrante, como una manera de facilitar código generado automáticamente.

```
var table = [1, 2, 3, 4, 5, ]  // valid
```

2.5.10 Llamadas de función

Una llamada a función en Killa tiene la siguiente sintaxis:

```
FunctionCall ::= PrefixExp Args
```

En una llamada a función, *PrefixExp* y *Args* son evaluados primero.
Si el valor de *PrefixExp* tiene tipo de función, entonces dicha función es llamada con los argumentos *Arg* pasados.

La forma:

```
FunctionCall ::= PrefixExp '.' Name Args
```

permite llamar funciones que son campos de una tabla.

Los argumentos tienen la siguiente sintaxis:

```
Args ::= '(' [ExpList] ')'
```

Todos los argumentos en la expresión son evaluados antes de la llamada.
Ejemplos:

```
f(x, y);   // calling function f with arguments x and y
o.g();     // calling function g from table o without arguments
```

Una llamada de la forma `return` *FunctionCall* es una llamada "final".
Killa implementa estas llamadas adecuadamente: la función llamada al final reutiliza el *stack* de memoria usada por la función llamadora inicial.
De esa manera, no hay límite en el número de llamadas recursivas anidadas que se puede hacer. Killa a diferencia de Lua obliga a usar paréntesis para encerrar los argumentos.

2.5.11 Definiciones de función

La manera de definir funciones esta dada por:

```
Stat ::= [private] function FuncName FuncBody
Stat ::= public function FuncName FuncBody
FuncName ::= Name {'.' Name}
FuncBody ::= '(' [ParameterList] ')' Block
```

Se pueden declarar funciones anónimas usando:

```
Function ::= function FuncBody
```

La sentencia:

```
function f(...) { }
```

es entendida como:

```
private var f = function(...) { }
```

La sentencia:

```
function t.a.b.c.f() { }
```

es equivalente a:

```
t.a.b.c.f = function() { }
```

La sentencia:

```
public function f(...) { }
```

es equivalente a:

```
public var f = function(...) { }
```

Una definición de función es una expresión ejecutable, cuyo valor tiene tipo de función. Cuando Killa pre-compila un *script*, todos los cuerpos de funciones son pre-compilados también, de esa manera, cuando quiera que Killa ejecute la definición de la función, la función es instanciada (o cerrada).
Esta instancia de función (o *closure*) es el valor final de la expresión.
Diferentes instancias de la misma función pueden hacer referencia a diferentes variables externas.

Los parámetros actúan como variables locales que son inicializadas con los valores de los argumentos:

```
ParameterList ::= NameList [',' '...'] | '...'
NameList ::= Name [',' Name]
```

Cuando una función es llamada, la lista de argumentos es ajustada a la longitud de la lista de parámetros, a menos que la función sea de tipo de argumentos variables, lo cual es indicado con los tres puntos (…) al final de la lista de parámetros. Una función de este tipo no ajusta su lista de argumentos, en vez de ello, este tipo de función junta todos los argumentos extra y los envía a la función a través de una expresión de argumentos variables, la cual es también denotada por tres puntos consecutivos.

Si una expresión de argumentos variables es usada al inicio de otra expresión o en el medio de una lista de expresiones, entonces es ajustada a un solo elemento. Si la expresión es usada como el último elemento de la lista de expresiones, entonces no se realiza ningún ajuste, (a menos que la ultima expresión este encerrada entre paréntesis).

Como ejemplo, considerar las siguientes definiciones:

```
function f(a, b) { };
function g(a, b, ...) { };
function r() { return 1,2,3 };
```

La siguiente tabla que relaciona los argumentos con sus parámetros:

```
LLAMADA          PARAMETROS

f(3)             a=3, b=null
f(3, 4)          a=3, b=4
f(3, 4, 5)       a=3, b=4
f(r(), 10)       a=1, b=10
f(r())           a=1, b=2

g(3)             a=3, b=null, ... -->  (nothing)
g(3, 4)          a=3, b=4,    ... -->  (nothing)
g(3, 4, 5, 8)    a=3, b=4,    ... -->  5  8
g(5, r())        a=5, b=1,    ... -->  2  3
```

Los resultados son devueltos usando la expresión `return`.
Si el flujo del programa alcanza el final de la función sin encontrar una sentencia de retorno, entonces la función retorna sin resultados.

El token (`::`) es usado para definir y llamar a métodos de tablas. Cuando se usa este token las funciones reciben un parámetro extra implícito `this`.

La expresión:

```
function t.a.b.c::f(params) { }
```

Es idéntica a:

```
t.a.b.c.f = function(this, params) { } // this is t.a.b.c
```

2.5.12 Declaraciones locales

Killa es un lenguaje de alcance de definición lexicográfico (también conocido como estático). El alcance de las variables empieza desde la primera sentencia después de su declaración y hasta el final del bloque que contiene la declaración.

Considerar el siguiente ejemplo:

```
public var x = 10     // global variable
{                     // new block
    var x = global.x  // new 'x', with value 10
    print(x)          // 10
    x = x + 1
    {                 // another block
        var x = x + 1 // another 'x'
        print(x)      // 12
    }
    print(x)          // 11
}
print(global.x)       // 10  (the global one)
```

Véase que en una declaración como `var x = global.x`, el nuevo `x` que esta siendo declarado no esta dentro de su ámbito de definición aún y es necesario hacer inicializarla con la variable global externa.

Debido al alcance lexicográfico, las variables locales pueden ser libremente accedidas por funciones definidas dentro de su alcance. Una variable local usada por una función interna es llamada variable externa local, dentro de la función interna.

Nótese que cada ejecución de la sentencia var define nuevas variables.
Considérese el siguiente ejemplo:

```
var closures = []
var cx = 20
for (var i = 0; i < 10; i += 1) {
    var y = 0
    closures[i] = function() { y += 1; return cx + y }
}

print("c[0](): "..closures[0]())  // prints: c[0](): 21
print("c[0](): "..closures[0]())  // prints: c[0](): 22
print("c[0](): "..closures[0]())  // prints: c[0](): 23
print("c[1](): "..closures[1]())  // prints: c[1](): 21
cx = 200
print("c[0](): "..closures[0]())  // prints: c[0](): 204
print("c[1](): "..closures[1]())  // prints: c[1](): 202
```

Este bucle creará diez *closures* (esto es, diez instancias de una función anónima). Cada uno de esos *closures* usa una variable `'y'` diferente, mientras todas ellas comparten la misma variable 'cx'.

Las variables y funciones globales (`public`) pueden ser accedidas desde cualquier *script*. Las variables y funciones locales (`private`) solo pueden ser accedidas desde su script o bloque correspondiente.

En Killa, a diferencia de Lua, las variables globales necesitan ser llamadas usando el símbolo `global`. Las funciones globales no requieren esto debido a que cuando se llama una función global que no existe se dará un error en tiempo de ejecución, algo que no ocurre en Lua si es una variable global pues en esta se crea una nueva variable global con valor `null` dando lugar a errores silenciosos.

2.5.13 Expresiones condicionales ternarias

Killa soporta el operador condicional ternario (?).

TernaryExp ::= '(' *ExpreConditional* ')?' *TrueExpr* ':' *FalseExpr*

Su uso es similar a otros lenguajes donde existe sin embargo Killa requiere el uso de los paréntesis antes del carácter (?). Esto es por diseño del lenguaje para evitar los errores que suceden en otros lenguajes con el operador ternario debido al desconocimiento de la precedencia de dicho operador.

En el siguiente ejemplo, véase como la función `falseExpr` no es evaluada y es posible anidar múltiples expresiones condicionales ternarias:

```
function trueExpr() {
    print("trueExpr")
    return 10
}
function falseExpr() {
    print("falseExpr")
    return 9
}

var y = (true)? trueExpr(): falseExpr() // falseExpr is not called
var x = (y < 6)? "less than 6" : (y % 2 == 0)? "even" : "odd"

print(x) // Output: even
```

3. Capítulo III: Diferencias y ventajas de Killa

3.1 Sintaxis basada en C similar a JavaScript

Considérese el ejemplo inicial mostrado en Lua para calcular el promedio de valores de una lista, el mismo ejemplo en Killa es más fácil de entender para programadores experimentados en C++ y Java:

```
function getAverage(array) {
    var sum = 0
    var length = $array
    if (length < 1) {
        return null
    }
    for (var k = 0; k < length; k += 1) {
        sum += array[k]
    }
    return sum / length
}
print(getAverage([10, 20, 40, 50])) // prints: 30
print(getAverage([]))               // prints: null
```

Adicionalmente el código anterior es en gran parte JavaScript válido, usando el intérprete JavaScript NodeJs [10] los cambios a realizar para convertirlo en un programa JavaScript válido son modificar la obtención del tamaño del array con `array.length` (en vez de `$array`) y renombrar la función de impresión de resultados en pantalla de `print` a `console.log`:

```
function getAverage(array) {
    var sum = 0
    var length = array.length
    if (length < 1) {
        return null
    }
    for (var k = 0; k < length; k += 1) {
        sum += array[k]
    }
    return sum / length
}

console.log(getAverage([10, 20, 40, 50])) // prints: 30
console.log(getAverage([]))               // prints: null
```

Sin embargo aunque Killa tenga una gran similitud con JavaScript, debe reafirmarse el hecho de que no es JavaScript y no pretende serlo, siendo la similitud entre ambos lenguajes válida solo para facilitar el desarrollo de programas en Killa.

3.2 Declaración de variables obligatoria

En Lua y JavaScript son típicos los errores por escribir mal el nombre de una variable. En dichos casos una nueva variable global es creada y dicha variable es inicializada con valor nulo. Esto permite la introducción de errores silenciosos.

Como un ejemplo considérese el siguiente fragmento de código Lua:

```
else
    local t = type(value)
    if t == "userdata" and value.type then t = valye.type end
    error("Cannot send value (unsupported type: "..t..").")
end
```

El código arriba mostrado contiene un error debido a una variable mal escrita. Aunque relativamente inocuo, este ejemplo ha sido sacado de un proyecto de código abierto con años de desarrollo y decenas de contribuidores que sin embargo fue descubierto casualmente cuando el autor de esta tesis migró el código de Lua a Killa [11]. El error esta en la tercera línea:

```
if t == "userdata" and value.type then t = value.type end
```

Se estaba creando una variable global 'valye'. Así como este, son muchos los errores que el autor de esta tesis cometió creando su código de ejemplo pero que el compilador de Killa detectó en tiempo de compilación, permitiendo ahorrar tiempo en búsqueda y corrección de errores. Esta es una de las ventajas importantes de Killa con respecto a Lua y JavaScript.

3.3 Minimización de variables globales

Lua y JavaScript soportan declaraciones locales de bloque (Lua) y función (JavaScript) pero es necesario especificar la palabra local (o var) para ello, cosa que algunos programadores prefieren evitar por ahorrar algunos caracteres de digitado, lo que facilita y fomenta el uso de variables globales. Killa por defecto trata todas las funciones y variables declaradas como locales o privadas, es decir hace lo mas seguro por defecto. Definir y usar variables globales es más costoso, forzando a los programadores a disminuir su uso.

3.4 Bloques con llaves obligatorias

En JavaScript, las llaves en bloques son opcionales, pero esto facilita el desarrollo de código que puede ser peligroso, por ejemplo:

```
if (someConditional)
     doThis()
```

Podría convertirse en:

```
if (someConditional)
    doThis()
    doAlsoThis() // This is not part of the if statement
```

Killa requiere el uso obligatorio de llaves y sigue el estándar JSLint [12].

3.5 Operador ternario

Lua no tiene el operador ternario que es muy útil para simplificar expresiones de asignación. En Killa este operador ha sido reforzado con el requerimiento de estar delimitado por paréntesis para el condicional, para evitar problemas de prioridad con otros operadores. Es algo bastante reclamado en Lua pero lamentablemente no implementado por los autores.

3.6 Asignaciones compuestas

En Lua no están implementados los operadores (+=), (–=), etc., esto hace el código de incrementar o disminuir una variable mas verboso y largo de lo que debería ser, Killa permite este tipo de expresiones y simplifica código redundante.

3.7 Conversiones automáticas controladas

En JavaScript existe un caos total con las conversiones de cadena a numero o numero a cadena, arrays a valores lógicos o cadenas a valores lógicos, estas conversiones suceden si se usan los operadores de comparación de igualdad (==) y desigualdad (!=) siendo por ello necesario usar los operadores de comparación estrictos (===) y (!==) que no convierten entre los tipos de datos. Killa usa el concepto de operadores separados para concatenación y adición de Lua y no permite la conversión de tipos en expresiones de comparación, eliminando de raíz todos los errores que se pueden producir en JavaScript.

3.8 Arrays con índice en base 0

En el mundo de JavaScript, C/C++ o Java, los índices de arrays empiezan en cero, los creadores de Lua consideraron que empezar los índices en base 1 era mas natural para sus usuarios, lamentablemente esto no es así para el resto de programadores experimentados en esos lenguajes (que son muy usados) los cuales encuentran esta convención extraña y difícil de comprender, solo aceptable porque no hay otra opción. Killa usa arrays con base cero. Adicionalmente Killa permite el uso de base 1 cambiando solo una línea de configuración, esto se permite para poder portar código de Lua a Killa y descartar errores debidos a cambio de índice.

3.9 Ligero para incrustación

Debido al auge de JavaScript muchos han considerado usarlo para la incrustación en sistemas grandes (el motor de videojuegos 3D Unity [13] es un ejemplo). Pero Javascript aun en sus mejores implementaciones (V8) es enorme y difícil de usar comparado con Killa (o Lua). Killa permite tener lo mejor de la sintaxis de JavaScript sin necesidad de pagar un alto precio de integración.

4. Capítulo IV: Implementación

El código completo de Killa es su versión actual consta de alrededor de 21,000 líneas de código ANSI-C y ha sido desarrollado usando como base el código fuente de Lua 5.2. Debido a ello preferimos no incluirlo como apéndice en esta tesis sino indicar la dirección del cual puede descargarse.

https://github.com/ex/Killa

Al igual que Lua, Killa tiene una licencia de código abierto y permisivo MIT, lo cual lo hace ideal para el desarrollo de videojuegos comerciales sin restricciones.

4.1 Componentes principales de Lua

Una descripción resumida de la implementación de Lua puede encontrarse en el informe que los autores de Lua hicieron de la versión 5.0 [14]. Un diagrama de resumen se muestra a continuación:

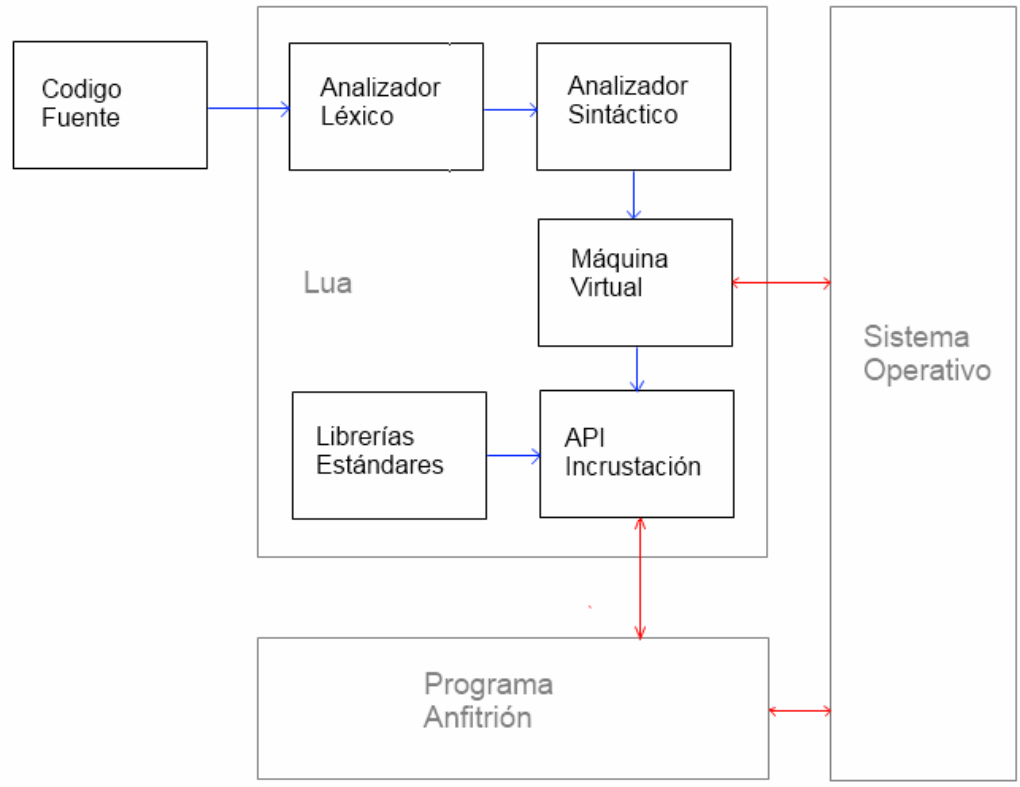

FIGURA 4.1: Esquema de componentes de Lua

La implementación de Killa se centró principalmente en la modificación del analizador léxico y sintáctico de Lua aunque ciertos cambios también fueron necesarios en las librerías estándar y la máquina virtual de Lua para poder soportar índices de base cero.

Debido a que era necesario tener la posibilidad de usar Lua y Killa al mismo tiempo en los programas anfitriones, se renombró todos los símbolos de `lua` a `killa` para evitar colisiones de nombres. De este modo es posible tener una sola versión del código del programa anfitrión con soporte de Lua o Killa al mismo tiempo usando compilación condicional.

4.2 Incrustación de Killa

El plan para poder incrustar fácilmente Killa en programas anfitriones que ya soportaban Lua fue el de no modificar en absoluto la interfaz de aplicaciones de Lua, Killa usa exactamente el mismo API de incrustación de Lua aunque renombrada para evitar conflictos. Para soportar los nombres de Killa un archivo de traducción `klua.h` es empleado el cual se encarga de realizar las redefiniciones usando el pre-procesador de C/C++.

De esta manera el programa anfitrión es en gran parte indiferente al hecho de estar usando Killa o Lua, como se muestra en el siguiente gráfico:

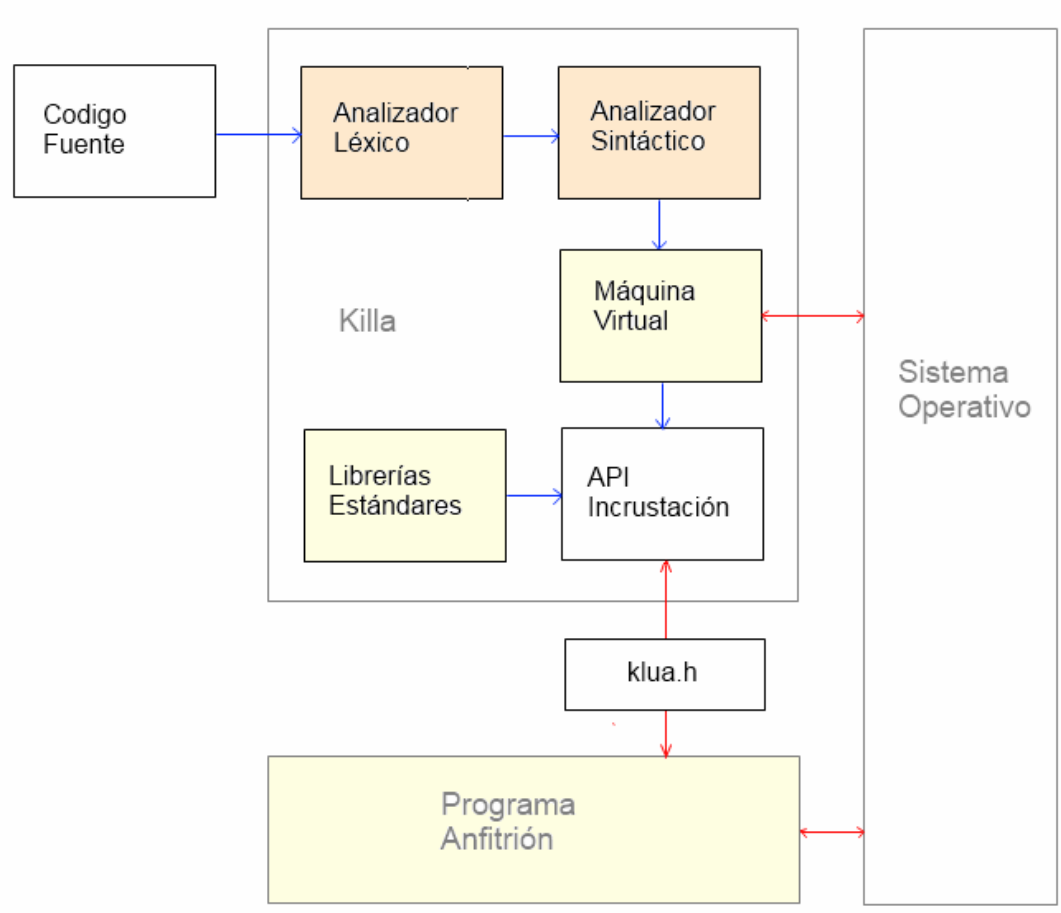

FIGURA 4.2: Esquema de interfaz de incrustación de Killa

Incrustar Killa en programas C++ que soporten Lua no es muy complicado si se usa el archivo de redefiniciones `klua.h` que es anexado en el apéndice A.

El hecho de que Lua usa índices base uno y Killa usa índices base cero hace necesario cambios en ciertas partes del programa anfitrión que usen índices, dichos cambios deben ser controlados usando el símbolo de pre-procesador KILLA_BASE que es la base que usa Killa para índices de array como puede verse en el siguiente extracto de código C++ del programa Love2D:

```
#ifdef LOVE_USE_LUA
            lua_rawseti(L, -2, i + 1);
#else
            lua_rawseti(L, -2, i + KILLA_BASE);
#endif
```

Killa ha sido incrustado con éxito en 2 motores de videojuegos modernos de código abierto y de gran uso, como son: Love2D [15] y cocos2d-x [16].

Usando el motor de videojuegos Love2D se ha realizado un simple clon de Tetris programado enteramente en Killa el cual es jugable y es anexado completamente en el apéndice B.

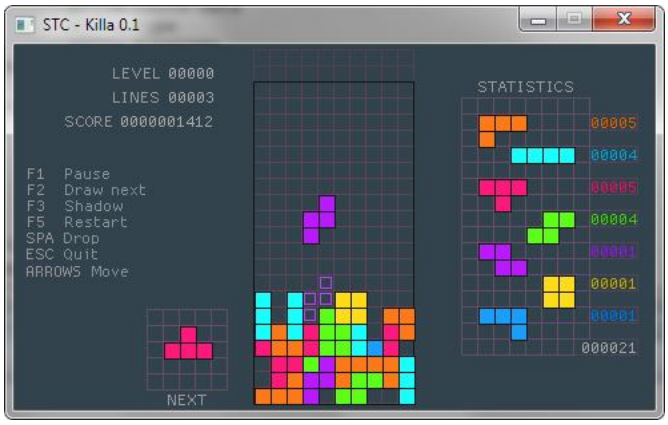

FIGURA 4.3: Captura de pantalla de Killa incrustado en Love2D

Este simple ejemplo demuestra como usar Killa para dibujar imágenes en pantalla, controlar el programa con el teclado y reproducir música y efectos de sonido.

Usando Killa con el motor de videojuegos cocos2d-x fue posible también desplegar programas escritos en Killa en dispositivos móviles como iPhone y Android. Un simple visor del fractal de Mandelbrot ha sido realizado como ejemplo de lo que es posible y se anexa en el apéndice C:

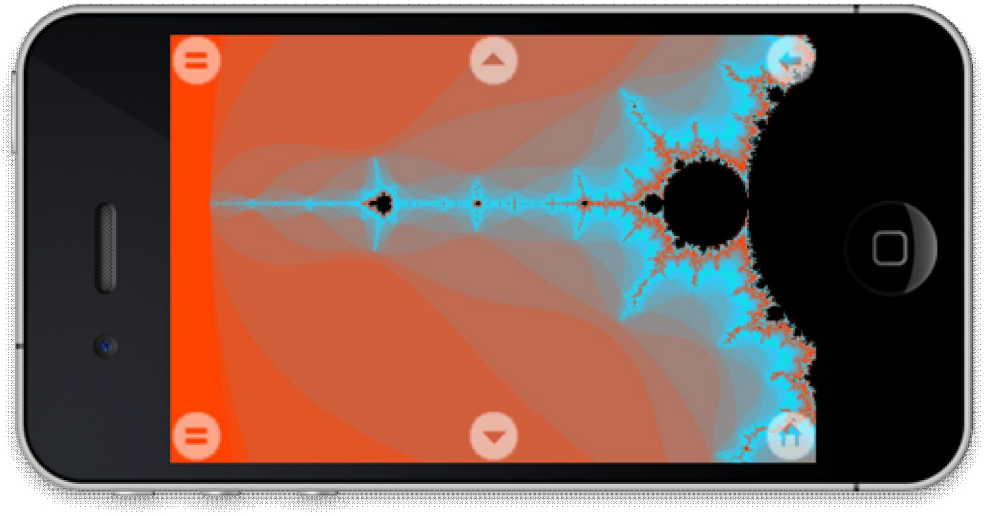

FIGURA 4.4: Killa incrustado en cocos2d-x para dispositivos móviles

4.3 Dificultades encontradas

Debido a que el autor de esta tesis tenía experiencia previa usando Lua en proyectos comerciales muchos de los cambios a realizar fueron relativamente sencillos de hacer, en especial los cambios relacionados con el analizador léxico.

Quizás la parte más complicada del proyecto haya sido la modificación de la base de los índices de array para que empiecen en cero y no en uno, debido a que el código de Lua no esta estructurado para lograr esto fácilmente. Como se dijo anteriormente, Killa permite también usar índices de base uno cambiando sólo una línea de configuración lo cual es útil para poder migrar código existente de Lua a Killa o probar la incrustación de Killa en otros programas que usen Lua y descartar problemas debido a los índices de array.

Esto característica fue muy útil cuando se migró el código de los scripts de inicio de Love2D a Killa pues primero se uso el código traducido de Lua a Killa pero usando Killa con índices de base 1 y solo cuando todo estuvo funcionando correctamente se cambio el código para usar índices de base cero.

5. Observaciones, conclusiones y recomendaciones

Esta tesis ha presentado el origen y desarrollo de Killa, la descripción de la sintaxis del lenguaje además de ejemplos de código, sus ventajas frente a lenguajes como Lua y JavaScript, además de la implementación de ejemplos incrustados en motores de videojuegos modernos. Se ha preferido no incluir en este informe los conceptos de meta programación que son soportados por Killa debido a su herencia Lua, pues estos serán remplazados por programación orientada a objetos en siguientes versiones del lenguaje.

5.1 Trabajo futuro

Killa es un lenguaje completamente funcional y practico en su estado actual pero aún quedan muchas cosas por hacer para poder mejorarlo, como agregar tipos de datos opcionales, un sistema de clases, excepciones, compilación condicional, compilación JIT y macros en tiempo de compilación entre otras cosas lo cual da amplio espacio para posibles trabajos futuros.

5.1.1 Tipos de datos condicionales

Lenguajes modernos como Haxe [16] y ActionScript [17] soportan declaración de tipos opcionales, lo cual ofrece cierta ventaja en el desarrollo de aplicaciones grandes pues evita errores que pueden ser descubiertos en la fase de compilación.

```
var age:int = 25              // type int
var man:Entity = new Entity() // type object
age = man                     // this would fail
```

5.1.2 Soporte de clases

El desarrollo de programas complejos requiere medios de abstracción y ocultamiento de la información, algo que la programación orientada a objetos permite. En Killa se piensa implementar un sistema de clases parecido al de Java o Haxe sin permitir herencia múltiple pero con soporte de múltiples interfaces. El sistema de metatablas heredado de Lua puede seguir siendo usado en paralelo para implementar sistemas alternativos de objetos basados en prototipos.

5.1.3 Compilación JIT

La búsqueda del máximo rendimiento de las maquinas virtuales ha obligado a crear variantes de Lua [18] que usan compilación a instrucciones de procesador en tiempo de ejecución: *Just In Time*. Dichas mejoras pueden ser implementadas en Killa para obtener un mayor rendimiento.

5.1.4 Macros en tiempo de compilación

Lenguajes como Lisp [19] y Haxe soportan macros que permiten modificar y agregar características nuevas al lenguaje en tiempo de compilación. Este soporte existe de manera bastante primitiva en C a través del pre-procesador y en C++ a través del uso de plantillas. Killa debería tener algún tipo de soporte de macros así como un pre-procesador para compilación condicional.

6. Referencias

1. J. K. Ousterhout.
1998 Scripting: Higher-Level Programming for the 21st Century.
Computer, 31(3):23–30

2. TIOBE Software BV
2012 TIOBE Programming Community Index for may 2012
Consulta: 1 de junio 2012
<http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>

3. A. Johnstone, P. Mosses, and E. Scott.
2010 An Agile Approach to Language Modelling and Development.
Innovations in Systems and Software Engineering, 6: 145--153

4. Roberto Ierusalimschy , Luiz Henrique de Figueiredo , Waldemar Celes Filho
1996 Lua - an extensible extension language
Software - Practice & Experience, v.26 n.6, p.635-652

5. Lua community
2012 Lista de proyectos que usan Lua
Consulta: 1 de junio 2012
<http://lua-users.org/wiki/LuaUses>

6. Roberto Ierusalimschy , Luiz Henrique de Figueiredo , Waldemar Celes
2007 The evolution of Lua
Proceedings of the third ACM SIGPLAN
Conference on History of programming languages, p.2-1-2-26,
June 09-10, San Diego, California

7. Wikipedia
2012 JavaScript Language
Consulta: 1 de junio 2012
<http://en.wikipedia.org/wiki/Javascript>

8. Edsger W. Dijkstra
1982 Why numbering should start at zero, 11 August
Consulta: 1 de junio 2012
<http://www.cs.utexas.edu/users/EWD/transcriptions/EWD08xx/EWD831.html>

9. Terry Moore
2008 Bright - Yet Another Lua Derivative [diapositivas] Washington, D.C
Consulta: 1 de junio 2012
<http://www.lua.org/wshop08.html#moore>

10. Joyent, Inc
2012 NodeJs
Consulta: 1 de junio 2012
<http://nodejs.org/>

11. Atlasian
2012 Revisión de código en Love2D debido a un error de digitado
Consulta: 1 de junio 2012
<https://bitbucket.org/rude/love/changeset/f2da62fae3e7>

12. Douglas Crockford
2012 Conventions for the JavaScript Programming Language
Consulta: 1 de junio 2012
<http://javascript.crockford.com/code.html>

13. Unity Technologies
2012 Unity 3D engine
Consulta: 1 de junio 2012
<http://unity3d.com/>

14. Roberto Ierusalimschy , Luiz Henrique de Figueiredo , Waldemar Celes
2005 The Implementation of Lua 5.0
Journal of Universal Computer Science, vol. 11, no. 7, 1159-1176

15. Love community
2012 Love2D engine
Consulta: 1 de junio 2012
<http://love2d.org/>

16. Cocos2d community
2012 cocos2d-x
Consulta: 1 de junio 2012
<http://www.cocos2d-x.org/>

17. Haxe community
2012 Haxe Language
Consulta: 1 de junio 2012
<http://haxe.org/>

18. John McCarthy
1960 Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I.
Massachusetts Institute of Technology, Cambridge

19. W. Wulf, Mary Shaw
1973 Global variable considered harmful.
ACM SIGPLAN Notices Homepage archive, Pages 28 - 34

20. Ricky E. Sward
2004 Re-engineering global variables in Ada.
ACM SIGAda Ada Letters Homepage
Volume XXIV Issue 4, December 2004, Pages 29 - 34

Apéndice A: klua.h

```
/* ========================================================================== */
/*   Lua to Killa                                                             */
/* -------------------------------------------------------------------------- */
/*   Copyright (c) 2012 Laurens Rodriguez Oscanoa.                            */
/*                                                                            */
/*   This code is licensed under the MIT license:                             */
/*   http://www.opensource.org/licenses/mit-license.php                       */
/* -------------------------------------------------------------------------- */

#ifndef KLUA_H_
#define KLUA_H_

#ifndef KILLA_COMPAT_ALL
#define KILLA_COMPAT_ALL
#endif

#define LUA_VERSION_NUM           501

#include <killa.h>
#include <killalib.h>
#include <kauxlib.h>


#define LUA_ERRMEM          KILLA_ERRMEM
#define LUA_ERRRUN          KILLA_ERRRUN
#define LUA_ERRSYNTAX       KILLA_ERRSYNTAX
#define LUA_GCCOLLECT       KILLA_GCCOLLECT
#define LUA_MULTRET         KILLA_MULTRET
#define LUA_REFNIL          KILLA_REFNIL
#define LUA_REGISTRYINDEX   KILLA_REGISTRYINDEX
#define LUA_TBOOLEAN        KILLA_TBOOLEAN
#define LUA_TFUNCTION       KILLA_TFUNCTION
#define LUA_TLIGHTUSERDATA  KILLA_TLIGHTUSERDATA
#define LUA_TNONE           KILLA_TNONE
#define LUA_TNUMBER         KILLA_TNUMBER
#define LUA_TSTRING         KILLA_TSTRING
#define LUA_TTABLE          KILLA_TTABLE
#define LUA_TUSERDATA       KILLA_TUSERDATA

#define lua_call            killa_call
#define lua_close           killa_close
#define lua_concat          killa_concat
#define lua_createtable     killa_createtable
#define lua_CFunction       killa_CFunction
#define lua_error           killa_error
#define lua_gc              killa_gc
#define lua_getfenv         killa_getuservalue
#define lua_getfield        killa_getfield
#define lua_getglobal       killa_getglobal
#define lua_getmetatable    killa_getmetatable
#define lua_gettable        killa_gettable
#define lua_gettop          killa_gettop
#define lua_insert          killa_insert
#define lua_Integer         killa_Integer
#define lua_isboolean       killa_isboolean
#define lua_iscfunction     killa_iscfunction
#define lua_isfunction      killa_isfunction
#define lua_islightuserdata killa_islightuserdata
#define lua_islightuserdata killa_islightuserdata
#define lua_isnil           killa_isnull
#define lua_isnumber        killa_isnumber
#define lua_isnoneornil     killa_isnoneornull
#define lua_isstring        killa_isstring
#define lua_istable         killa_istable
#define lua_isuserdata      killa_isuserdata
#define lua_newtable        killa_newtable
#define lua_next            killa_next
#define lua_newuserdata     killa_newuserdata
#define lua_Number          killa_Number
#define lua_objlen          killa_objlen
#define lua_open            killaL_newstate
#define lua_pcall           killa_pcall
#define lua_pop             killa_pop
#define lua_pushboolean     killa_pushboolean
#define lua_pushcclosure    killa_pushcclosure
#define lua_pushcfunction   killa_pushcfunction
```

```
#define lua_pushfstring     killa_pushfstring
#define lua_pushinteger     killa_pushinteger
#define lua_pushlightuserdata   killa_pushlightuserdata
#define lua_pushliteral     killa_pushliteral
#define lua_pushlstring     killa_pushlstring
#define lua_pushnil         killa_pushnull
#define lua_pushnumber      killa_pushnumber
#define lua_pushstring      killa_pushstring
#define lua_pushvalue       killa_pushvalue
#define lua_rawequal        killa_rawequal
#define lua_rawget          killa_rawget
#define lua_rawgeti         killa_rawgeti
#define lua_rawset          killa_rawset
#define lua_rawseti         killa_rawseti
#define lua_replace         killa_replace
#define lua_remove          killa_remove
#define lua_setfenv         killa_setuservalue
#define lua_setfield        killa_setfield
#define lua_setglobal       killa_setglobal
#define lua_setmetatable    killa_setmetatable
#define lua_settable        killa_settable
#define lua_settop          killa_settop
#define lua_State           killa_State
#define lua_strlen          killa_strlen
#define lua_toboolean       killa_toboolean
#define lua_tocfunction     killa_tocfunction
#define lua_tointeger       killa_tointeger
#define lua_tolstring       killa_tolstring
#define lua_tonumber        killa_tonumber
#define lua_tostring        killa_tostring
#define lua_touserdata      killa_touserdata
#define lua_type            killa_type
#define lua_typename        killa_typename
#define lua_upvalueindex    killa_upvalueindex

#define luaL_addlstring     killaL_addlstring
#define luaL_addstring      killaL_addstring
#define luaL_argcheck       killaL_argcheck
#define luaL_argerror       killaL_argerror
#define luaL_Buffer         killaL_Buffer
#define luaL_buffinit       killaL_buffinit
#define luaL_checkint       killaL_checkint
#define luaL_checklstring   killaL_checklstring
#define luaL_checkinteger   killaL_checkinteger
#define luaL_checknumber    killaL_checknumber
#define luaL_checkstring    killaL_checkstring
#define luaL_checkudata     killaL_checkudata
#define luaL_dofile         killaL_dofile
#define luaL_dostring       killaL_dostring
#define luaL_error          killaL_error
#define luaL_getmetafield   killaL_getmetafield
#define luaL_getmetatable   killaL_getmetatable
#define luaL_loadbuffer     killaL_loadbuffer
#define luaL_loadstring     killaL_loadstring
#define luaL_newmetatable   killaL_newmetatable
#define luaL_openlib        killaL_openlib
#define luaL_openlibs       killaL_openlibs
#define luaL_optint         killaL_optint
#define luaL_optlstring     killaL_optlstring
#define luaL_optnumber      killaL_optnumber
#define luaL_optstring      killaL_optstring
#define luaL_putchar        killaL_addchar
#define luaL_pushresult     killaL_pushresult
#define luaL_ref            killaL_ref
#define luaL_reg            killaL_Reg
#define luaL_Reg            killaL_Reg
#define luaL_register       killaL_register
#define luaL_typename       killaL_typename
#define luaL_typerror       killaL_typerror
#define luaL_unref          killaL_unref

#endif
```

Apéndice B: Juego de Tetris en Killa

```
/* ========================================================================== */
/*   game.kia                                                                 */
/*   Game logic implementation.                                               */
/* -------------------------------------------------------------------------- */

// Local reference to Platform instance.
var Platform

// Initial time delay (in milliseconds) between falling moves.
var INIT_DELAY_FALL = 1000

// Score points given by filled rows (we use the original NES * 10)
// http://tetris.wikia.com/wiki/Scoring
var SCORE_1_FILLED_ROW = 400
var SCORE_2_FILLED_ROW = 1000
var SCORE_3_FILLED_ROW = 3000
var SCORE_4_FILLED_ROW = 12000

// The player gets points every time he accelerates downfall.
// The added points are equal to SCORE_2_FILLED_ROW divided by this value.
var SCORE_MOVE_DOWN_DIVISOR = 1000

// The player gets points every time he does a hard drop.
// The added points are equal to SCORE_2_FILLED_ROW divided by these
// values. If the player is not using the shadow he gets more points.
var SCORE_DROP_DIVISOR             = 20
var SCORE_DROP_WITH_SHADOW_DIVISOR = 100

// Number of filled rows required to increase the game level.
var FILLED_ROWS_FOR_LEVEL_UP = 10

// The falling delay is multiplied and divided by
// these factors with every level up.
var DELAY_FACTOR_FOR_LEVEL_UP = 9
var DELAY_DIVISOR_FOR_LEVEL_UP = 10

// Delayed autoshift initial delay.
var DAS_DELAY_TIMER = 200

// Delayed autoshift timer for left and right moves.
var DAS_MOVE_TIMER = 40

// Rotation auto-repeat delay.
var ROTATION_AUTOREPEAT_DELAY = 375

// Rotation autorepeat timer.
var ROTATION_AUTOREPEAT_TIMER = 200

var NULL = 0

var Game = {
    // Playfield size (in tiles).
    BOARD_TILEMAP_WIDTH  : 10;
    BOARD_TILEMAP_HEIGHT : 22;

    // Error codes.
    Error : {
        NONE         :  0,   // Everything is OK, oh wonders!
        PLAYER_QUITS :  1,   // The user quits, our fail
        NO_MEMORY    : -1,   // Not enough memory
        NO_VIDEO     : -2,   // Video system was not initialized
        NO_IMAGES    : -3,   // Problem loading the image files
        PLATFORM     : -4,   // Problem creating platform
        ASSERT       : -100  // Something went very very wrong...
    };

    // Game events.
    Event : {
        NONE        : 0,
        MOVE_DOWN   : 1,
        MOVE_LEFT   : 2,
        MOVE_RIGHT  : 4,
        ROTATE_CW   : 8,    // rotate clockwise
        ROTATE_CCW  : 16,   // rotate counter-clockwise
        DROP        : 32,
        PAUSE       : 64,
```

```
        RESTART     : 128,
        SHOW_NEXT   : 256,  // toggle show next tetromino
        SHOW_SHADOW : 512,  // toggle show shadow
        QUIT        : 1024  // finish the game
    };

    // We are going to store the tetromino cells in a square matrix
    // of this size (this is the size of the biggest tetromino).
    TETROMINO_SIZE: 4;

    // Number of tetromino types.
    TETROMINO_TYPES: 7;

    // Tetromino definitions.
    // They are indexes and must be between: 0 - [TETROMINO_TYPES - 1]
    // http://tetris.wikia.com/wiki/Tetromino
    // Initial cell disposition is commented below.
    TetrominoType: {
        //      ....
        //      ####
        //      ....
        //      ....
        I : 0,
        //      ##..
        //      ##..
        //      ....
        //      ....
        O : 1,
        //      .#..
        //      ###.
        //      ....
        //      ....
        T : 2,
        //      .##.
        //      ##..
        //      ....
        //      ....
        S : 3,
        //      ##..
        //      .##.
        //      ....
        //      ....
        Z : 4,
        //      #...
        //      ###.
        //      ....
        //      ....
        J : 5,
        //      ..#.
        //      ###.
        //      ....
        //      ....
        L : 6
    };

    // Color indexes.
    Cell: {
        EMPTY  : -1, // This value used for empty tiles.
        CYAN   : 1,
        RED    : 2,
        BLUE   : 3,
        ORANGE : 4,
        GREEN  : 5,
        YELLOW : 6,
        PURPLE : 7,
        WHITE  : 0  // Used for effects (if any)
    };

    COLORS: 8;

    // Create data structure that holds information about our tetromino blocks.
    createTetromino: function() {
        var tetromino = {
            cells: [];  // Tetromino buffer
            x: 0;
            y: 0;
            size: 0;
            type: 0
        }
        return tetromino
    };
```

```
    // Create data structure for statistical data.
    createStatics: function () {
        var stats = {
            score: 0;       // user score for current game
            lines: 0;       // total number of lines cleared
            totalPieces: 0; // total number of tetrominoes used
            level: 0;       // current game level
            pieces: []      // number of tetrominoes per type
        }
        return stats
    };

    // Game events are stored in bits in this variable.
    // It must be cleared to Game.Event.NONE after being used.
    m_events:       0;

    // Matrix that holds the cells (tilemap)
    m_map:          NULL;

    m_stats:        NULL;  // statistic data
    m_fallingBlock: NULL;  // current falling tetromino
    m_nextBlock:    NULL;  // next tetromino

    m_stateChanged: false; // true if game state has changed
    m_errorCode:    0;     // stores current error code
    m_isPaused:     false; // true if the game is over
    m_isOver:       false; // true if the game is over
    m_showPreview:  true;  // true if we must show the preview block

    m_showShadow:   true;  // true if we must show the shadow block
    m_shadowGap:    0;     // distance between falling block and shadow

    m_systemTime:   0; // system time in milliseconds
    m_fallingDelay: 0; // delay time for falling tetrominoes
    m_lastFallTime: 0; // last time the falling tetromino dropped

    // For delayed autoshift: http://tetris.wikia.com/wiki/DAS
    m_delayLeft:     -1;
    m_delayRight:    -1;
    m_delayDown:     -1;
    m_delayRotation: -1;
}

// The platform must call this method after processing a changed state.
function Game::onChangeProcessed()   { this.m_stateChanged = false }

// Return the cell at the specified position.
function Game::getCell(column, row)  { return this.m_map[column][row] }

// Return true if the game state has changed, false otherwise.
function Game::hasChanged()          { return this.m_stateChanged }

// Return a reference to the game statistic data.
function Game::stats()               { return this.m_stats }

// Return current falling tetromino.
function Game::fallingBlock()        { return this.m_fallingBlock }

// Return next tetromino.
function Game::nextBlock()           { return this.m_nextBlock }

// Return current error code.
function Game::errorCode()           { return this.m_errorCode }

// Return true if the game is paused, false otherwise.
function Game::isPaused()            { return this.m_isPaused }

// Return true if we must show preview tetromino.
function Game::showPreview()         { return this.m_showPreview }

// Return true if we must show ghost shadow.
function Game::showShadow()          { return this.m_showShadow }

// Return height gap between shadow and falling tetromino.
function Game::shadowGap()           { return this.m_shadowGap }
```

```
// Set matrix elements to indicated value.
function setMatrixCells(matrix, width, height, value) {
    for (var i = 0; i < width; i += 1) {
        matrix[i] = []
        for (var j = 0; j < height; j += 1) {
            matrix[i][j] = value
        }
    }
}

// Bit flags utility helpers.
function isFlagSet(set, flag) {
    return (set % (2*flag) >= flag)
}

function setFlag(set, flag) {
    if (set % (2*flag) >= flag) {
        return set
    }
    return (set + flag)
}

function clearFlag(set, flag) {
    if (set % (2*flag) >= flag) {
        return (set - flag)
    }
    return set
}

// Initialize tetromino cells for every type of tetromino.
function Game::setTetromino(indexTetromino, tetromino) {
    // Initialize tetromino cells to empty cells.
    setMatrixCells(tetromino.cells, this.TETROMINO_SIZE, this.TETROMINO_SIZE, this.Cell.EMPTY)

    // Almost all the blocks have size 3.
    tetromino.size = this.TETROMINO_SIZE - 1

    // Initial configuration from: http://tetris.wikia.com/wiki/SRS
    if (indexTetromino == this.TetrominoType.I) {
        tetromino.cells[0][1] = this.Cell.CYAN
        tetromino.cells[1][1] = this.Cell.CYAN
        tetromino.cells[2][1] = this.Cell.CYAN
        tetromino.cells[3][1] = this.Cell.CYAN
        tetromino.size = this.TETROMINO_SIZE
    }
    else if (indexTetromino == this.TetrominoType.O) {
        tetromino.cells[0][0] = this.Cell.YELLOW
        tetromino.cells[0][1] = this.Cell.YELLOW
        tetromino.cells[1][0] = this.Cell.YELLOW
        tetromino.cells[1][1] = this.Cell.YELLOW
        tetromino.size = this.TETROMINO_SIZE - 2
    }
    else if (indexTetromino == this.TetrominoType.T) {
        tetromino.cells[0][1] = this.Cell.PURPLE
        tetromino.cells[1][0] = this.Cell.PURPLE
        tetromino.cells[1][1] = this.Cell.PURPLE
        tetromino.cells[2][1] = this.Cell.PURPLE
    }
    else if (indexTetromino == this.TetrominoType.S) {
        tetromino.cells[0][1] = this.Cell.GREEN
        tetromino.cells[1][0] = this.Cell.GREEN
        tetromino.cells[1][1] = this.Cell.GREEN
        tetromino.cells[2][0] = this.Cell.GREEN
    }
    else if (indexTetromino == this.TetrominoType.Z) {
        tetromino.cells[0][0] = this.Cell.RED
        tetromino.cells[1][0] = this.Cell.RED
        tetromino.cells[1][1] = this.Cell.RED
        tetromino.cells[2][1] = this.Cell.RED
    }
    else if (indexTetromino == this.TetrominoType.J) {
        tetromino.cells[0][0] = this.Cell.BLUE
        tetromino.cells[0][1] = this.Cell.BLUE
        tetromino.cells[1][1] = this.Cell.BLUE
        tetromino.cells[2][1] = this.Cell.BLUE
    }
    else if (indexTetromino == this.TetrominoType.L) {
        tetromino.cells[0][1] = this.Cell.ORANGE
        tetromino.cells[1][1] = this.Cell.ORANGE
        tetromino.cells[2][0] = this.Cell.ORANGE
        tetromino.cells[2][1] = this.Cell.ORANGE
```

```
    }
    tetromino.type = indexTetromino
}

// Initialize the game.
function Game::init(platform) {
    Platform = platform

    // Initialize platform.
    Platform::init(this)

    // If everything is OK start the game.
    this::start()
}

// Start a new game.
function Game::start() {
    // Initialize game data.
    this.m_map = []
    this.m_stats = this::createStatics()
    this.m_fallingBlock = this::createTetromino()
    this.m_nextBlock = this::createTetromino()

    this.m_errorCode = this.Error.NONE
    this.m_systemTime = Platform::getSystemTime()
    this.m_lastFallTime = this.m_systemTime
    this.m_isOver = false
    this.m_isPaused = false
    this.m_showPreview = true
    this.m_events = this.Event.NONE
    this.m_fallingDelay = INIT_DELAY_FALL
    this.m_showShadow = true

    // Initialize game statistics.
    for each (var i = 0 to this.TETROMINO_TYPES - 1) {
        this.m_stats.pieces[i] = 0
    }

    // Initialize game tile map.
    setMatrixCells(this.m_map, this.BOARD_TILEMAP_WIDTH, this.BOARD_TILEMAP_HEIGHT,
this.Cell.EMPTY)

    // Initialize falling tetromino.
    this::setTetromino(Platform::random() % this.TETROMINO_TYPES, this.m_fallingBlock)
    this.m_fallingBlock.x = math.floor((this.BOARD_TILEMAP_WIDTH - this.m_fallingBlock.size)
/ 2)
    this.m_fallingBlock.y = 0

    // Initialize preview tetromino.
    this::setTetromino(Platform::random() % this.TETROMINO_TYPES, this.m_nextBlock)

    // Initialize events.
    this::onTetrominoMoved()

    // Initialize delayed autoshift.
    this.m_delayLeft = -1
    this.m_delayRight = -1
    this.m_delayDown = -1
    this.m_delayRotation = -1
}

// Rotate falling tetromino. If there are no collisions when the
// tetromino is rotated this modifies the tetromino's cell buffer.
function Game::rotateTetromino(clockwise) {
    var i, j

    // Temporary array to hold rotated cells.
    var rotated = []

    // If TETROMINO_O is falling return immediately.
    if (this.m_fallingBlock.type == this.TetrominoType.O) {
        // Rotation doesn't require any changes.
        return
    }

    // Initialize rotated cells to blank.
    setMatrixCells(rotated, this.TETROMINO_SIZE, this.TETROMINO_SIZE, this.Cell.EMPTY)

    // Copy rotated cells to the temporary array.
    for (i = 0; i < this.m_fallingBlock.size; i += 1) {
        for (j = 0; j < this.m_fallingBlock.size; j += 1) {
```

```
            if (clockwise) {
                rotated[this.m_fallingBlock.size - j - 1][i]
                                = this.m_fallingBlock.cells[i][j]
            }
            else {
                rotated[j][this.m_fallingBlock.size - i - 1]
                                = this.m_fallingBlock.cells[i][j]
            }
        }
    }

    var wallDisplace = 0

    // Check collision with left wall.
    if (this.m_fallingBlock.x < 0) {
        for (i = 0; (wallDisplace == 0) && (i < -this.m_fallingBlock.x); i += 1) {
            for (j = 0; j < this.m_fallingBlock.size; j += 1) {
                if (rotated[i][j] != this.Cell.EMPTY) {
                    wallDisplace = i - this.m_fallingBlock.x
                    break
                }
            }
        }
    }
    // Or check collision with right wall.
    else if (this.m_fallingBlock.x > this.BOARD_TILEMAP_WIDTH - this.m_fallingBlock.size) {
        for (i = this.m_fallingBlock.size - 1; (wallDisplace == 0)
            && (i >= this.BOARD_TILEMAP_WIDTH - this.m_fallingBlock.x); i -= 1) {

            for (j = 0; j < this.m_fallingBlock.size; j += 1) {
                if (rotated[i][j] != this.Cell.EMPTY) {
                    wallDisplace = -this.m_fallingBlock.x - i + this.BOARD_TILEMAP_WIDTH - 1
                    break
                }
            }
        }
    }

    // Check collision with board floor and other cells on board.
    for (i = 0; i < this.m_fallingBlock.size; i += 1) {
        for (j = 0; j < this.m_fallingBlock.size; j += 1) {
            if (rotated[i][j] != this.Cell.EMPTY) {
                // Check collision with bottom border of the map.
                if (this.m_fallingBlock.y + j >= this.BOARD_TILEMAP_HEIGHT) {
                    // There is a collision therefore return.
                    return
                }
                // Check collision with existing cells in the map.
                if (this.m_map[i + this.m_fallingBlock.x + wallDisplace]
                                [j + this.m_fallingBlock.y]  != this.Cell.EMPTY) {
                    // There is a collision therefore return.
                    return
                }
            }
        }
    }

    // Move the falling piece if there was wall collision and it's a legal move.
    if (wallDisplace != 0) {
        this.m_fallingBlock.x = this.m_fallingBlock.x + wallDisplace
    }

    // There are no collisions, replace tetromino cells with rotated cells.
    for (i = 0; i < this.TETROMINO_SIZE; i += 1) {
        for (j = 0; j < this.TETROMINO_SIZE; j += 1) {
            this.m_fallingBlock.cells[i][j] = rotated[i][j]
        }
    }
    this::onTetrominoMoved()
}

// Check if tetromino will collide with something if it is moved in the requested direction.
// If there are collisions returns 1 else returns 0.
function Game::checkCollision(dx, dy) {

    var newx = this.m_fallingBlock.x + dx
    var newy = this.m_fallingBlock.y + dy

    for each (var i = 0 to this.m_fallingBlock.size - 1) {
        for each (var j = 0 to this.m_fallingBlock.size - 1) {
            if (this.m_fallingBlock.cells[i][j] != this.Cell.EMPTY) {
```

```
                // Check that tetromino would be inside the left, right and bottom borders.
                if ((newx + i < 0) || (newx + i >= this.BOARD_TILEMAP_WIDTH)
                                   || (newy + j >= this.BOARD_TILEMAP_HEIGHT)) {
                    return true
                }
                // Check that tetromino won't collide with existing cells in the map.
                if (this.m_map[newx + i][newy + j] != this.Cell.EMPTY) {
                    return true
                }
            }
        }
    }
    return false
}

// Game scoring: http://tetris.wikia.com/wiki/Scoring
function Game::onFilledRows(filledRows) {
    // Update total number of filled rows.
    this.m_stats.lines = this.m_stats.lines + filledRows

    // Increase score accordingly to the number of filled rows.
    if (filledRows == 1) {
        this.m_stats.score = this.m_stats.score + SCORE_1_FILLED_ROW * (this.m_stats.level +
1)
    }
    else if (filledRows == 2) {
        this.m_stats.score = this.m_stats.score + SCORE_2_FILLED_ROW * (this.m_stats.level +
1)
    }
    else if (filledRows == 3) {
        this.m_stats.score = this.m_stats.score + SCORE_3_FILLED_ROW * (this.m_stats.level +
1)
    }
    else if (filledRows == 4) {
        this.m_stats.score = this.m_stats.score + SCORE_4_FILLED_ROW * (this.m_stats.level +
1)
    }
    else {
        // This shouldn't happen, but if happens kill the game.
        this.m_errorCode = this.Error.ASSERT
    }

    // Check if we need to update the level.
    if (this.m_stats.lines >= FILLED_ROWS_FOR_LEVEL_UP * (this.m_stats.level + 1)) {
        this.m_stats.level = this.m_stats.level + 1

        // Increase speed for falling tetrominoes.
        this.m_fallingDelay = math.floor(DELAY_FACTOR_FOR_LEVEL_UP * this.m_fallingDelay
                                         / DELAY_DIVISOR_FOR_LEVEL_UP)
    }

    Platform::onFilledRows(filledRows)
}

// Move tetromino in the direction specified by (x, y) (in tile units)
// This function detects if there are filled rows or if the move
// lands a falling tetromino, also checks for game over condition.
function Game::moveTetromino(x, y) {
    var i, j

    // Check if the move would create a collision.
    if (this::checkCollision(x, y)) {
        // In case of collision check if move was downwards (y == 1)
        if (y == 1) {
            // Check if collision occurs when the falling
            // tetromino is on the 1st or 2nd row.
            if (this.m_fallingBlock.y <= 1) {
                // If this happens the game is over.
                this.m_isOver = true
            }
            else {
                // The falling tetromino has reached the bottom,
                // so we copy their cells to the board map.
                for (i = 0; i < this.m_fallingBlock.size; i += 1) {
                    for (j = 0; j < this.m_fallingBlock.size; j += 1) {
                        if (this.m_fallingBlock.cells[i][j] != this.Cell.EMPTY) {
                            this.m_map[this.m_fallingBlock.x + i][this.m_fallingBlock.y + j]
                                       = this.m_fallingBlock.cells[i][j]
                        }
                    }
                }
```

```
                // Check if the landing tetromino has created full rows.
                var numFilledRows = 0
                for (j = 1; j < this.BOARD_TILEMAP_HEIGHT; j += 1) {
                    var hasFullRow = true
                    for (i = 0; i < this.BOARD_TILEMAP_WIDTH; i += 1) {
                        if (this.m_map[i][j] == this.Cell.EMPTY) {
                            hasFullRow = false
                            break
                        }
                    }
                    // If we found a full row we need to remove that row from the map
                    // we do that by just moving all the above rows one row below.
                    if (hasFullRow) {
                        for each (var x = 0 to this.BOARD_TILEMAP_WIDTH - 1) {
                            for each (var y = j to 1, -1) {
                                this.m_map[x][y] = this.m_map[x][y - 1]
                            }
                        }
                        // Increase filled row counter.
                        numFilledRows += 1
                    }
                }

                // Update game statistics.
                if (numFilledRows > 0) {
                    this::onFilledRows(numFilledRows)
                }
                else {
                    Platform::onTetrominoLanded()
                }

                this.m_stats.totalPieces = this.m_stats.totalPieces + 1
                this.m_stats.pieces[this.m_fallingBlock.type]
                        = this.m_stats.pieces[this.m_fallingBlock.type] + 1

                // Use preview tetromino as falling tetromino.
                // Copy preview tetromino for falling tetromino.
                for (i = 0; i < this.TETROMINO_SIZE; i += 1) {
                    for (j = 0; j < this.TETROMINO_SIZE; j += 1) {
                        this.m_fallingBlock.cells[i][j] = this.m_nextBlock.cells[i][j]
                    }
                }
                this.m_fallingBlock.size = this.m_nextBlock.size
                this.m_fallingBlock.type = this.m_nextBlock.type

                // Reset position.
                this.m_fallingBlock.y = 0
                this.m_fallingBlock.x = math.floor((this.BOARD_TILEMAP_WIDTH
                                        - this.m_fallingBlock.size) / 2)
                this::onTetrominoMoved()

                // Create next preview tetromino.
                this::setTetromino(Platform::random() % this.TETROMINO_TYPES, this.m_nextBlock)
            }
        }
    }
    else {
        // There are no collisions, just move the tetromino.
        this.m_fallingBlock.x = this.m_fallingBlock.x + x
        this.m_fallingBlock.y = this.m_fallingBlock.y + y
    }
    this::onTetrominoMoved()
}

// Hard drop.
function Game::dropTetromino() {
    // Shadow has already calculated the landing position.
    this.m_fallingBlock.y = this.m_fallingBlock.y + this.m_shadowGap

    // Force lock.
    this::moveTetromino(0, 1)

    // Update score.
    if (this.m_showShadow) {
        this.m_stats.score = this.m_stats.score + (SCORE_2_FILLED_ROW * (this.m_stats.level +
1)
                                / SCORE_DROP_WITH_SHADOW_DIVISOR)
    }
    else {
```

```
        this.m_stats.score = this.m_stats.score + (SCORE_2_FILLED_ROW * (this.m_stats.level +
1)
                                / SCORE_DROP_DIVISOR)
    }
}

// Main game function called every frame.
function Game::update() {
    // Update game state.
    if (this.m_isOver) {
        if (isFlagSet(this.m_events, this.Event.RESTART)) {
            this.m_isOver = false
            this::start()
        }
    }
    else {
        var currentTime = Platform::getSystemTime()

        // Process delayed autoshift.
        var timeDelta = currentTime - this.m_systemTime

        if (this.m_delayDown > 0) {
            this.m_delayDown = this.m_delayDown - timeDelta
            if (this.m_delayDown <= 0) {
                this.m_delayDown = DAS_MOVE_TIMER
                this.m_events = setFlag(this.m_events, this.Event.MOVE_DOWN)
            }
        }

        if (this.m_delayLeft > 0) {
            this.m_delayLeft = this.m_delayLeft - timeDelta
            if (this.m_delayLeft <= 0) {
                this.m_delayLeft = DAS_MOVE_TIMER
                this.m_events = setFlag(this.m_events, this.Event.MOVE_LEFT)
            }
        }
        else if (this.m_delayRight > 0) {
            this.m_delayRight = this.m_delayRight - timeDelta
            if (this.m_delayRight <= 0) {
                this.m_delayRight = DAS_MOVE_TIMER
                this.m_events = setFlag(this.m_events, this.Event.MOVE_RIGHT)
            }
        }

        if (this.m_delayRotation > 0) {
            this.m_delayRotation = this.m_delayRotation - timeDelta
            if (this.m_delayRotation <= 0) {
                this.m_delayRotation = ROTATION_AUTOREPEAT_TIMER
                this.m_events = setFlag(this.m_events, this.Event.ROTATE_CW)
            }
        }

        // Always handle pause event.
        if (isFlagSet(this.m_events, this.Event.PAUSE)) {
            this.m_isPaused = ! this.m_isPaused
            this.m_events = this.Event.NONE
        }

        // Check if the game is paused.
        if (this.m_isPaused) {
            // We achieve the effect of pausing the game
            // adding the last frame duration to lastFallTime.
            this.m_lastFallTime = this.m_lastFallTime + (currentTime - this.m_systemTime)
        }
        else {
            if (this.m_events != this.Event.NONE) {
                if (isFlagSet(this.m_events, this.Event.SHOW_NEXT)) {
                    this.m_showPreview = ! this.m_showPreview
                    this.m_stateChanged = true
                }
                if (isFlagSet(this.m_events, this.Event.SHOW_SHADOW)) {
                    this.m_showShadow = ! this.m_showShadow
                    this.m_stateChanged = true
                }
                if (isFlagSet(this.m_events, this.Event.DROP)) {
                    this::dropTetromino()
                }
                if (isFlagSet(this.m_events, this.Event.ROTATE_CW)) {
                    this::rotateTetromino(true)
                }
```

```
                if (isFlagSet(this.m_events, this.Event.MOVE_RIGHT)) {
                    this::moveTetromino(1, 0)
                }
                else if (isFlagSet(this.m_events, this.Event.MOVE_LEFT)) {
                    this::moveTetromino(-1, 0)
                }

                if (isFlagSet(this.m_events, this.Event.MOVE_DOWN)) {
                    // Update score if the player accelerates downfall.
                    this.m_stats.score = this.m_stats.score
                                       + (SCORE_2_FILLED_ROW * (this.m_stats.level + 1)
                                                             / SCORE_MOVE_DOWN_DIVISOR)

                    this::moveTetromino(0, 1)
                }
                this.m_events = this.Event.NONE
            }
            // Check if it's time to move downwards the falling tetromino.
            if (currentTime - this.m_lastFallTime >= this.m_fallingDelay) {
                this::moveTetromino(0, 1)
                this.m_lastFallTime = currentTime
            }
        }
        // Save current time for next game update.
        this.m_systemTime = currentTime
    }
}

// This event is called when the falling tetromino is moved.
function Game::onTetrominoMoved() {
    var y = 1
    // Calculate number of cells where shadow tetromino would be.
    while (! this::checkCollision(0, y)) {
        y += 1
    }
    this.m_shadowGap = y - 1
    this.m_stateChanged = true
}

// Process a key down event.
function Game::onEventStart(command) {
    if (command == this.Event.QUIT) {
        this.m_errorCode = this.Error.PLAYER_QUITS
    }
    else if (command == this.Event.MOVE_DOWN) {
        this.m_events = setFlag(this.m_events, this.Event.MOVE_DOWN)
        this.m_delayDown = DAS_DELAY_TIMER
    }
    else if (command == this.Event.ROTATE_CW) {
        this.m_events = setFlag(this.m_events, this.Event.ROTATE_CW)
        this.m_delayRotation = ROTATION_AUTOREPEAT_DELAY
    }
    else if (command == this.Event.MOVE_LEFT) {
        this.m_events = setFlag(this.m_events, this.Event.MOVE_LEFT)
        this.m_delayLeft = DAS_DELAY_TIMER
    }
    else if (command == this.Event.MOVE_RIGHT) {
        this.m_events = setFlag(this.m_events, this.Event.MOVE_RIGHT)
        this.m_delayRight = DAS_DELAY_TIMER
    }
    else if ((command == this.Event.DROP)
            || (command == this.Event.RESTART)
            || (command == this.Event.PAUSE)
            || (command == this.Event.SHOW_NEXT)
            || (command == this.Event.SHOW_SHADOW)) {
        this.m_events = setFlag(this.m_events, command)
    }
}

// Process a key up event.
function Game::onEventEnd(command) {
    if (command == this.Event.MOVE_DOWN) {
        this.m_delayDown = -1
    }
    else if (command == this.Event.MOVE_LEFT) {
        this.m_delayLeft = -1
    }
    else if (command == this.Event.MOVE_RIGHT) {
        this.m_delayRight = -1
    }
    else if (command == this.Event.ROTATE_CW) {
```

```
        this.m_delayRotation = -1
    }
}

// Return game instance
return Game


/* ========================================================================== */
/*   platform.kia                                                             */
/*   Platform implementation for Love 2D.                                     */
/* -------------------------------------------------------------------------- */

// Local reference to engine.
var love = global.love

// Local reference to game.
var Game

// Screen size
var SCREEN_WIDTH  = 480
var SCREEN_HEIGHT = 272

// Size of square tile
var TILE_SIZE = 12

// Board up-left corner coordinates
var BOARD_X = 180
var BOARD_Y = 4

// Preview tetromino position
var PREVIEW_X = 112
var PREVIEW_Y = 210

// Score position and length on screen
var SCORE_X      = 72
var SCORE_Y      = 52
var SCORE_LENGTH = 10

// Lines position and length on screen
var LINES_X      = 108
var LINES_Y      = 34
var LINES_LENGTH = 5

// Level position and length on screen
var LEVEL_X      = 108
var LEVEL_Y      = 16
var LEVEL_LENGTH = 5

// Tetromino subtotals position
var TETROMINO_X   = 425
var TETROMINO_L_Y = 53
var TETROMINO_I_Y = 77
var TETROMINO_T_Y = 101
var TETROMINO_S_Y = 125
var TETROMINO_Z_Y = 149
var TETROMINO_O_Y = 173
var TETROMINO_J_Y = 197

// Size of subtotals
var TETROMINO_LENGTH = 5

// Tetromino total position
var PIECES_X      = 418
var PIECES_Y      = 221
var PIECES_LENGTH = 6

// Size of number
var NUMBER_WIDTH  = 7
var NUMBER_HEIGHT = 9

var NULL = 0
var MUSIC_VOLUME = 0.25

var Platform = {
    // Images.
    m_bmpBackground: NULL;
    m_bmpBlocks:     NULL;
    m_bmpNumbers:    NULL;

    // Sprite arrays.
```

```
    m_blocks:  NULL;
    m_numbers: NULL;

    // Sounds.
    m_musicLoop:  NULL;
    m_musicIntro: NULL;
    m_soundLine:  NULL;
    m_soundDrop:  NULL;
    m_musicMute:  false;
}

// Initializes platform.
function Platform::init(game) {
    Game = game

    // Set window title
    love.graphics.setCaption("STC - " .. global._VERSION)

    // Set window size
    assert(love.graphics.setMode(SCREEN_WIDTH, SCREEN_HEIGHT, false, true, 0),
           "Could not set screen mode")

    // Initialize random generator
    math.randomseed(os.time())

    // Load images.
    this.m_bmpBackground = love.graphics.newImage("back.png")

    this.m_bmpBlocks = love.graphics.newImage("blocks.png")
    this.m_bmpBlocks::setFilter("nearest", "nearest")
    var w = this.m_bmpBlocks::getWidth()
    var h = this.m_bmpBlocks::getHeight()

    // Create quads for blocks
    this.m_blocks = []
    for each (var shadow = 0 to 1) {
        this.m_blocks[shadow] = []
        for each (var color = 0 to Game.COLORS - 1) {
            this.m_blocks[shadow][color]
                    = love.graphics.newQuad(TILE_SIZE * color, (TILE_SIZE + 1) * shadow,
                                            TILE_SIZE + 1, TILE_SIZE + 1, w, h)
        }
    }

    this.m_bmpNumbers = love.graphics.newImage("numbers.png")
    this.m_bmpNumbers::setFilter("nearest", "nearest")
    w = this.m_bmpNumbers::getWidth()
    h = this.m_bmpNumbers::getHeight()

    // Create quads for numbers
    this.m_numbers = []
    for each (var color = 0 to Game.COLORS - 1) {
        this.m_numbers[color] = []
        for each (var digit = 0 to 9) {
            this.m_numbers[color][digit]
                    = love.graphics.newQuad(NUMBER_WIDTH * digit, NUMBER_HEIGHT * color,
                                            NUMBER_WIDTH, NUMBER_HEIGHT, w, h)
        }
    }

     // Load music.
    this.m_musicIntro = love.audio.newSource("stc_theme_intro.ogg", "stream")
    this.m_musicIntro::setVolume(MUSIC_VOLUME)
    this.m_musicIntro::play()
    this.m_musicLoop = love.audio.newSource("stc_theme_loop.ogg", "stream")
    this.m_musicLoop::setLooping(true)
    this.m_musicLoop::setVolume(MUSIC_VOLUME)
    this.m_soundLine = love.audio.newSource("fx_line.wav", "static")
    this.m_soundDrop = love.audio.newSource("fx_drop.wav", "static")
    this.m_musicMute = false
}

// Called when a tetromino landed
function Platform::onTetrominoLanded() {
    if (! this.m_musicMute) {
        this.m_soundDrop::rewind()
        this.m_soundDrop::play()
    }
}

// Called when a line was made
```

```
function Platform::onFilledRows(lines) {
    if (! this.m_musicMute) {
        this.m_soundLine::rewind()
        this.m_soundLine::play()
    }
}

// Process events and notify game.
function Platform::onKeyDown(key) {
    if (key == "escape") {
        love.event.push("quit")
    }
    if ((key == "left") || (key == "a")) {
        Game::onEventStart(Game.Event.MOVE_LEFT)
    }
    if ((key == "right") || (key == "d")) {
        Game::onEventStart(Game.Event.MOVE_RIGHT)
    }
    if ((key == "down") || (key == "s")) {
        Game::onEventStart(Game.Event.MOVE_DOWN)
    }
    if ((key == "up") || (key == "w")) {
        Game::onEventStart(Game.Event.ROTATE_CW)
    }
    if (key == " ") {
        Game::onEventStart(Game.Event.DROP)
    }
    if (key == "f5") {
        Game::onEventStart(Game.Event.RESTART)
    }
    if (key == "f1") {
        Game::onEventStart(Game.Event.PAUSE)
    }
    if (key == "f2") {
        Game::onEventStart(Game.Event.SHOW_NEXT)
    }
    if (key == "f3") {
        Game::onEventStart(Game.Event.SHOW_SHADOW)
    }
}

function Platform::onKeyUp(key) {
    if ((key == "left") || (key == "a")) {
        Game::onEventEnd(Game.Event.MOVE_LEFT)
    }
    if ((key == "right") || (key == "d")) {
        Game::onEventEnd(Game.Event.MOVE_RIGHT)
    }
    if ((key == "down") || (key == "s")) {
        Game::onEventEnd(Game.Event.MOVE_DOWN)
    }
    if ((key == "up") || (key == "w")) {
        Game::onEventEnd(Game.Event.ROTATE_CW)
    }
    if (key == "f4") {
        if (this.m_musicMute) {
            if (this.m_musicIntro) {
                this.m_musicIntro::resume()
            } else {
                this.m_musicLoop::resume()
            }
        }
        else {
            if (this.m_musicIntro) {
                this.m_musicIntro::pause()
            } else {
                this.m_musicLoop::pause()
            }
        }
        this.m_musicMute = ! this.m_musicMute
    }
}

// Draw a tile from a tetromino
function Platform::drawTile(x, y, tile, shadow) {
    love.graphics.drawq(this.m_bmpBlocks, this.m_blocks[shadow][tile], x, y)
}

// Draw a number on the given position
function Platform::drawNumber(x, y, number, length, color) {
    var pos = 0
```

```
    do {
        love.graphics.drawq(this.m_bmpNumbers, this.m_numbers[color][number % 10],
                            x + NUMBER_WIDTH * (length - pos), y)
        number = math.floor(number / 10)
        pos += 1
    } while (pos < length)
}

// Render the state of the game using platform functions.
function Platform::renderGame() {
    var i, j

    // Draw background
    love.graphics.draw(this.m_bmpBackground, 0, 0)

    // Draw preview block
    if (Game::showPreview()) {
        for (i = 0; i < Game.TETROMINO_SIZE; i += 1) {
            for (j = 0; j < Game.TETROMINO_SIZE; j += 1) {
                if (Game::nextBlock().cells[i][j] != Game.Cell.EMPTY) {
                    this::drawTile(PREVIEW_X + TILE_SIZE * i,
                                   PREVIEW_Y + TILE_SIZE * j,
                                   Game::nextBlock().cells[i][j], 0)
                }
            }
        }
    }

    // Draw shadow tetromino
    if (Game::showShadow() && Game::shadowGap() > 0) {
        for (i = 0; i < Game.TETROMINO_SIZE; i += 1) {
            for (j = 0; j < Game.TETROMINO_SIZE; j += 1) {
                if (Game::fallingBlock().cells[i][j] != Game.Cell.EMPTY) {
                    this::drawTile(BOARD_X + (TILE_SIZE * (Game::fallingBlock().x + i)),
                                   BOARD_Y + (TILE_SIZE * (Game::fallingBlock().y
                                   + Game::shadowGap() + j)),
                                   Game::fallingBlock().cells[i][j], 1)
                }
            }
        }
    }

    // Draw the cells in the board
    for (i = 0; i < Game.BOARD_TILEMAP_WIDTH; i += 1) {
        for (j = 0; j < Game.BOARD_TILEMAP_HEIGHT; j += 1) {
            if (Game::getCell(i, j) != Game.Cell.EMPTY) {
                this::drawTile(BOARD_X + (TILE_SIZE * i),
                               BOARD_Y + (TILE_SIZE * j),
                               Game::getCell(i, j), 0)
            }
        }
    }

    // Draw falling tetromino
    for (i = 0; i < Game.TETROMINO_SIZE; i += 1) {
        for (j = 0; j < Game.TETROMINO_SIZE; j += 1) {
            if (Game::fallingBlock().cells[i][j] != Game.Cell.EMPTY) {
                this::drawTile(BOARD_X + TILE_SIZE * (Game::fallingBlock().x + i),
                               BOARD_Y + TILE_SIZE * (Game::fallingBlock().y + j),
                               Game::fallingBlock().cells[i][j], 0)
            }
        }
    }

    // Draw game statistic data
    if (! Game::isPaused()) {
        this::drawNumber(LEVEL_X, LEVEL_Y, Game::stats().level, LEVEL_LENGTH,
Game.Cell.WHITE)
        this::drawNumber(LINES_X, LINES_Y, Game::stats().lines, LINES_LENGTH,
Game.Cell.WHITE)
        this::drawNumber(SCORE_X, SCORE_Y, Game::stats().score, SCORE_LENGTH,
Game.Cell.WHITE)

        this::drawNumber(TETROMINO_X, TETROMINO_L_Y,
Game::stats().pieces[Game.TetrominoType.L],
                         TETROMINO_LENGTH, Game.Cell.ORANGE)
        this::drawNumber(TETROMINO_X, TETROMINO_I_Y,
Game::stats().pieces[Game.TetrominoType.I],
                         TETROMINO_LENGTH, Game.Cell.CYAN)
        this::drawNumber(TETROMINO_X, TETROMINO_T_Y,
Game::stats().pieces[Game.TetrominoType.T],
```

```
                        TETROMINO_LENGTH, Game.Cell.PURPLE)
        this::drawNumber(TETROMINO_X, TETROMINO_S_Y,
Game::stats().pieces[Game.TetrominoType.S],
                        TETROMINO_LENGTH, Game.Cell.GREEN)
        this::drawNumber(TETROMINO_X, TETROMINO_Z_Y,
Game::stats().pieces[Game.TetrominoType.Z],
                        TETROMINO_LENGTH, Game.Cell.RED)
        this::drawNumber(TETROMINO_X, TETROMINO_O_Y,
Game::stats().pieces[Game.TetrominoType.O],
                        TETROMINO_LENGTH, Game.Cell.YELLOW)
        this::drawNumber(TETROMINO_X, TETROMINO_J_Y,
Game::stats().pieces[Game.TetrominoType.J],
                        TETROMINO_LENGTH, Game.Cell.BLUE)

        this::drawNumber(PIECES_X, PIECES_Y, Game::stats().totalPieces, PIECES_LENGTH,
                        Game.Cell.WHITE)
    }

    // Adding music loop check here for convenience.
    if (this.m_musicIntro) {
        if (this.m_musicIntro::isStopped()) {
            this.m_musicIntro = null
            this.m_musicLoop::play()
        }
    }
}

function Platform::getSystemTime() {
    return math.floor(1000 * love.timer.getTime())
}

function Platform::random() {
    return math.random(1_000_000_000)
}

// Return platform instance
return Platform
```

```
/* ========================================================================== */
/*   main.kia                                                                 */
/*   Main game file.                                                          */
/* -------------------------------------------------------------------------- */

var game     = require("game")
var platform = require("platform")

function love.load() {
    game::init(platform)
}

function love.update(dt) {
    game::update()
}

function love.draw() {
    platform::renderGame()
}

function love.keypressed(key, unicode) {
    platform::onKeyDown(key)
}

function love.keyreleased(key) {
    platform::onKeyUp(key)
}
```

Apéndice C: Visor del fractal de Mandelbrot en Killa

```
/* ========================================================================== */
/*   Mandelbrot.kia                                                           */
/* -------------------------------------------------------------------------- */

var INIT_STEPS = 32
var INCREMENT_STEPS = 64

var MINIMUM_COLOR_STEP = 5
var COLOR_RANGE = 64
var COLOR_STEP = 6

var COLOR_BLUE   = 1
var COLOR_RED    = 2
var COLOR_GREEN  = 3
var COLOR_GRAY   = 4
var COLOR_RANDOM = 5
var COLOR_PALETTE = 6

var MINIMUM_CELL_SIZE = 9e-15

var ZOOM_FACTOR = 0.5

//----------------------------------------------------------------------------
var m_width  // canvas width
var m_height // canvas height

var m_canvas // node where the fractal is drawn

var m_colors // palette of colors
var m_colorSteps

// Number of iterations for computing a pixel of the fractal.
var m_steps

// Fractal zone coordinates
var m_fractalZoneX1
var m_fractalZoneY1
var m_fractalZoneX2
var m_fractalZoneY2

var m_zoomIsVisible
var m_isDrawingFractal

// Fractal zones history.
// Each element is an array with the zone coordinates: [x1, y1, x2, y2]
var m_zones

// Some nice palette colors
var m_palette

var m_x
var m_dx
var m_dy

//----------------------------------------------------------------------------
// Logging messages
function log(...) {
    CCLuaLog(string.format(...))
}

function drawZoomZone(x, y) {
    var x1 = x - ZOOM_FACTOR * m_width / 2
    var y1 = y - ZOOM_FACTOR * m_height / 2

    m_canvas::begin()
    m_canvas::setColor(1.0, 1.0, 1.0)
    m_canvas::drawRectangle(x1, y1, ZOOM_FACTOR*m_width, ZOOM_FACTOR*m_height)
    m_canvas::finish()
}

function drawSegmentMandelbrot() {
    var px, py
    var steps
    var fx, fy
    var temp

    for (var y = 0; y < m_height; y += 1) {
```

```
        // Point in fractal zone
        px = m_fractalZoneX1 + m_x * m_dx
        py = m_fractalZoneY1 + y * m_dy

        // Iterate fractal computation.
        steps = 0
        fx = 0.0
        fy = 0.0

        while (true) {
            // Mandelbrot recurrence:
            // ----------------------
            // F(n+1) = F(n)*F(n) + (px + i*py)
            temp = fx*fx - fy*fy + px
            fy = 2*fx*fy + py
            fx = temp

            steps += 1

            // F(z) belongs to Mandelbrot set if |F(z)| < 2
            // We give up if we passed the limit of number of iterations.
            if ((steps >= m_steps) || (fx*fx + fy*fy >= 4.0)) {
                break;
            }
        }

        if (steps < m_steps) {
            // We found that: |F(z)| >= 2 (the point doesn't belong to
            // the Mandelbrot set)
            var indexColor = (steps - 1) % 28 + 1
            if (indexColor > 15) {
                indexColor = 30 - indexColor
            }
            // Sustractiong 0.5 to coordinates for correct rendering
            // in some points (it was plotting the x=239 pixel as x=240).
            m_canvas::drawPoint(m_x - 0.5, y - 0.5,
                                m_colors[indexColor][0]/255,
                                m_colors[indexColor][1]/255,
                                m_colors[indexColor][2]/255)
        }
        else {
            // We suspect this point belongs to the Mandelbrot set.
            // We can't be sure because the point can belong to the set if
            // the number of iteration steps is increased.
            // Leave this pixel in black.
            m_canvas::drawPoint(m_x - 0.5, y - 0.5, 0, 0, 0)
        }
    }
}

function drawMandelbrot(initializeZone) {

    if (initializeZone) {
        m_fractalZoneX1 = -2.3
        m_fractalZoneY1 = -1.2
        m_fractalZoneX2 = 1.3
        m_fractalZoneY2 = 1.2
    }

    m_dx = (m_fractalZoneX2 - m_fractalZoneX1) / (m_width - 1)
    m_dy = (m_fractalZoneY2 - m_fractalZoneY1) / (m_height - 1)

    m_x = 0
    m_isDrawingFractal = true
    m_canvas::begin()
    drawSegmentMandelbrot()
    m_canvas::finish()
}

function redraw(fx, fy) {

    // If the zoom grid is visible we want to redraw the zoom zone.
    var dx = ZOOM_FACTOR * (m_fractalZoneX2 - m_fractalZoneX1) / 2
    var dy = ZOOM_FACTOR * (m_fractalZoneY2 - m_fractalZoneY1) / 2

    if ((dx >= MINIMUM_CELL_SIZE) && (dy >= MINIMUM_CELL_SIZE)) {

        // Save actual zone position
        table.insert(m_zones, [m_fractalZoneX1, m_fractalZoneY1,
                               m_fractalZoneX2, m_fractalZoneY2])
```

```
        // Set new drawing zone:
        // (x1,y1) bottom-left corner, (x2,y2) up-right corner
        var cx = fx * (m_fractalZoneX2 - m_fractalZoneX1)
        var cy = fy * (m_fractalZoneY2 - m_fractalZoneY1)

        m_fractalZoneX2 = m_fractalZoneX1 + cx + dx
        m_fractalZoneY2 = m_fractalZoneY1 + cy + dy
        m_fractalZoneX1 += cx - dx
        m_fractalZoneY1 += cy - dy

        drawMandelbrot()
    }
}

function increaseSteps(increment) {
    if (m_steps + increment > 0) {
        m_steps += increment
        drawMandelbrot()
    }
}

function zoomOut() {
    if ($m_zones > 0) {
        var zone = table.remove(m_zones)
        m_fractalZoneX1 = zone[0]
        m_fractalZoneY1 = zone[1]
        m_fractalZoneX2 = zone[2]
        m_fractalZoneY2 = zone[3]
    }
    else {
        var dx = (m_fractalZoneX2 - m_fractalZoneX1)/2
        var dy = (m_fractalZoneY2 - m_fractalZoneY1)/2
        m_fractalZoneX1 -= dx
        m_fractalZoneY1 -= dy
        m_fractalZoneX2 += dx
        m_fractalZoneY2 += dy
    }
    drawMandelbrot()
}

function setColors(opc) {
    var k
    m_colors = []

    if (opc == COLOR_GRAY) {
        for (k = 0; k < 16; k += 1) {
            table.insert(m_colors, [4*m_colorSteps[k][1],
                        4*m_colorSteps[k][1], 4*m_colorSteps[k][1]])
        }
    }
    else if (opc == COLOR_BLUE) {
        for (k = 0; k < 16; k += 1) {
            table.insert(m_colors, [4*m_colorSteps[k][0],
                        4*m_colorSteps[k][1], 4*m_colorSteps[k][2]])
        }
    }
    else if (opc == COLOR_RED) {
        for (k = 0; k < 16; k += 1) {
            table.insert(m_colors, [4*m_colorSteps[k][2],
                        13 * k, 3*m_colorSteps[k][0]])
        }
    }
    else if (opc == COLOR_GREEN) {
        for (k = 0; k < 16; k += 1) {
            table.insert(m_colors, [14 * k, 4*m_colorSteps[k][2],
                        4*m_colorSteps[k][0]])
        }
    }
    else if (opc == COLOR_RANDOM) {
        var c1, c2, c3
        do {
            c1 = math.random(COLOR_STEP) - 1
            c2 = math.random(COLOR_STEP) - 1
            c3 = math.random(COLOR_STEP) - 1

        } while ((c1 + c2 + c3) < MINIMUM_COLOR_STEP)

        table.insert(m_colors, [0, 0, 0]) // base color is black

        var a1 = math.random(COLOR_RANGE) - 1
        var a2 = math.random(COLOR_RANGE) - 1
```

```
        var a3 = math.random(COLOR_RANGE) - 1

        // Fill color table.
        for (k = 1; k < 16; k += 1) {
            var t1 = (a1 + (k-1) * c1) % (COLOR_RANGE * 2)
            var t2 = (a2 + (k-1) * c2) % (COLOR_RANGE * 2)
            var t3 = (a3 + (k-1) * c3) % (COLOR_RANGE * 2)

            if (t1 >= COLOR_RANGE) {
                t1 = (COLOR_RANGE * 2) - t1 - 1
            }
            if (t2 >= COLOR_RANGE) {
                t2 = (COLOR_RANGE * 2) - t2 - 1
            }
            if (t3 >= COLOR_RANGE) {
                t3 = (COLOR_RANGE * 2) - t3 - 1
            }
            table.insert(m_colors, [4*t1, 4*t2, 4*t3])
        }
    }
    else if (opc == COLOR_PALETTE) {
        var palette = math.random($m_palette) - 1
        table.insert(m_colors, [0, 0, 0]) // base color is black

        // Fill color table.
        for (k = 1; k < 16; k += 1) {
            var p1 = (m_palette[palette][0]
                      + (k-1) * m_palette[palette][1]) % (COLOR_RANGE * 2)
            var p2 = (m_palette[palette][2]
                      + (k-1) * m_palette[palette][3]) % (COLOR_RANGE * 2)
            var p3 = (m_palette[palette][4]
                      + (k-1) * m_palette[palette][5]) % (COLOR_RANGE * 2)

            if (p1 >= COLOR_RANGE) {
                p1 = (COLOR_RANGE * 2) - p1 - 1
            }
            if (p2 >= COLOR_RANGE) {
                p2 = (COLOR_RANGE * 2) - p2 - 1
            }
            if (p3 >= COLOR_RANGE) {
                p3 = (COLOR_RANGE * 2) - p3 - 1
            }
            table.insert(m_colors, [4*p1, 4*p2, 4*p3])
        }
    }
}

function initializeColorData() {
    m_palette = [
        [0, 4, 62, 5, 31, 3],  [7, 4, 4, 4, 42, 5],   [8, 0, 55, 4, 4, 4],
        [8, 5, 8, 4, 8, 1],    [12, 4, 44, 2, 46, 3], [17, 4, 35, 5, 41, 4],
        [20, 5, 43, 4, 57, 3], [20, 5, 58, 5, 21, 2], [21, 2, 35, 4, 59, 0],
        [24, 4, 53, 2, 54, 3], [25, 2, 36, 2, 50, 2], [25, 5, 52, 5, 0, 5],
        [27, 3, 19, 3, 31, 5], [27, 3, 35, 4, 39, 2], [29, 5, 63, 2, 34, 2],
        [32, 5, 58, 5, 33, 2], [33, 5, 61, 5, 34, 3], [35, 2, 16, 5, 22, 0],
        [35, 4, 2, 0, 10, 3],  [36, 4, 43, 4, 35, 2], [38, 4, 63, 3, 55, 5],
        [39, 5, 8, 2, 48, 2],  [39, 5, 59, 3, 7, 2],  [40, 3, 6, 0, 61, 5],
        [40, 5, 58, 5, 25, 2], [41, 2, 49, 5, 52, 3], [41, 5, 59, 5, 0, 3],
        [43, 5, 56, 3, 43, 4], [44, 2, 11, 3, 54, 4], [44, 4, 61, 4, 13, 2],
        [45, 3, 61, 3, 10, 1], [45, 4, 63, 5, 6, 3],  [45, 5, 46, 5, 11, 0],
        [46, 3, 5, 5, 17, 3],  [47, 1, 30, 4, 14, 0], [48, 5, 58, 5, 17, 2],
        [48, 5, 63, 4, 6, 4],  [48, 5, 63, 5, 6, 4],  [49, 3, 17, 4, 38, 2],
        [50, 2, 63, 5, 57, 3], [51, 5, 62, 3, 37, 0], [53, 1, 56, 5, 13, 1],
        [53, 3, 57, 2, 49, 1], [53, 3, 56, 5, 44, 2], [54, 4, 1, 0, 33, 3],
        [54, 5, 53, 4, 45, 2], [55, 4, 13, 0, 4, 4],  [55, 4, 40, 5, 34, 2],
        [55, 5, 57, 2, 56, 2], [57, 2, 14, 3, 20, 0], [58, 1, 15, 5, 9, 2],
        [58, 3, 38, 4, 13, 4], [59, 0, 48, 5, 6, 2],  [59, 3, 3, 0, 6, 3],
        [59, 5, 4, 4, 60, 0],  [59, 5, 52, 3, 5, 0],  [60, 1, 51, 5, 0, 3],
        [60, 5, 14, 2, 24, 3], [61, 5, 42, 5, 24, 3], [63, 4, 14, 3, 0, 5]
    ]

    m_colorSteps = [
        [ 0,  0,  0], [10,  8, 23], [13, 16, 36], [15, 18, 41],
        [17, 21, 46], [18, 23, 49], [20, 26, 50], [22, 29, 52],
        [24, 35, 55], [22, 40, 58], [20, 45, 60], [17, 46, 61],
        [16, 47, 62], [25, 52, 62], [38, 58, 63], [63, 63, 63]
    ]
}

//--------------------------------------------------------------------------------
function onTouchBegan(tx, ty) {
```

```
    if (!m_zoomIsVisible) {
        m_zoomIsVisible = true
        drawZoomZone(tx, ty)

        // The closures below control the zoom enable/disable process
        var idSelector
        var scheduler
        function refresh() {
            CCScheduler::sharedScheduler()::unscheduleScriptEntry(idSelector)
            redraw(tx / m_width, ty / m_height)

            function enableZoom() {
                CCScheduler::sharedScheduler()::unscheduleScriptEntry(idSelector)
                m_zoomIsVisible = false
            }
            // Give a little rest before enabling zoom
            idSelector = scheduler::scheduleScriptFunc(enableZoom, 0.1, false)
        }
        scheduler = CCScheduler::sharedScheduler()
        // Show the zoom area
        idSelector = scheduler::scheduleScriptFunc(refresh, 0.05, false)
    }
    return true // CCTOUCHBEGAN event must return true
}

function init() {
    // Initialize random generator
    math.randomseed(os.time())

    // Avoid memory leaks
    collectgarbage("setpause", 100)
    collectgarbage("setstepmul", 5000)

    // Disable FPS label.
    CCDirector::sharedDirector()::setDisplayFPS(false)

    // Get size.
    var size = CCDirector::sharedDirector()::getWinSize()
    m_width = size.width
    m_height = size.height

    //---------------------------------------------------------------------------
    // Create scene
    var scene = CCScene::node()
    CCDirector::sharedDirector()::runWithScene(scene)

    m_canvas = CCRenderTexture::renderTextureWithWidthAndHeight(m_width, m_height)
    m_canvas::setPosition(m_width/2, m_height/2)
    m_canvas::setPointSize(1);
    scene::addChild(m_canvas, 1)

    m_steps = INIT_STEPS
    m_zones = []
    m_zoomIsVisible = false
    m_isDrawingFractal = false

    //---------------------------------------------------------------------------
    // Create menu of buttons on screen
    //
    var halfSize = 20
    var btnPalette = CCMenuItemImage::itemFromNormalImage(
                        "mandelbrot/random.png", "mandelbrot/button_on.png")
    btnPalette::setPosition(-m_width / 2 + halfSize, -m_height / 2 + halfSize);
    function onBtnPaletteClicked() {
        setColors(COLOR_PALETTE)
        drawMandelbrot()
    }
    btnPalette::registerScriptHandler(onBtnPaletteClicked)

    var btnRandom = CCMenuItemImage::itemFromNormalImage(
                        "mandelbrot/random.png", "mandelbrot/button_on.png")
    btnRandom::setPosition(-m_width / 2 + halfSize, m_height / 2 - halfSize);
    function onBtnRandomClicked() {
        setColors(COLOR_RANDOM)
        drawMandelbrot()
    }
    btnRandom::registerScriptHandler(onBtnRandomClicked)

    var btnBack = CCMenuItemImage::itemFromNormalImage(
                        "mandelbrot/back.png", "mandelbrot/button_on.png")
    btnBack::setPosition(m_width / 2 - halfSize, m_height / 2 - halfSize);
```

```
    function onBtnBackClicked() {
        zoomOut()
    }
    btnBack::registerScriptHandler(onBtnBackClicked)

    var btnHome = CCMenuItemImage::itemFromNormalImage(
                        "mandelbrot/home.png", "mandelbrot/button_on.png")
    btnHome::setPosition(m_width / 2 - halfSize, -m_height / 2 + halfSize);
    function onBtnHomeClicked() {
        m_steps = INIT_STEPS
        drawMandelbrot(true)
    }
    btnHome::registerScriptHandler(onBtnHomeClicked)

    var btnStepsUp = CCMenuItemImage::itemFromNormalImage(
                        "mandelbrot/up.png", "mandelbrot/button_on.png")
    btnStepsUp::setPosition(0, m_height / 2 - halfSize);
    function onBtnStepsUpClicked() {
        increaseSteps(INCREMENT_STEPS)
    }
    btnStepsUp::registerScriptHandler(onBtnStepsUpClicked)

    var btnStepsDown = CCMenuItemImage::itemFromNormalImage(
                        "mandelbrot/down.png", "mandelbrot/button_on.png")
    btnStepsDown::setPosition(0, -m_height / 2 + halfSize);
    function onBtnStepsDownClicked() {
        increaseSteps(-INCREMENT_STEPS)
    }
    btnStepsDown::registerScriptHandler(onBtnStepsDownClicked)

    var menu = CCMenu::menuWithItem(btnPalette);
       menu::setPosition(m_width / 2, m_height / 2);
    menu::addChild(btnRandom, 2);
    menu::addChild(btnBack, 3);
    menu::addChild(btnHome, 4);
    menu::addChild(btnStepsUp, 5);
    menu::addChild(btnStepsDown, 6);

       scene::addChild(menu, 2);

    //-------------------------------------------------------------------------
    // Touch events handler
    var layer = CCLayer::node();
    scene::addChild(layer);

    function onTouch(eventType, x, y) {
        if (eventType == global.CCTOUCHBEGAN) {
            return onTouchBegan(x, y)
        }
    }

    layer::registerScriptTouchHandler(onTouch)
    layer::setIsTouchEnabled(true)

    // Schedule refresh update.
    function update(dt) {
        if (m_isDrawingFractal) {
            if (m_x < m_width) {
                m_canvas::begin();
                drawSegmentMandelbrot()
                m_canvas::finish();
                m_x += 1
                if (m_x >= m_width) {
                    m_isDrawingFractal = false
                }
            }
        }
    }
    CCScheduler::sharedScheduler()::scheduleScriptFunc(update, 0, false)

    //-------------------------------------------------------------------------
    // Initialize color data
    initializeColorData()
    setColors(COLOR_PALETTE)

    // Draw Mandelbrot fractal in default zone
    drawMandelbrot(true)
}

init();
```