Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación
Centro de Computación Gráfica

Extensión de un Lenguaje Interpretado para el Despliegue 3D en la Web

Trabajo Especial de Grado presentado ante la ilustre Universidad Central de Venezuela por el Br. Amaro David Duarte Raybauid Massilia para optar al título de Licenciado en Computación, Mención Computación Gráfica.
Tutor: Esmitt Ramírez.

Mayo 2016

Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación
Centro de Computación Gráfica

ACTA DEL VEREDICTO

Quienes suscriben, miembros del Jurado designado por el Consejo de Escuela de Computación, para examinar el Trabajo Especial de Grado presentado por el bachiller Duarte Raybaudi-Massilia, Amaro David, portador de la Cédula de Identidad V-20.616.922, con el título: **"Extensión de un Lenguaje Interpretado para el Despliegue 3D en la Web"**, a los fines de optar al título de Licenciado en Computación, dejan constancia de lo siguiente:

Leído como fue, dicho trabajo por cada uno de los miembros del jurado se fijó el día 20 de mayo de 2016, a las 9:00 am, para que su autor lo defienda en forma pública, lo que hizo en **Centro de Computación Gráfica** de la Escuela de Computación, mediante una presentación oral de su contenido, luego de lo cual respondió las preguntas formuladas. Finalizada la defensa pública del Trabajo Especial de Grado, el jurado decidió aprobarlo con la nota de 20 puntos.

En fe de lo cual se levanta la presente Acta, en Caracas el día 20 de mayo de 2016.

Prof. Esmitt Ramírez

Tutor

Prof. Héctor Navarro

Jurado

Prof. Jaime Parada

Jurado

Resumen

Actualmente existen herramientas que facilitan el desarrollo del despliegue de gráficos debido a que este proceso es complejo y extenso. Sin embargo, estas herramientas no son accesibles para todo usuario, en su lugar muchas requieren alguna licencia de uso que debe pagarse. Por otro lado también es común que no proporcionen o que escasamente proporcionen formas de exportar el desarrollo realizado a otra herramienta o lenguaje conocido, de manera que el usuario pueda desarrollar a su gusto el despliegue de gráficos o incluso mejorar las técnicas proporcionadas por la herramienta.

En este trabajo se presenta una herramienta que facilita el desarrollo del despliegue de gráficos, mostrando el despliegue de forma inmediata y que provee al usuario la posibilidad de generar una *Plantilla* de código en un lenguaje conocido, de manera que obtenga un despliegue básico y estándar que pueda extender o mejorar a su gusto.

Palabras Clave: *Despliegue Gráfico, Fácil Desarrollo, Plantilla de Código, Lenguaje, Resultados Inmediatos.*

Abstract

Nowadays there are tools to ease the development of graphics display due its complex and extensive process. However, these tools are not accessible to all users, instead many of them require any paid license. Moreover, it is also common do not provides or merely provides ways to export the achieved development to another tool or well-known language, where user can develop a personalized graphics render or even improve techniques provided by the tool.

In this paper, we presented a tool that reached an easy development on graphics display, showing the render immediately and provides the user the ability to generate a code template in a well-known language, to get a basic and standard render which can be extended or improved.

Keywords: *Graphical Display, Easy Develop, Code Template, Language, Instant Results.*

Índice General

Índice de Figuras

Introducción

Para un desarrollador de aplicaciones gráficas, el despliegue de gráficos suele ser complicado y extenso en líneas de código, lo cual inspira a crear herramientas que lo faciliten y reduzcan la cantidad de líneas de código. Existen diferentes herramientas que cumplen ambos o uno de estos puntos, ya sea abstrayendo al desarrollador del código o proporcionándole uno sencillo.

Los lenguajes proporcionados por dichas herramientas pueden ser extensiones de lenguajes ya conocidos o lenguajes completamente nuevos y dedicados al despliegue de gráficos. Si bien ambos reducen la cantidad de líneas de código, también tienen inconvenientes. Las extensiones sólo pueden ser empleadas a través del lenguaje que extienden, acarreando los inconvenientes del lenguaje. Por ejemplo, el lenguaje puede ser no conocido por el programador, ser de lógica complicada y no permitir visualizar los resultados de la ejecución de manera inmediata. Los lenguajes completamente nuevos obligan al desarrollador a aprender este nuevo lenguaje, el cual puede no proveer las funcionalidades que el desarrollador desea utilizar en su despliegue y no suelen proveer una forma de exportar el despliegue a otro lenguaje o herramienta conocida en donde el desarrollador si pueda implementar o utilizar la funcionalidad que desea.

Por ello, en este trabajo propone el uso de una herramienta que facilite el despliegue de gráficos, proporcionando al desarrollador un lenguaje o una extensión de uno conocido que no requiera de muchas líneas de código, que sea intuitivo para el desarrollador, que muestre el despliegue de manera inmediata y que el código pueda exportarse a otra herramienta o lenguaje conocido en caso de que el desarrollador lo necesite.

Este trabajo está estructurado por capítulos de la siguiente manera: En el capítulo 1 se expone el problema, los objetivos generales y específicos contemplados, la justificación del proyecto, la solución propuesta y el alcance definido. El capítulo 2 define los conceptos de extensibilidad en lenguajes, procesamiento de lenguajes, analizadores sintácticos, API’s de despliegue gráficos, fuentes de luz y modelos de sombreado. En el capítulo 3 se explica el ciclo de vida de la metodología utilizada, así como algunos detalles de importancia. En el capítulo 4 se explica detalladamente la construcción de la solución siguiendo la estructura de la metodología utilizada. En el capítulo 5 se exponen las conclusiones y el análisis de los resultados extraídos de las pruebas realizadas.

Capítulo 1

Problema de Investigación

1.1 Planteamiento del Problema

Existen varios lenguajes y extensiones de lenguajes que soportan el despliegue de gráficos. Las extensiones de lenguajes por su condición de extender de otro lenguaje proporcionan un ambiente de desarrollo conocido a los programadores, por la misma razón están ampliamente disponibles y son fáciles de mantener. Sin embargo, estas extensiones suelen ser complejas de utilizar y comprender, además de extender los códigos en demasiadas líneas. Por otra parte los lenguajes completos, es decir que no son extensiones de otro, suelen ser más expresivos en el dominio gráfico y por lo tanto fácil para su desarrollo. Sin embargo suelen poseer licencias comerciales, siendo no accesibles para cualquier usuario. Otra característica importante que no se provee con frecuencia es la posibilidad de exportar el despliegue realizado a otro lenguaje, de manera que el usuario pueda escoger qué lenguaje es más adecuado para su despliegue o qué lenguaje le provee las herramientas que necesita para su despliegue.

1.2 Objetivo General

Crear una extensión de un lenguaje de programación con su procesador, que facilite el despliegue de gráficos proporcionando *Plantilla*s de código en un lenguaje conocido con ejecución inmediata.

1.3 Objetivos Específicos

- Desarrollar una extensión de un lenguaje de ejecución inmediata.
- Diseñar e implementar un *parser* para esta extensión.
- Construir un analizador semántico que interprete el código y que haga una traducción a un lenguaje conocido en forma de *Plantilla* del despliegue.
- Implementar en esta extensión las funcionalidades y técnicas básicas para el despliegue de gráficos.
- Realizar una documentación de las funcionalidades que proporciona la extensión y las características del procesamiento de la misma.
- Realizar comparaciones entre los resultados generados por el procesamiento del código escrito en la extensión y el código en forma de *Plantilla* generado por el procesador.

1.4 Solución

La solución propuesta consta de una aplicación que contiene todos los recursos que necesita, es decir, no requiere hacer peticiones a un ente externo en ninguna situación. La propuesta estará formada por un intérprete de un subconjunto del lenguaje de programación Lua más una extensión del mismo, escrito en lenguaje JavaScript. La aplicación también contará con un módulo denominado desplegador de gráficos para el navegador, el cual despliega lo que indica el intérprete. Además, permitirá generar un código en lenguaje JavaScript con el despliegue equivalente al generado por la aplicación al interpretar el código en lenguaje Lua.

Para hacer uso de esta aplicación el desarrollador debe abrir una página en cualquier navegador y utilizar la interfaz que provee la misma como se ilustra en la Fig. 1. El desarrollador puede escribir su código en lenguaje Lua y desarrollar despliegues utilizando la extensión propuesta, la cual es fácil de aprender y de emplear, además de ser intuitiva por las palabras claves utilizadas. El desarrollador puede ordenar a la aplicación que interprete su código y los resultados podrán ser visualizados hasta que se ordene interpretar de nuevo o se cierre la aplicación. Además puede indicar si desea generar un archivo con el despliegue que desarrolló traducido a lenguaje JavaScript independiente de la aplicación. Para esto la aplicación genera un archivo que contiene todos los recursos que se necesitan para hacer el despliegue, y el desarrollador sólo necesita abrir la página del código generado con un navegador para ver exactamente el mismo despliegue que visualizó al desarrollarlo empleando la aplicación.

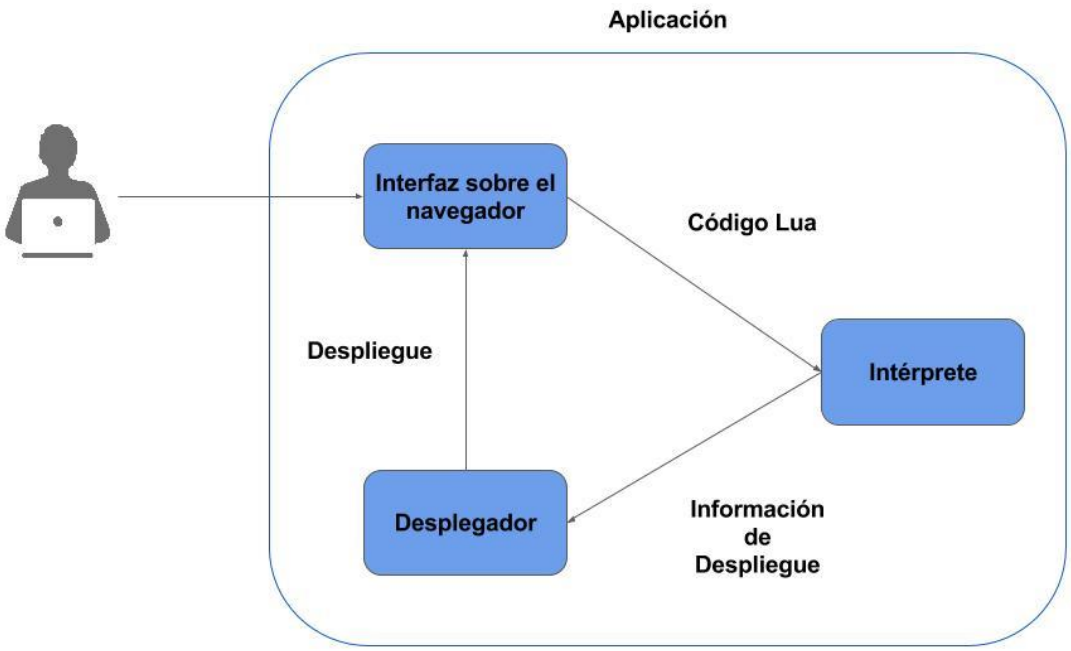


Figura 1. Esquema de uso de la aplicación.

1.5 Justificación e Importancia

Sería de gran importancia crear un lenguaje o una extensión de uno, el cual facilite el desarrollo de gráficos a través de pocas líneas de código. Debe contar con su propio procesador que pueda arrojar resultados visuales inmediatos, de manera que el programador pueda verificar su implementación mientras se está desarrollando. Igualmente, es ideal que sea accesible para todo usuario.

Del mismo modo, debe proporcionar una *Plantilla* de implementación de despliegue de gráficos en un lenguaje conocido para permitir al usuario mejorar o innovar sobre técnicas de despliegue. Al mismo tiempo que sea extensible, dinámico en estructura y estructurado.

1.6 Alcance

Se asegura el correcto funcionamiento de todas las características y funcionalidades presentes en la aplicación al utilizar los navegadores: Google Chrome, Mozilla Firefox y Opera.

La aplicación interpreta un subconjunto del lenguaje Lua más una extensión, y este se muestra a continuación:

```
start ::= chunk
chunk ::= {stat}[last_stat]
last_stat ::= 'return' [exp_list][';']
stat ::= 'local' name_list ['=' exp_list] |';' |var_list '=' exp_list |func_call |
'while' exp 'do' chunk {'elseif' exp 'then' chunk} ['else' chunk] |
'for' name '=' exp ',' exp [',' exp] 'do' chunk 'end' |'repeat' chunk 'until' exp |
'function' func_name func_body | 'break' | line_comment | long_comment
func_name ::= name
var_list ::= var {',' var}
var ::= name {('.' name | '[' exp ']')}
name_list ::= name {',' name}
exp_list ::= exp {',' exp}
exp ::= exp bin_op exp | un_op exp | '(' exp ')' |'nil' |'true' |'false' |
string |char_string |numeral | var | func_call | hash_const
bin_op ::= 'or' | 'and' | '==' | '~=' | '>=' | '<=' | '>' | '<' | '|' |
'~' | '&' | '>>' | '<<' | '..' | '-' | '+' | '%' | '//' | '/' | '*' | '^'
un_op ::= '~' | '-' | '#' | 'not'
func_call ::= var args
args ::= '(' [exp_list] ')'
hash_const ::= '{' [field_list] '}'
field_list ::= field {field_sep field}
field_sep ::= ';' | ','
field ::= '[' exp ']' '=' exp | name '=' exp | exp
func_body ::= '(' [name_list] ')' chunk 'end'
```

Figura 2. Subconjunto del lenguaje Lua.

```
DrawCube (String displayMode);
DrawCone (String displayMode);
DrawSphere (String displayMode);
DrawCylinder (String displayMode);
DrawGrid (String displayMode);
DrawObject (String displayMode);
TranslateObject (Array vector);
RotateObject (Number angle, Array vector);
ScaleObject (Array vector);
DrawPointLight (Array pos);
DrawDirectionalLight (Array pos, Array dir);
DrawSpotLight (Array pos, Array dir, Number cutoff, Number exponent);
ChangeLighting (String model);
AmbientalComponent (Array vector);
DiffuseComponent (Array vector);
SpecularComponent (Array vector);
```

Figura 3. Extensión propuesta.

La aplicación permite el despliegue de figuras 2D y 3D, ya sean predeterminados o modelos en formato *obj*. Además soporta el modelo de iluminación Blinn-Phong y los modelos de sombreado Phong, Gouraud y Flat sobre toda la escena, así como el cambio de componentes ambiental, difuso y especular tanto en modelos como luces. Además, para los componentes se ofrece el uso del formato RGB ya sea con rango de 0 a 1, de 0 a 255 o utilizando colores predefinidos con nombre definidos por Cascade Style Sheet, y soportados por todos los navegadores modernos. También soporta transformaciones de traslación, rotación y escalado tanto en modelos como en luces. Por último, en caso de no utilizar ningún modelo de iluminación la aplicación despliega los objetos (modelos o luces) utilizando las normales como el color para ayudar a detallar correctamente la forma del objeto.

Capítulo 2

Marco Teórico

Este capítulo contempla la teoría involucrada en el desarrollo de la solución propuesta, con el fin de comprender las técnicas utilizadas en el mismo. Primero se expondrá el procesamiento de los lenguajes y su extensibilidad, seguido de los API's para el despliegue de gráficos y por último las técnicas de iluminación y sombreado.

2.1 Extensibilidad en Lenguajes

Una extensión añade nuevas características, funciones, identificadores o incluso nueva sintaxis al lenguaje original. Cuando un lenguaje define formas para extenderse a sí mismo se le conoce como Lenguaje Extensible. La mayoría de los lenguajes, hoy en día, son extensibles en cierto grado. Por ejemplo, casi todos los lenguajes permiten al programador definir nuevos tipos de datos y nuevas operaciones (funciones o procedimientos). Algunos lenguajes permiten al programador incluir estos nuevos recursos en una unidad de programa más grande, como paquetes o módulos (Louden & Lambert, 2011).

Los diseñadores de lenguajes modernos como Java y Python también extienden las características del lenguaje a través del lanzamiento de nuevas versiones del mismo regularmente. Estas nuevas versiones de lenguajes normalmente son compatibles hacia atrás con programas escritos en versiones anteriores del lenguaje, sin embargo algunas características se vuelven obsoletas, lo que significa que pueden no ser soportadas en versiones futuras del lenguaje.

Muy pocos lenguajes permiten al programador agregar sintaxis y semántica al lenguaje en sí. Un ejemplo es LISP, donde los programadores no sólo pueden añadir funciones, clases y tipos de datos nuevos, también pueden extender la sintaxis y la semántica de Lisp a través de macros. Una macro especifica la sintaxis de una porción de código que expande a otro código estándar en Lisp, cuando el intérprete o compilador encuentra la primera porción de código en un programa. Después, el código expandido es evaluado de acuerdo a las reglas semánticas nuevas. Por ejemplo, Lisp incluye un ciclo do bastante común para iterar, pero no incluye un ciclo while. Eso no es problema para un programador que utiliza Lisp, pues este puede simplemente definir una macro que le indica al intérprete lo que debe hacer cuando encuentre un ciclo while en un programa.

2.2 Procesamiento de los lenguajes de programación

De acuerdo con (Aho, Lam, Sethi & Ullman, 2008; Sebesta, 2012) los lenguajes de programación son notaciones que describen los cálculos a las personas y las máquinas. Nuestra percepción del mundo que vivimos depende de los lenguajes de programación, ya que todo el software que se ejecuta en todas las computadoras se

escribió en algún lenguaje de programación. Pero antes de poder ejecutar un programa, primero debe traducirse a un formato que una computadora pueda ejecutarlo. Existen tres principales procesamientos (sin tomar en cuenta híbridos o combinaciones de estos) que se pueden aplicar a los lenguajes de programación, ya sea para ejecutar el programa o para otros fines, y serán discutidos a continuación.

2.2.1 Compilación

Se le llama compilación al proceso de traducción de un programa escrito en un lenguaje origen a un programa equivalente en otro lenguaje, llamado lenguaje destino. El programa encargado de realizar este proceso se conoce como Compilador. En el caso de querer ejecutar un programa, el lenguaje destino sería el lenguaje de máquina. Este procesamiento tiene una gran ventaja en ese caso, pues una vez terminada la traducción del programa la ejecución del mismo será muy rápida.

El proceso de compilación y ejecución de un programa comprende varias fases, tal como se observa en la Fig. 4, donde cada una de las cuales transforma una representación del programa fuente en otro. En la Fig. 4 se ilustran las fases con su respectiva entrada y salida en cada una.

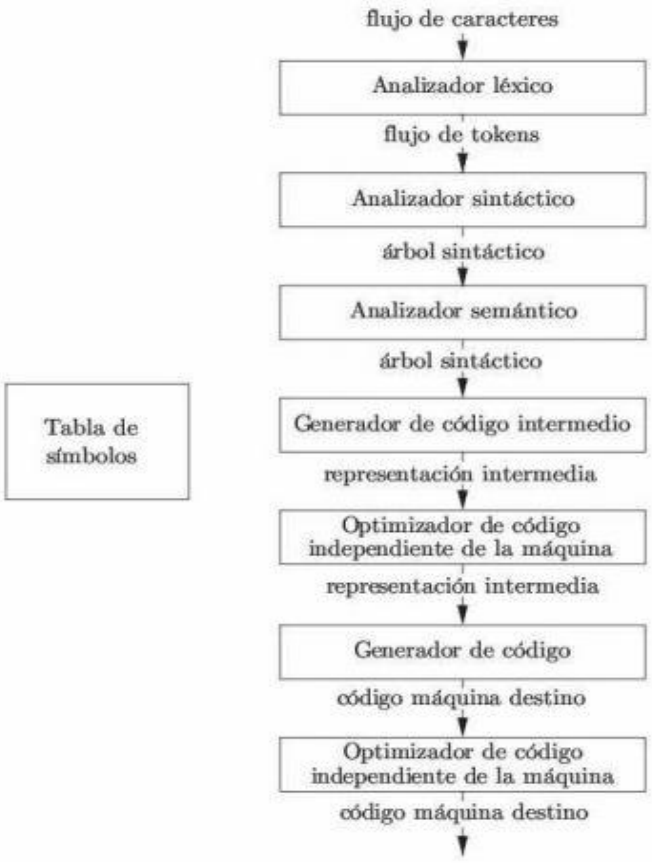


Figura 4. Etapas del proceso de compilación.

Análisis Léxico. A la primera fase de un compilador se le llama análisis léxico o escaneo. El analizador léxico escanea o lee el flujo de caracteres que componen el programa fuente y los agrupa en secuencias significativas, conocidas como lexemas. Los lexemas de un programa son identificadores, palabras clave, operadores, símbolos de puntuación y en algunos los espacios y tabulaciones también forman parte de los lexemas. Para cada lexema, el analizador léxico produce como salida un *token* de la forma:

(*nombre-token, valor-atributo*)

que pasa a la fase siguiente, el análisis de la sintaxis. En el token, el primer componente *nombre-token* es un símbolo abstracto que se utiliza durante el análisis sintáctico, y el segundo componente *valor-atributo* apunta a una entrada en la tabla de símbolos, a la cual se hará referencia como *TS*, para este token. La información de la entrada en la *TS* se necesita para el análisis semántico y la generación de código. La *TS* es una estructura de datos que contiene un registro para cada nombre de variable, con campos para los atributos del nombre. La estructura de datos debe diseñarse de tal forma que permita al compilador buscar el registro para cada nombre, y almacenar u obtener datos de ese registro con rapidez

Análisis Sintáctico. La segunda fase del compilador es el análisis sintáctico o *parsing*. El parser (analizador sintáctico) utiliza los primeros componentes de los tokens producidos por el analizador léxico para crear una representación intermedia en forma de árbol que describe la estructura gramatical del flujo de tokens. Una representación típica es el árbol sintáctico, donde cada nodo interior representa una operación y los hijos del nodo representan los argumentos de la operación.

El análisis sintáctico se divide en dos principales tipos, descendente y ascendente.

El análisis sintáctico descendente puede verse como un análisis que construye un árbol para la cadena de entrada, partiendo desde la raíz (la parte superior) y creando los nodos del árbol en preorden.

Un análisis sintáctico ascendente corresponde a la construcción de un árbol de análisis sintáctico para una cadena de entrada que empieza en las hojas (la parte inferior) y avanza hacia la raíz (la parte superior).

Análisis Semántico. El analizador semántico utiliza el árbol sintáctico y la información en la *TS* para comprobar la consistencia semántica del programa fuente con la definición del lenguaje. También recopila información sobre el tipo y la almacena, ya sean en el árbol sintáctico o en la *TS*, para usarla más tarde durante la generación de código intermedio.

Una parte importante del análisis semántico es la comprobación (verificación) de tipos, en donde el compilador verifica que cada operador contenga operandos que coincidan. Por ejemplo, muchas definiciones de lenguajes de programación

requieren que el índice de un arreglo sea entero; el compilador debe reportar un error si se utiliza un número de punto flotante para tener acceso al arreglo.

La especificación del lenguaje puede permitir ciertas conversiones de tipo conocidas como coerciones. Por ejemplo, puede aplicarse un operador binario aritmético a un par de enteros o a un par de números de punto flotante. Si el operador se aplica a un número de punto flotante y a un entero, el compilador puede convertir u obligar a que se convierta en un número de punto flotante.

Generación de código intermedio. En el proceso de traducir un programa fuente a código destino, un compilador puede construir una o más representaciones intermedias, las cuales pueden tener una variedad de formas. Los árboles sintácticos son una forma de representación intermedia; por lo general, se utilizan durante el análisis sintáctico y semántico.

Después del análisis sintáctico y semántico del programa fuente, muchos compiladores generan un nivel bajo o explícito, o una representación intermedia similar al código de máquina, que podemos considerar como un programa para una máquina abstracta. Esta representación intermedia debe tener dos propiedades importantes: debe ser fácil de producir y fácil de traducir en la máquina destino.

Optimización de código. La fase de optimización de código independiente de la máquina trata de mejorar el código intermedio, de manera que se produzca un mejor código destino. Por lo general, mejor significa más rápido, pero pueden lograrse otros objetivos, como un código más corto, o un código destino que consuma menos recursos.

Hay una gran variación en la cantidad de optimización de código que realizan los distintos compiladores. En aquellos que se realizan la mayor optimización, a los denominados “compiladores optimizadores”, se invierte mucho tiempo en esta fase. Hay optimizaciones simples que mejoran en forma considerable el tiempo de ejecución del programa destino, sin reducir demasiado la velocidad de la compilación.

Generación de código. El generador de código recibe como entrada una representación intermedia del programa fuente y la asigna al lenguaje destino. Si el lenguaje destino es código de máquina, se seleccionan registros o ubicaciones (localidades) de memoria para cada una de las variables que utiliza el programa. Después, las instrucciones intermedias se traducen en secuencias de instrucciones de máquina que realizan la misma tarea. Un aspecto crucial de la generación de código es la asignación juiciosa de los registros para guardar las variables.

2.2.2 Interpretación

La interpretación según (Sebesta, 2012) se encuentra en el extremo opuesto (comparado con compilación) de los tipos de procesamientos. Con este enfoque, los programas son interpretados por otro programa llamado intérprete, sin ninguna traducción, como se ilustra en la Fig. 5. El intérprete actúa como un software de simulación de la máquina, su ciclo de ejecución soporta instrucciones de programas escritos en lenguajes de alto nivel en vez de instrucciones de máquina. Este software de simulación provee una máquina virtual para el lenguaje en cuestión.

La interpretación tiene la ventaja de permitir una fácil implementación de muchas operaciones de depuración. Por ejemplo, si un índice de arreglo se encuentra fuera de rango, el mensaje de error puede fácilmente indicar la línea origen y el nombre del arreglo. Por otra parte, este procesamiento tiene la seria desventaja de que la ejecución es de 10 a 100 veces más lenta que al utilizar la compilación. La causa principal de esta lentitud es la decodificación de instrucciones de lenguajes de alto nivel, las cuales son mucho más complejas que las instrucciones de máquina (aunque pueda haber menos instrucciones que el equivalente en instrucciones de máquina). Además, sin importar cuántas veces una instrucción es ejecutada, debe ser decodificada cada vez.

Otra desventaja de la interpretación es que usualmente requiere más espacio. Además del programa fuente, la *TS* debe estar presente durante la interpretación.

Aunque algunos de los primeros lenguajes simples de los años 60 (ASP, SNOBOL, y LISP) eran interpretados, para los años 80, el enfoque era raramente utilizado en lenguajes de alto nivel. Sin embargo, en los años recientes, la interpretación volvió con algunos lenguajes web de scripting, como JavaScript y PHP, los cuales son ampliamente utilizados. El proceso de interpretación se ilustra en la Fig. 5.

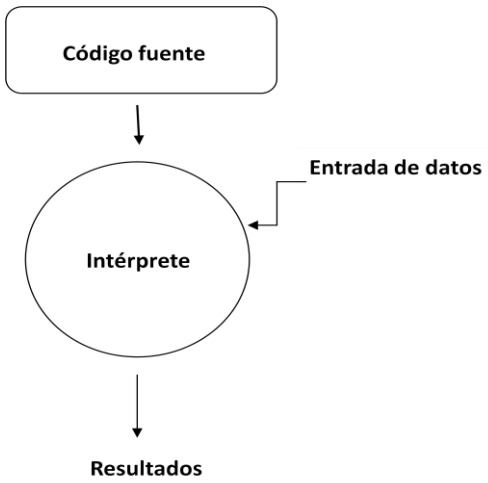


Figura 5. Proceso de interpretación.

2.2.3 Traducción

La compilación se puede definir también como una traducción de un lenguaje de alto nivel al nivel de máquina, la misma tecnología puede aplicarse para realizar traducciones entre distintos tipos de lenguajes. A continuación se muestran algunas de las aplicaciones más importantes de las técnicas de traducción de programas (Aho, Lam, Sethi & Ullman, 2008).

Traducción binaria: La tecnología de compiladores puede utilizarse para traducir el código binario para una máquina al código binario para otra máquina distinta, permitiendo a una máquina ejecutar los programas que originalmente eran compilados para otro conjunto de instrucciones. Varias compañías de computación han utilizado la tecnología de la traducción binaria para incrementar la disponibilidad de software en sus máquinas. En especial, debido al dominio en el mercado de la computadora personal x86, la mayoría de los títulos de software están disponibles como código x86. Se han desarrollado traductores binarios para convertir código x86 en código Alpha y Sparc. Además, Transmeta Inc. utilizó la traducción binaria en su implementación del conjunto de instrucciones x86. En vez de ejecutar el complejo conjunto de instrucciones x86 directamente en el hardware, el procesador Transmeta Crusoe es un procesador VLIW que se basa en la traducción binaria para convertir el código x86 en código VLIW nativo.

La traducción binaria también puede usarse para ofrecer compatibilidad inversa.

Síntesis de hardware: No sólo la mayoría del software está escrito en lenguajes de alto nivel; incluso hasta la mayoría de los diseños de hardware se describen en lenguajes de descripción de hardware de alto nivel, como Verilog y VHDL (Very High-Speed Integrated Circuit Hardware Description Language, Lenguaje de descripción de hardware de circuitos integrados de muy alta velocidad). Por lo general, los diseños de hardware se describen en el nivel de transferencia de registros (RTL), en donde las variables representan registros y las expresiones representan la lógica combinacional. Las herramientas de síntesis de hardware traducen las descripciones RTL de manera automática en compuertas, las cuales a su vez se asignan a transistores y, en un momento dado, a un esquema físico. A diferencia de los compiladores para los lenguajes de programación, estas herramientas a menudo requieren horas para optimizar el circuito. También existen técnicas para traducir diseños a niveles más altos, como al nivel de comportamiento o funcional.

2.3 API’s para el Despliegue de Gráficos

2.3.1 OpenGL

OpenGL es una interfaz de software para hardware gráfico. Específicamente, es un API para gráficos 2D y 3D, proporcionando operaciones y abstracciones para el modelado, animación, y visualización. Fue creado para proporcionar capacidades de

alta calidad y de alto rendimiento a las aplicaciones de software gráfico en todas las industrias, como las de difusión, diseño asistido por computadora, fabricación asistida por computadora, entretenimiento, e imágenes médicas. Para dar soporte a ese amplio rango de aplicaciones, OpenGL es compatible con la mayoría de los sistemas operativos, y el API está disponible para los lenguajes de programación C, C++, Java, Fortran y Ada. OpenGL se han convertido en el más ampliamente utilizado, soportado, y documentado GPL desde su introducción en 1992 [Yee, 2004].

OpenGL es mantenido por OpenGL ARB (the OpenGL Architecture Review Board), la cual consiste en representantes de industrias, proporcionando una amplia gama de experiencias y conocimientos para el desarrollo de OpenGL. No solamente diseñaron el estándar OpenGL original, también se aseguran de que las futuras mejoras del lenguaje se adherirán al objetivo original - hacer a OpenGL un estándar de la industria en todas las plataformas - revisando las solicitudes de mejora y haciendo pruebas de conformidad. OpenGL es un verdadero estándar abierto.

Para establecer a OpenGL como un estándar abierto, OpenGL debe ser estable, legible, portable, escalable, que pueda evolucionar, fácil de usar, y bien documentado. Para proveer un lenguaje estable, los desarrolladores de software deben tener tiempo para aprobar cualquier modificación del lenguaje y mantener su compatibilidad hacia atrás. Por lo tanto OpenGL ARB cuidadosamente controla las mejoras del lenguaje y las modificaciones a las especificaciones de OpenGL.

OpenGL hizo muchos requisitos clave de diseño para proporcionar un lenguaje seguro. Primero, el estado del modelado debe ser consistente con la ejecución completa de todos los comandos de OpenGL invocados previamente. OpenGL es un lenguaje secuencial, por como procesa los comandos uno a la vez como son recibidos. Por lo tanto el objeto previo debe ser dibujado completamente, antes de que sea consultado o el siguiente objeto será dibujado por comandos posteriores. Segundo, OpenGL enlaza los datos en las llamadas. Los argumentos de las funciones son interpretados cuando la función es llamada. Incluso si el argumento es un apuntador, los datos del apuntador son interpretados en la llamada de la función, así que cualquier cambio posterior a los datos del apuntador no afectarán a la función. Tercero, los errores punto flotantes no deberían causar que la aplicación termine, desde que OpenGL realiza una cantidad considerable de operaciones punto flotante. Cualquier número es aceptable para cualquier argumento de función punto flotante de OpenGL. Números distintos de punto flotantes producirán resultados inesperados, pero no causarán que la aplicación termine. De la misma forma sucede con la división entre cero, causará resultados inesperados pero no terminará la aplicación.

Para ser un lenguaje portable, OpenGL ARB requirió que todas las implementaciones de OpenGL den el mismo resultado de despliegue sin importar los

sistemas operativos, arquitectura de computadora, o versión del API. Los sistemas operativos soportados son MAC OS, UNIX, Windows y Linux. Las versiones del API para C, C++, Java, Fortran y Ada son probadas a fondo para asegurar que se ajustan a las especificaciones del lenguaje y producen resultados y comportamientos correctos.

Para proveer un lenguaje escalable, las aplicaciones de OpenGL pueden ejecutarse en sistemas de cualquier clase, desde dispositivos electrónicos portátiles hasta súper computadoras. Para este propósito, dos diferentes implementaciones no siempre tienen que estar de acuerdo en una base pixel a pixel, pero si tienen que aproximar la misma salida.

Para proporcionar un lenguaje que evolucione, OpenGL tiene un mecanismo de extensión que permite a los desarrolladores de software acceder a nuevas capacidades del hardware. Este mecanismo permite a los desarrolladores de software estar al corriente del estándar actual de OpenGL para así aprovechar tanto el nuevo hardware como las nuevas tecnologías sin esperar por una mejora de la especificación de OpenGL. Lo que permite a los desarrolladores de software adaptar OpenGL a sus necesidades específicas, las cuales pueden incluir la obtención de mayor rendimiento del hardware o portar la aplicación a un dispositivo electrónico de uso diario. Las diferentes implementaciones de OpenGL pueden dividir el tiempo de procesamiento entre la unidad de procesamiento gráfico (GPU) y la unidad central de procesamiento (CPU), dependiendo de la función. Este mecanismo también permite a los desarrolladores de software adaptar al usuario y explícitamente ejecutar ciertos comandos de OpenGL en la GPU o CPU.

Para ser un lenguaje fácil de usar, OpenGL ha sido diseñado para ser un lenguaje de modelado, que contiene rutinas simples y eficientes para desarrollar software gráfico. Incluye desde funcionalidades para crear figuras geométricas simples (ejemplo, líneas y polígonos) hasta modelos de iluminación muy complejos. Permite la manipulación de cómo son renderizados los objetos a través de transformaciones de matrices, funciones de colorado y mezcla de ellos, y otras características. Está diseñado y estructurado para ser intuitivo para cualquier programador gráfico. El lenguaje esconde cualquier detalle del sistema operativo (ejemplo, creando ventanas) e información específica del hardware (ejemplo, el manejo de frame buffers que almacenan la escena actual), para que el programador pueda concentrarse en la parte gráfica de la aplicación y producir programas con menos líneas de código. Sin embargo, el ámbito de OpenGL abarca sólo lo que se renderiza en el frame buffer. No soporta periféricos que usualmente están asociados con hardware gráfico como el mouse y el teclado. Los desarrolladores deben utilizar otros métodos para manejar esos dispositivos. Y mientras OpenGL tiene comandos para controlar qué tan compleja debe ser renderizada la geometría (ejemplo, color, transformaciones), no tiene comando para crear figuras geométricas complejas.

2.3.2 Direct3D

Microsoft DirectX es un conjunto de tecnologías para el diseño e implementación de aplicaciones basadas en Windows. Contiene APIs para la programación de gráficos, video, música y sonido en C, C++ y Visual Basic. Los APIs gráficos están incluidos en el componente Direct3D del conjunto DirectX. Direct3D fue originalmente creado por desarrolladores de juegos 3D y fue posteriormente comprado por Microsoft en 1996.

Direct3D es un lenguaje de alto nivel para el modelado 3D con API's de bajo nivel para acceder a las capacidades del hardware gráfico. Direct3D tiene muchos de los mismos objetivos de diseño que OpenGL, principalmente fácil de usar, portabilidad, componibilidad, rendimiento, extensibilidad y compatibilidad hacia atrás. Microsoft quería alentar a los desarrolladores a hacer aplicaciones para su sistema operativo Windows. Debido a que Microsoft Windows originalmente no soportaba gráficos 3D, los vendedores de software tuvieron tiempos difíciles al hacer aplicaciones y juegos 3D para Windows. Así que Microsoft sacó Direct3D para proporcionar un ambiente de desarrollo más fácil, proporcionando un lenguaje consistente con sus componentes de modelo de objetos (Component Object Model) en DirectX. Direct3D también soporta portabilidad entre arquitecturas de computadoras. Previo a Direct3D, los vendedores tenían que personalizar sus aplicaciones para configuraciones diferentes de hardware. Sin embargo Direct3D permite a los desarrolladores de software acceder al hardware gráfico a través de un solo API, sin importar la arquitectura de hardware subyacente. Con Direct3D, los desarrolladores de software pueden hacer gráficos de alto rendimiento en cualquier sistema basado en Windows. También expone algunas de las interfaces de hardware de bajo nivel y proporciona mecanismos de extensión para desarrolladores que necesitan rendimiento extra. Con cada versión, Direct3D ha mantenido su compatibilidad hacia atrás, continuando el soporte de todas las interfaces y objetos incluidos en versiones anteriores.

En contraste a OpenGL, Direct3D es propiedad de Microsoft y mantenido por el mismo para el desarrollo en Windows. Aunque Windows es el sistema operativo más ampliamente utilizado para computadoras domésticas, es tan solo uno de los muchos sistemas operativos. Direct3D no soporta usuarios en UNIX, Linux, y MAC OS.

Un gran beneficio de Direct3D es que está en el paquete del conjunto de tecnologías DirectX, permitiendo que el software Direct3D interactúe fácilmente con otras características de Windows. Con las tecnologías de DirectX, los programadores pueden acceder a tarjetas de sonido, memoria, dispositivos de entrada, y capacidades de red. Por si solo Direct3D puede no ser tan poderoso como OpenGL, pero empaquetado con DirectX, es el lenguaje gráfico de primer nivel para el software basado en Windows.

2.3.3 RenderMan

RenderMan es un API de renderizado desarrollado por Pixar a finales de los años 80, un momento en el que ellos estaban desarrollando hardware de renderizado personalizados. Ellos estaban preocupados en que la más amplia gama de usuarios pudiera hacer uso de su sistema, y por lo tanto entró en discusiones con otras grandes empresas de gráficos. El resultado de esta negociación fue la publicación del estándar RenderMan. La idea era que cualquiera pudiera desarrollar software de modelado que pudiera entenderse con el nuevo hardware de Pixar. Además, cualquiera podría construir un sistema de renderizado en concordancia con el estándar. Cualquier modelador que cumpla con el estándar sería capaz de renderizar sus imágenes utilizando cualquier renderizador que cumpla con el estándar [Stephenson, 2007].

Cuando el estándar RenderMan se propuso por primera vez, la computación gráfica aún era un tópico esotérico de investigación practicado por los proponentes calificados. La expectativa típica era que estos usuarios estarían escribiendo sus propios programas para generar geometría, probablemente en el lenguaje de programación C. Como resultado, la primera versión del API RenderMan definía un conjunto de funciones en C que podían ser llamadas por programas de modelado para pasar instrucciones a un renderizador.

Mientras que el API en C es un mecanismo apropiado para que los investigadores usen al momento de comunicarse con un renderizador, rápidamente estuvo claro que para la producción comercial era requerido un mecanismo más flexible. Los usuarios necesitan ser capaces de generar una escena en una máquina, y luego pasarla al renderizador de su elección. Por esta razón la corriente del formato de archivo RenderMan Interface Byte (RIB) fue introducida.

Un programa de modelado seguirá haciendo llamadas a un API en C internamente, pero en vez de renderizar, esto creará un archivo RIB. Estos archivos RIB luego pueden ser examinados, modificados, y finalmente ser pasados a un programa de renderizado, que tal vez podría estar corriendo en una máquina completamente diferente.

La interfaz entre el modelador y el renderizador es se encuentra en la forma de un archivo RIB, Como resultado, es probablemente razonable decir que el formato de archivo RIB representa el estándar de RenderMan - un modelador compatible con RenderMan escribe RIBs mientras que un renderizador compatible con RenderMan lee RIBs.

Mientras que es perfectamente posible simplemente confiar en un paquete de modelado para generar RIBs, una gran cantidad de poder y flexibilidad se puede ganar de inclusive un conocimiento limitado del formato RIB. Los archivos RIB son generalmente almacenados como texto y pueden ser fácilmente ser creados o modificados por un usuario. A pesar de que estos tienden a contener una gran

cantidad de números representando puntos en el espacio 3D los cuales son difíciles de interpretar manualmente, la estructura general consiste en una lista de comandos simples que pueden fácilmente ser identificados y entendidos. Incluso trabajando enteramente en el paquete de modelado, la interfaz normalmente hace referencia a características a nivel de RIB.

A pesar de que los RIBs definen la geometría de una escena, RenderMan distingue entre la figura de un objeto y el detalle de su superficie. La mayoría de los objetos son geométricamente simples, y pueden ser representados por geometrías muy primitivas. Por ejemplo, una naranja es básicamente una esfera. Sin embargo, los objetos reales difieren de los objetos generados por computadora porque al examinar detalladamente estos muestran texturas complejas en su superficie, e interactúan con la luz en un rango interesante.

Mientras todos los sistemas de renderizado modernos proporcionan al usuario maneras para controlar las propiedades de la superficie de un objeto, el estándar RenderMan va más allá al definir sistema altamente complejo y flexible donde las propiedades de la superficie de un objeto son definidas por Shaders. Estos toman la forma de pequeñas piezas de código de computadora escritas en un lenguaje específico de RenderMan llamado SL - abreviatura para Shading Language. Un renderizador típicamente será suministrado con un programa para convertir shaders escritos en este lenguaje en un formato que pueda usar ese renderizador en particular. Esto es conocido como un compilador de shaders.

Cuando una superficie es definida en un archivo RIB, es simplemente marcada como que tiene un shader en particular adjunto. Cuando la escena es renderizada, el renderizador buscará ese shader, y utilizará el código que contiene para calcular la apariencia de la superficie. La habilidad de tener ese buen control sobre la apariencia de una superficie es lo que hace a los renderizadores RenderMan tan poderosos.

2.3.4 WebGL

WebGL es un API derivado de una versión de OpenGL diseñada específicamente para dispositivos como teléfonos inteligentes y consolas de videojuegos. Esa versión, conocida como OpenGL ES, fue desarrollada entre el 2003 y 2004, y fue actualizada en el 2007 (a ES 2.0) seguida de otra actualización en el 2012 (a ES 3.0). WebGL está basado en la versión ES 2.0 [Kouichi Matsuda, Rodger Lea, 2013].

En los últimos años, el número de dispositivos y procesadores que soportan WebGL ha incrementado rápidamente, y ahora incluye teléfonos inteligentes (Android e iOS), tabletas y consolas de videojuegos. En parte, la razón de aprobación de este API ha sido porque OpenGL ES añadió nuevas características y removió muchas características obsoletas e innecesarias de OpenGL. Resultando en un API ligero

que aún tenía un poder expresivo visual suficiente para realizar gráficos 3D atractivos.

La Fig. 6 ilustra la relación entre OpenGL, OpenGL ES 1.1 /2.0 / 3.0 y WebGL. Debido a que OpenGL continuó evolucionando de 1.5 a 2.0 y de 2.0 a 4.3, OpenGL ES se estandarizó como un subconjunto de versiones específicas de OpenGL (1.5 y 2.0).

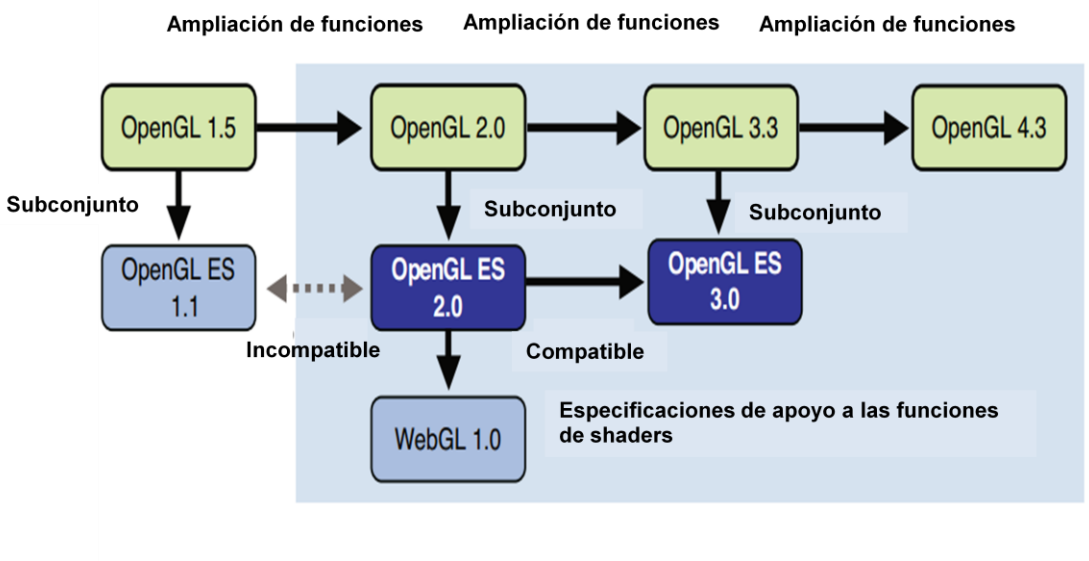


Figura 6. Relación entre OpenGL, OpenGL ES 1.1/ 2.0 / 3.0 y WebGL.

Como se muestra en la Fig. 6, el avance a la versión 2.0 de OpenGL trajo nuevas capacidades significativas, se introdujo **programmable shader functions**.

Shader functions o **shaders** son programas de computadora que hacen posible el desarrollo de efectos visuales sofisticados utilizando un lenguaje de programación especial similar al lenguaje C. Este se conoce como **shading language** (lenguaje de sombreado). El lenguaje de sombreado utilizado en OpenGL ES 2.0 está basado en el **OpenGL shading language** (GLSL) y es conocido como **OpenGL ES shading language** (GLSL ES). Debido a que WebGL está basado en OpenGL ES 2.0, también utiliza GLSL ES para crear **shaders**.

El grupo **Khronos** es responsable de la evolución y estandarización de OpenGL. En el 2009, **Khronos** estableció el grupo de trabajo de WebGL y luego comenzó el proceso de estandarización de WebGL basado en OpenGL ES 2.0, publicando la primera versión de WebGL en el 2011.

Las aplicaciones WebGL son creadas utilizando tres (3) lenguajes: HTML5, JavaScript y GLSL ES. La Fig. 7 ilustra la arquitectura de **software** de una aplicación web y aplicaciones WebGL.

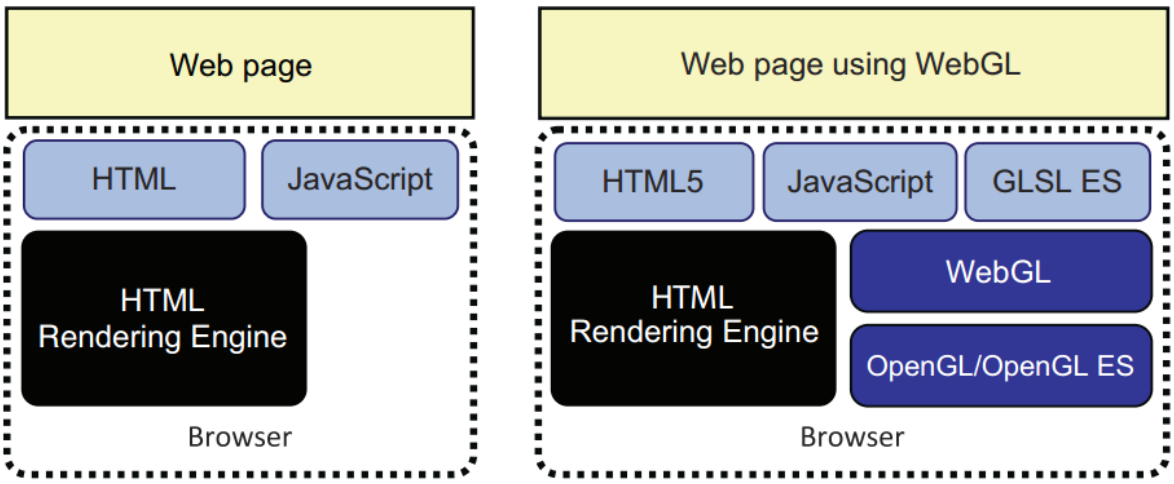


Figura 7. Arquitectura de una aplicación web (lado izquierdo) y de una aplicación WebGL (lado derecho).

WebGL es un API en lenguaje JavaScript que puede ser ejecutado en los navegadores web más modernos. Así, el usuario puede visitar la dirección de una aplicación WebGL desde un sistema remoto, y el programa se ejecutará en el sistema local y hará uso del **hardware** gráfico local. Sin embargo, WebGL no posee todas las características de las últimas versiones de OpenGL, sólo las básicas [Edward Angel, Dave Shreiner, 2014].

2.4 Fuentes de luz

Para lograr el efecto de iluminación se utilizan fuentes de luz, las cuales poseen diferentes características. Las tres (3) principales fuentes son luz puntual, luz direccional y luz concentrada [Shreiner, Sellers, Kessenich, 2013].

2.4.1 Luz puntual

La luz puntual imita fuentes de luz que estén cerca de la escena o dentro de ella, como lámparas. Se define como un punto que emite energía en todas direcciones, por lo que cada punto de la superficie iluminada tendrá una dirección diferente como se ilustra en la Fig. 8.

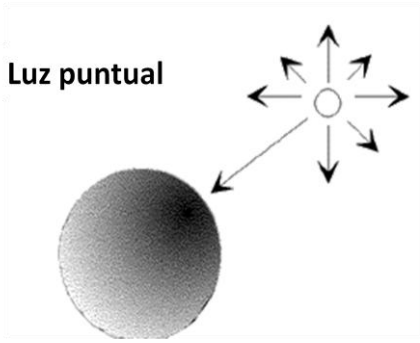


Figura 8. Esquema del funcionamiento de la Luz puntual.

2.4.2 Luz direccional

La luz direccional imita una fuente de luz muy lejana, la cual incide aproximadamente en todos los puntos de la superficie con la misma dirección. Esta características hacen de la luz direccional una fuente de luz fácil de implementar y más rápida de calcular. La Fig. 9 ilustra una luz direccional.

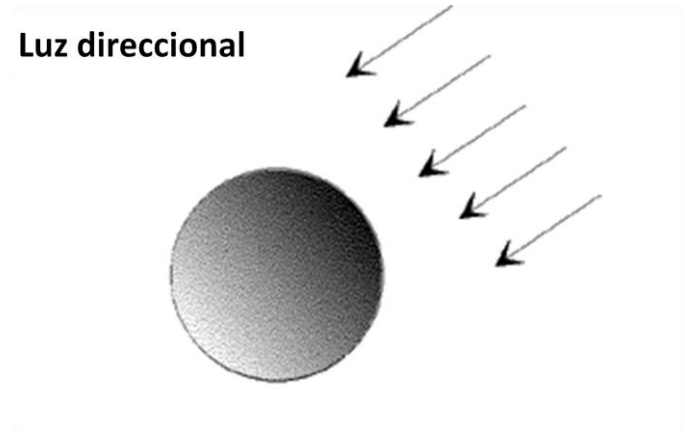


Figura 9. Esquema del funcionamiento de la Luz direccional.

2.4.3 Luz concentrada

La luz concentrada proyecta un fuerte rayo de luz que ilumina un área bien definida. Dicha área puede iluminarse de manera degradada hacia los límites del área, creando una forma de cono como se ilustra en la Fig. 10.

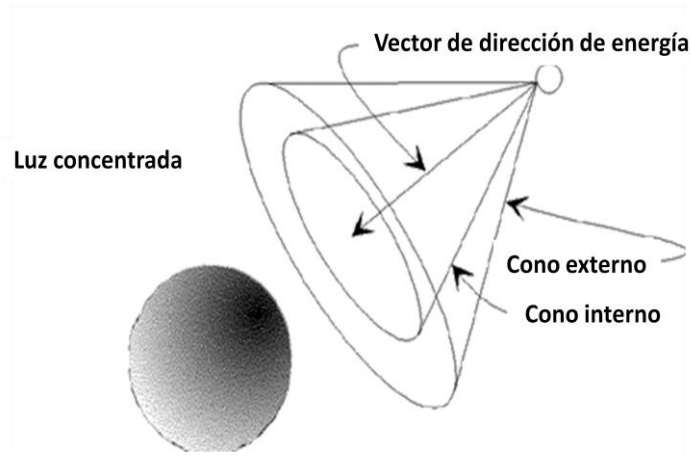


Figura 10. Esquema del funcionamiento de la Luz concentrada.

La luz concentrada emite energía en diferentes direcciones, y a diferencia de la luz puntual se emite de acuerdo al área donde se concentrará la energía o mejor conocida como área de foco.

2.5 Modelos de sombreado para polígonos

Los modelos de sombreado calculan el color de cada fragmento de los polígonos. Para ellos, se conocen tres (3) principales modelos:

- **Flat Shading**
- **Gouraud Shading**
- **Phong Shading**

2.5.1 Flat Shading

Cada vértice es seleccionado como un vértice clave, luego se hace el cálculo de la iluminación en ese vértice y el triángulo completo es rellenado con una copia de esa iluminación calculada para el vértice clave. Este utiliza la normal del polígono para calcular la iluminación correspondiente. La Fig. 11 ilustra un ejemplo utilizando el modelo **Flat Shading** [Hughes, Dam, Mcguire, Sklar, Foley, Feiner, Akeley, 2014].

Figura 11. Ejemplo visual de **Flat Shading** sobre una esfera.

2.5.2 Gouraud Shading

Este modelo aplica diferentes colores por vértice. Para esto, se calcula la normal de cada vértice, la cual se puede calcular con un promedio ponderado de las normales de los polígonos en los que se encuentre el vértice. Seguidamente, el polígono obtendrá un color equivalente a la interpolación de los colores por vértice. Como se ilustra en la Fig. 12, el resultado obtenido es mejor que el modelo **Flat Shading**.

Figura 12. Ejemplo visual de **Gouraud Shading** sobre una esfera.

2.5.3 Phong Shading

El modelo **Phong Shading** también utiliza las normales por vértice, pero hace el cálculo de iluminación por fragmento. Para obtener las normales en cada punto donde se calcule la iluminación, las normales por vértice son interpoladas bilinealmente. En la Fig. 13 se ilustra un ejemplo utilizando el modelo **Phong Shading**, del cual se obtienen mejores resultados que incluso el modelo **Gouraud Shading**.

Figura 13. Ejemplo visual de **Phong Shading** sobre una esfera.

2.6 Sistemas de Despliegue Gráfico en la Web

2.6.1 Three.js

Three.js es un API en JavaScript que simplifica el código de despliegue de gráficos. Utiliza WebGL para el despliegue de gráficos en los navegadores. Soporta muchas de las mismas características que Babylon.js y algunas otras como efectos de anaglifos, acceso completo a las capacidades del lenguaje de **shading** de OpenGL (GLSL), etc. Utiliza el mismo formato de archivo JSON. En la Fig. 14 se ve reflejada una escena hecha en Three.js (http://goo.gl/Vz5b53).

Figura 14. Ejemplo de escena utilizando Three.js.

2.6.2 Babylon.js

Babylon.js es un **framework** completo en lenguaje JavaScript para despliegue de gráficos en la Web, utilizando HTML5 y WebGL. En la Fig. 15 se puede apreciar un ejemplo obtenido con Babylon.js.

Babylon.js soporta una gran cantidad de características como escenas, luces, cámaras, materiales, sistema de detección de colisiones, **picking**, **frustum clipping**, **antialiasing**, etc. En cuanto a los materiales soporta difuso, especular, ambiental, opacidad y hasta cuatro luces simultáneas. Proporciona cuatro diferentes tipos de luces y su formato de archivo es JSON, el cual puede ser producido a partir de los formatos .OBJ, .FBX, .MXB y Blender (http://goo.gl/6Ykzn8).

Figura 15. Escena 3D utilizando Babylon.js.

2.6.3 SpiderGL

SpiderGL es un API que simplifica el despliegue de gráficos en lenguaje JavaScript. Utiliza WebGL para el despliegue de gráficos en los navegadores. También soporta muchas de las características que soportan Babylon.js y Three.js (Benedetto, Ponchio, Ganovelli & Scopigno, 2010).

SpiderGL fue diseñado tomando en cuenta tres cualidades fundamentales:

> **Eficiencia**. Con JavaScript y WebGL, la eficiencia no es sólo cuestión de algoritmos mejorados, sino la manera de encontrar el mecanismo más eficiente para implementar los gráficos, por ejemplo carga asíncrona o pase de parámetros a **shaders** sin sobrecargar el CPU.
>
> **Simplicidad y una sencilla curva de aprendizaje**. Los usuarios deberían poder reutilizar lo más que puedan sus conocimientos en el campo y tomar ventaja del API rápidamente. Por esto SpiderGL tiene una correspondencia uno a uno con los comandos de OpenGL y GLU.
>
> **Flexibilidad**. SpiderGL no trata de enmascarar las funciones nativas de WebGL, en vez de eso proporciona funcionalidades de nivel superior que satisfacen las necesidades más comunes de los desarrolladores de gráficos.

En la Fig. 16 se puede apreciar una escena realizada con SpiderGL.

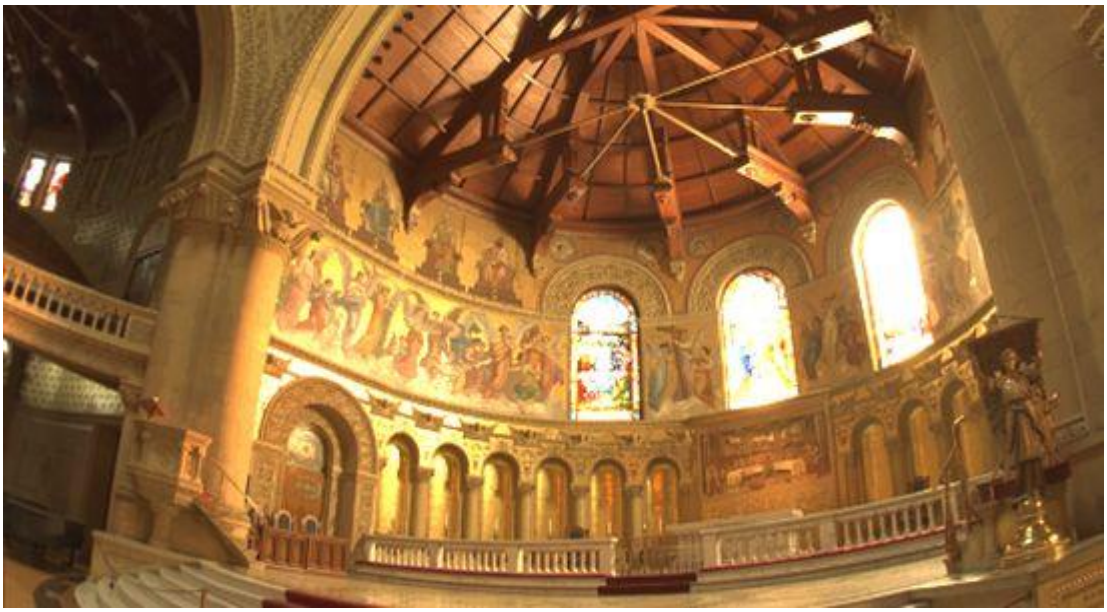


Figura 16. Escena realizada con SpiderGL.

Estos sistemas de despliegue gráfico en la web contribuyen a la facilitación de las herramientas de despliegue gráfico, así como a la innovación de estándares. Sin embargo, la solución propuesta provee un enfoque distinto, el cual toma en cuenta la complejidad de la sintaxis de los lenguajes y la necesidad de obtener resultados de manera inmediata.

Capítulo 3

Marco Metodológico

La metodología utilizada fue *modelo incremental*. Esta se basa en la filosofía de desarrollar incrementando las funcionalidades del programa. Para esto, se desarrollan módulos o funcionalidades que satisfacen conjuntos de requisitos por separado, y se van incrementando la cantidad de requisitos satisfechos a medida que se desarrollen más módulos [Laboratorio Nacional de Calidad del Software, 2009].

Cuando se utiliza un *modelo incremental*, el primer incremento se encarga de satisfacer los requisitos básicos. Este modelo se centra en la entrega de un producto funcional con cada incremento. Cada incremento, excepto el último, es una versión incompleta del producto final, pero provee funcionalidades que pueden ser evaluadas.

3.1 Características:

- La evaluación determina del incremento anterior determina el siguiente.
- Los requisitos de mayor prioridad se incluyen en los primeros incrementos.
- Se incrementa la cantidad de requisitos satisfechos con cada incremento, en forma de conjuntos de requisitos.
- Este modelo presupone que se conoce el conjunto completo de requisitos del producto final al comienzo para poder planificar los incrementos con éxito.
- Con cierta frecuencia se entregan versiones funcionales del producto final.

3.2 Ventajas:

- Desde etapas tempranas del ciclo de vida de este modelo se genera software funcional.
- La flexibilidad del modelo permite reducir el costo en el cambio de alcance y requisitos.
- Es más fácil probar y depurar en cada incremento que en el producto completo.

- De haber errores graves en el producto final, sólo es necesario descartar el último incremento.

3.3 Inconvenientes:

- Se requiere de experiencia para realizar una planificación exitosa de los incrementos.

- Los incrementos no debe ser interrumpidos o superpuestos unos con otros.

- Si los requisitos cambian se debe re-planificar la secuencia de incrementos, debido a que los mismos se planifican sólo al inicio.

En la Fig. 17 se ilustra el ciclo de vida del *modelo incremental*.

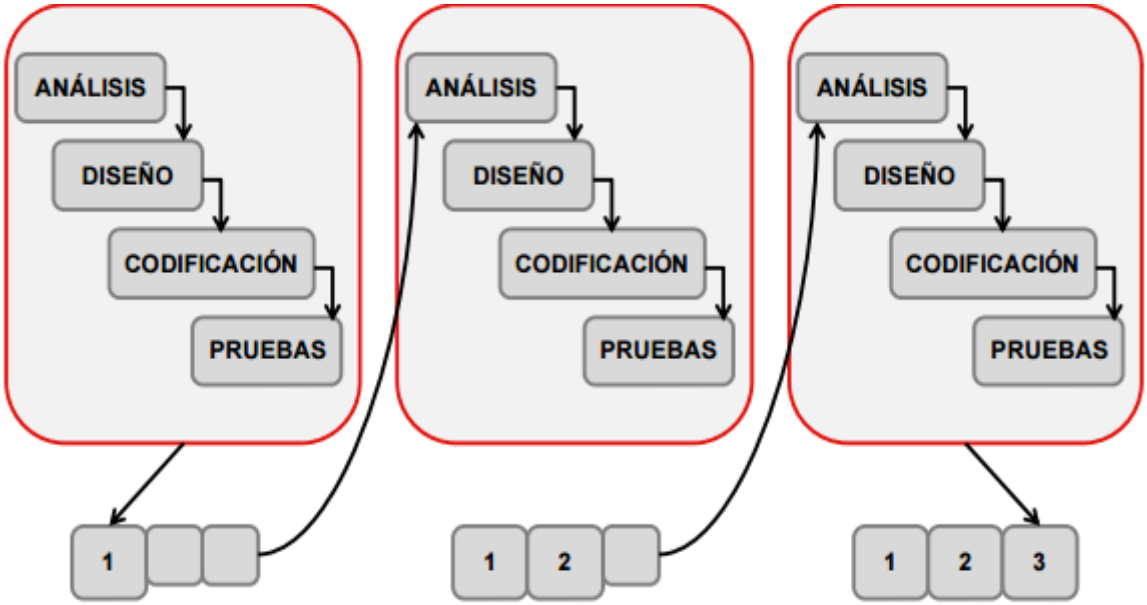


Figura 17. Ciclo de vida del *modelo incremental*.

Capítulo 4

Marco Aplicativo

Este capítulo describe las herramientas utilizadas y la solución planteada para el desarrollo de la aplicación de acuerdo a la metodología. Esta será explicada de acuerdo a cada etapa de la metodología utilizada. En total se emplearon tres (3) fases de desarrollo.

4.1 Fase de Procesamiento del Lenguaje

En esta fase se invirtieron cuatro incrementos para desarrollar el procesador del lenguaje utilizado, los cuales son: Diseño de la Gramática, Analizador Léxico, Analizador Sintáctico y Analizador Semántico. En las próximas secciones se describe cada uno de forma detallada.

1. **Diseño de la Gramática:** Se explica qué gramática se utilizó y por qué. Se ilustra la misma en notación BNF y también se explican algunos detalles de importancia.

2. **Analizador Léxico:** Se explica el proceso de análisis léxico. Se menciona la herramienta utilizada para ello y se detalla su uso sobre este trabajo con imágenes.

3. **Analizador Sintáctico:** Se explica la entrada que recibe este proceso y la salida generada. También se detalla el uso de la herramienta encargada.

4. **Analizador Semántico:** Se explica su proceso paso a paso, se abordan aspectos de importancia y se explica lo que no es válido para este análisis con respecto al lenguaje escogido.

4.1.1 Primer Incremento: Diseño de Gramática

Para el diseño de la gramática se consideró utilizar una gramática de un lenguaje conocido, pues así sería más intuitivo para el usuario. Además, el lenguaje debía ser interpretado para agilizar su ejecución a medida que el usuario agrega instrucciones al código. Así pues, fue seleccionado el lenguaje Lua. Un lenguaje de scripting, interpretado, con una gramática sencilla.

A la gramática original se le agregaría una extensión para el despliegue de gráficos. Además, no sería necesario emplear toda la gramática, con un subconjunto lo suficientemente amplio de la gramática como para hacer uso de la extensión propuesta y desplegar gráficos es suficiente.

La extensión para el despliegue de gráficos provee mecanismos para el despliegue de objetos 3D y 2D predefinidos, transformaciones (traslación, rotación escalado), carga de modelos en formato OBJ, modelo de iluminación Blinn-Phong, modelos de sombreado Phong, Gouraud y Flat, componentes ambiental, difuso y especular. Estos serán abordados más en detalle en la fase de despliegue de gráficos.

La gramática utilizada en notación BNF se muestra en la Fig. 18.

```
start ::= chunk
chunk ::= {stat}[last_stat]
last_stat ::= 'return' [exp_list][';']
stat ::= 'local' name_list ['=' exp_list] |';' |var_list '=' exp_list |func_call |
'while' exp 'do' chunk {'elseif' exp 'then' chunk} ['else' chunk] |
'for' name '=' exp ',' exp [',' exp] 'do' chunk 'end' |'repeat' chunk 'until' exp |
'function' func_name func_body | 'break' | line_comment | long_comment
func_name ::= name
var_list ::= var {',' var}
var ::= name {('.' name | '[' exp ']')}
name_list ::= name {',' name}
exp_list ::= exp {',' exp}
exp ::= exp bin_op exp | un_op exp | '(' exp ')' |'nil' |'true' |'false' |
string |char_string |numeral | var | func_call | hash_const
bin_op ::= 'or' | 'and' | '==' | '~=' | '>=' | '<=' | '>' | '<' | '|' |
'~' | '&' | '>>' | '<<' | '..' | '-' | '+' | '%' | '//' | '/' | '*' | '^'
un_op ::= '~' | '-' | '#' | 'not'
func_call ::= var args
args ::= '(' [exp_list] ')'
hash_const ::= '{' [field_list] '}'
field_list ::= field {field_sep field}
field_sep ::= ';' | ','
field ::= '[' exp ']' '=' exp | name '=' exp | exp
func_body ::= '(' [name_list] ')' chunk 'end'
```

Figura 18. Gramática utilizada como parte del lenguaje interpretado Lua.

La extensión propuesta se ilustra en la Fig. 19.

```
1  DrawCube (String displayMode);
2  DrawCone (String displayMode);
3  DrawSphere (String displayMode);
4  DrawCylinder (String displayMode);
5  DrawGrid (String displayMode);
6  DrawObject (String displayMode);
7  TranslateObject (Array vector);
8  RotateObject (Number angle, Array vector);
9  ScaleObject (Array vector);
10 DrawPointLight (Array pos);
11 DrawDirectionalLight (Array pos, Array dir);
12 DrawSpotLight (Array pos, Array dir, Number cutoff, Number exponent);
13 ChangeLighting (String model);
14 AmbientalComponent (Array vector);
15 DiffuseComponent (Array vector);
16 SpecularComponent (Array vector);
```

Figura 19. Extensión propuesta en el lenguaje interpretado Lua para el despliegue gráfico.

Esta extensión representa una inclusión de los aspectos básicos del despliegue de gráficos, de manera que el lenguaje soporte o proporcione herramientas para el mismo. Estas herramientas se proporcionan en forma de funciones con argumentos o parámetros, los cuales sirven para personalizar el resultado de la ejecución de la función. Esta extensión no interfiere con las demás características del lenguaje, debido a que sólo se incluyen funciones.

4.1.2 Segundo Incremento: Analizador Léxico

El análisis léxico se llevó a cabo utilizando la herramienta *Jison*. Un generador de procesadores de lenguajes, donde se definen tanto el analizador léxico como el sintáctico. La herramienta recibe una gramática escrita en una sintaxis predefinida y genera un código en lenguaje JavaScript equivalente al *parser* que utiliza la gramática diseñada.

En la Fig. 20 se muestra un ejemplo de gramática sencilla implementada utilizando Jison. Como resultado se obtiene un código en JavaScript equivalente a un intérprete, el cual evalúa y ejecuta las operaciones de suma, resta, multiplicación, división y potencia. El uso de este intérprete se muestra en la Fig. 21. En el campo de texto se ingresa la operación, el botón **equals** invoca una función en JavaScript que a su vez invoca al intérprete y el resultado generado se muestra a la derecha del botón.

```
%lex
%%
\s+                   /* skip whitespace */
[0-9]+("."[0-9]+)?\b  return 'NUMBER';
"*"                   return '*';
"/"                   return '/';
"-"                   return '-';
"+"                   return '+';
"^"                   return '^';
"("                   return '(';
")"                   return ')';
"PI"                  return 'PI';
"E"                   return 'E';
<<EOF>>               return 'EOF';
/lex

%left '+' '-'
%left '*' '/'
%left '^'
%left UMINUS

%start expressions

%%

expressions: e EOF {print($1); return $1;}
           ;

e    : e '+' e {$$ = $1+$3;}
     | e '-' e {$$ = $1-$3;}
     | e '*' e {$$ = $1*$3;}
     | e '/' e {$$ = $1/$3;}
     | e '^' e {$$ = Math.pow($1, $3);}
     | '-' e %prec UMINUS {$$ = -$2;}
     | '(' e ')' {$$ = $2;}
     | NUMBER {$$ = Number(yytext);}
     | E {$$ = Math.E;}
     | PI {$$ = Math.PI;}
     ;
```

Figura 20. Gramática ejemplo usando Jison.

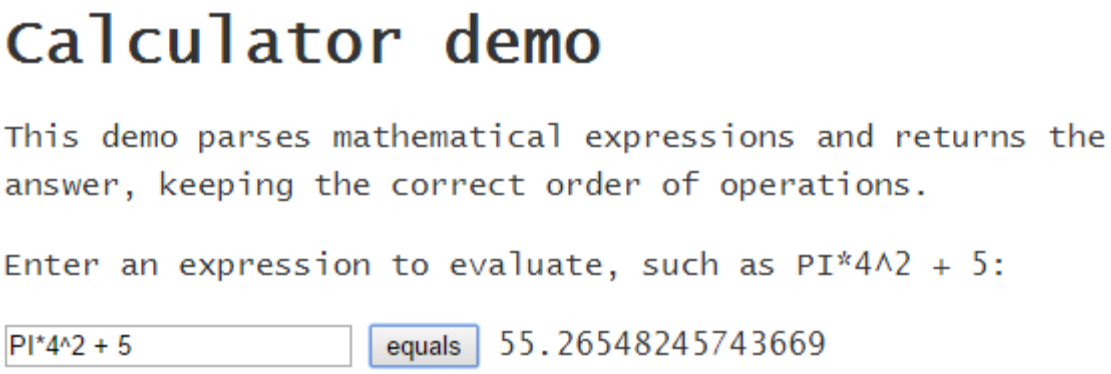


Figura 21. Uso del intérprete generado por Jison.

El analizador léxico identifica los *lexemas* o palabras claves que se utilizan en el lenguaje. Para ello, se hace uso de expresiones regulares y un orden establecido que no involucre un conflicto entre *lexemas*. Esto debido a que el analizador intenta identificar el *lexema* haciendo uso de las expresiones regulares secuencialmente desde la primera hasta la última. De no conseguir identificar el *lexema* se genera un error afirmando que el código no es válido para el lenguaje definido.

4.1.3 Tercer Incremento: Analizador Sintáctico

Como se dijo anteriormente, el analizador sintáctico también se llevó a cabo utilizando *Jison*. A diferencia del analizador léxico, este se compone por reglas gramaticales, las cuales definen la sintaxis que debe cumplir todo código evaluado por el *parser*. Igualmente, utiliza los lexemas identificados por el analizador léxico en sus reglas para identificar cuál regla gramatical equivale a la instrucción evaluada.

El analizador a medida que identifica instrucciones válidas genera nodos de una estructura llamada *AST* o *Árbol de sintaxis abstracta*. Se utilizó un analizador sintáctico ascendente para la generación de la estructura en cuestión. Esta estructura será utilizada en el siguiente analizador, pero este se encarga de llenarla de datos.

Jison provee una sintaxis parecida a *BNF* para escribir la gramática que utilizará el *parser*, pero a diferencia de *BNF* en esta sintaxis se requiere establecer un orden claro de evaluación de las reglas.

Para llenar las estructuras que empleará el analizador sintáctico, *Jison* provee bloques de código que pueden colocarse al final de la regla gramatical, de manera que sean ejecutados en el lenguaje destino una vez que se identifique la regla gramatical. La razón de utilizar un *AST* y no otra técnica, es que *Jison* no permite que los bloques de código sean colocados antes del final de la regla gramatical, lo que dificulta drásticamente otras técnicas que no requiere del uso de una estructura. Así, es mucho más fácil almacenar la información de cada instrucción por completa en forma de nodos del árbol que no almacenarla y pasar al siguiente analizador partes de cada instrucción hasta armarla por completo.

Una vez que el *parser* haya culminado, se obtiene el *AST* en el programa en lenguaje JavaScript y se da paso al siguiente análisis. En la Fig. 22 se ve reflejado un ejemplo de *AST*.

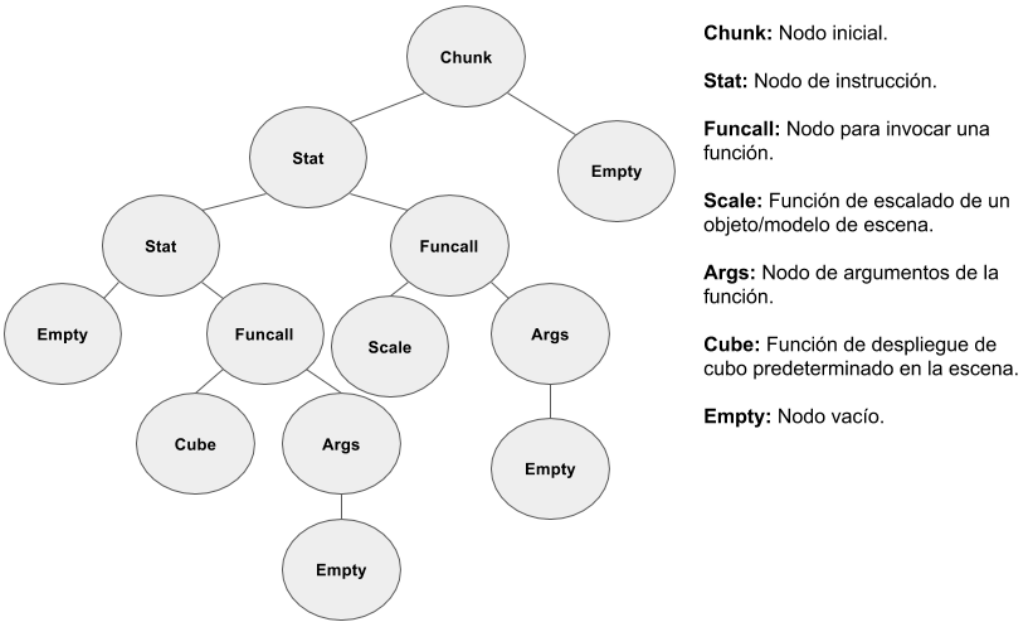


Figura 22. Ejemplo de un AST dentro de una gramática.

4.1.4 Cuarto Incremento: Analizador Semántico

Este análisis conlleva más peso o mayor inversión de recursos computacionales. El analizador se encarga de recorrer el *AST* generado por el analizador anterior, de llenar las entradas de la *TS*, de administrar los ambientes de ejecución o el alcance de las variables, de invocar al desplegador gráfico (*DG*) y de ejecutar las instrucciones en lenguaje Lua para obtener los parámetros que se le pasarán al *DG*.

Como se dijo anteriormente, el *parser* es un código en lenguaje JavaScript, el cual es invocado pasándole como parámetro el código del usuario por la aplicación. Una vez ejecutados los análisis del *parser*, la aplicación invoca a una función *eval*, la cual se encarga de recorrer recursivamente el *AST* en profundidad, es decir, los últimos nodos u hojas son los evaluados primero.

Cada instrucción dentro del lenguaje Lua simboliza un tipo de nodo que puede encontrarse en el *AST*, y por tanto la estructura contiene instancias de los mismos. Lo que diferencia a dos nodos del mismo tipo es la información que contenga para ser ejecutada. En la Fig. 22 se ilustran nodos repetidos, pero estos contienen diferente información e invocarán diferentes nodos como consecuencia.

Para los ambientes de ejecución se utilizó una *pila de ejecución*, la cual en cada nodo contiene un ambiente de ejecución. El primer ambiente es el ambiente global, el cual contiene todas las variables y funciones visibles en todo el programa. Los demás ambientes pueden ver únicamente los ambientes que tengan por debajo de ellos en la pila; de esta manera todos los ambientes pueden ver el ambiente global y todos los ambientes que se hayan invocado para llegar al ambiente actual. Cada

invocación a una función genera un nuevo ambiente de ejecución, que será insertado al principio de la pila. También existen los *ambientes temporales*, los cuales son invocados o creados al momento de ejecutar un ciclo o un bloque *do*. Esto permite que las variables que sean utilizadas dentro de estos ambientes no sobrescriban los demás ambientes y sean destruidas o descartadas en cuanto termine la ejecución del ambiente.

Todo ambiente de ejecución contiene una estructura llamada *TS*, la cual contiene entradas equivalentes a los nombres de variables y funciones que viven dentro del ambiente. Las entradas de la tabla son *string* que se utilizan como identificadores para la posición de la tabla que contiene la información relacionada a la entrada. Toda entrada es única para cada tabla, y por lo tanto también para cada ambiente. En el caso de las funciones, se almacena el código completo de la misma en la tabla, pero el código en forma de nodos del *AST*, es decir, contiene los nodos que serán evaluados al llamar a la función. Esto se encuentra de esta manera porque el lenguaje Lua así lo establece. Lo que realmente ocurre, es que los parámetros de las funciones se desconocen hasta que se invoca a la misma, por lo que evaluar la misma semánticamente no puede determinar con total seguridad si la semántica de esta es válida.

Al invocar al *DG* se realizan cuando se evalúa un nodo relacionado a una instrucción de la extensión para el despliegue de gráficos. Debido a que la misma se trata de un conjunto de funciones, su funcionamiento se comporta de manera igual a las funciones del lenguaje Lua, su única diferencia es que su ejecución ya está predefinida y sólo debe invocarse (no implementarse). Algunas de estas funciones de la extensión proveen parámetros predeterminados, de manera que el usuario no necesite colocar todos los parámetros que requiera la función.

En su mayoría las especificaciones del lenguaje Lua se cumplen. A continuación se detallan aquellas que no se cumplen o no son válidas dentro de la semántica del subconjunto establecido. En el manual de referencia de Lua se encuentran las especificaciones completas de la semántica válida.

- **Valores y Tipos**: Lua maneja 8 tipos básicos: *nil*, *boolean*, *number*, *string*, *function*, *userdata*, *thread* y *table*. En el lenguaje propuesto no es válido el uso de los tipos *userdata* y *thread*, o el uso de funciones en las tablas. También es posible en Lua hacer llamadas a funciones en lenguaje C, pero en el propuesto no es válido.

- **Coerción:** Lua puede convertir automáticamente entre valores *string* y valores numéricos. El lenguaje propuesto también cumple con esto, pero no existe la función **format** que en Lua permite un control completo en la conversión de números y el tipo string.

- **Variables:** Lua permite obtener la *TS* del ambiente actual, y también permite reemplazarla. El lenguaje propuesto no permite esto.

- **Chunks:** La unidad de ejecución en Lua se denomina *chunk*, el cual es simplemente un conjunto de sentencias que se ejecutan secuencialmente. A diferencia de Lua, en el lenguaje propuesto no permite almacenar los *chunks* en un fichero o dentro de un programa en forma de string, ni son pre-compilados. Toda sentencia es interpretada de la misma forma.

- **Asignación:** El mecanismo de asignación a las variables globales y a los campos de tablas no puede ser modificado, a diferencia de Lua, en el cual se puede mediante el uso de meta-tablas.

- **Estructuras de control:** En Lua existe una instrucción llamada ***goto***, la cual re-direcciona el programa a cierta instrucción. La misma no es permitida en el lenguaje propuesto.

- **La sentencia for:** A diferencia de Lua, el lenguaje propuesto no permite el uso de la forma genérica del *for*, lo que significa que sólo es permitida la forma numérica del *for* que repite un bloque mientras una variable de control sigue una progresión aritmética. En la Fig. 23 se muestra un ejemplo de la forma genérica de un *for*.

```
1  revDays = {}
2  for i,v in ipairs(days) do
3      revDays[v] = i
4  end
```

Figura 23. Forma genérica del ciclo *for*.

- **Expresiones:** El lenguaje propuesto no permite el uso de la palabra clave **…** (3 puntos continuos) en ninguna circunstancia.

- **Operadores relacionales:** A diferencia de Lua, el lenguaje propuesto no intenta comparar entre valores de diferentes tipos. Ambos deben ser del mismo tipo.

- **El operador longitud:** En Lua la longitud de un *string* es el número de bytes que representa en memoria. Además, en Lua cada carácter representa un byte en memoria. Debido a que el intérprete está hecho en JavaScript y este utiliza dos bytes para almacenar cada carácter en memoria, se decidió simular el cálculo de bytes que se hace en un intérprete común de Lua y

arrojar como resultado la mitad de bytes que se utilizan para almacenar el tipo string en memoria. Así el usuario no percibe ninguna diferencia, a pesar de que no se cumpla con la especificación.

- **Llamadas a funciones:** Lua permite utilizar una expresión que como resultado retorne la función a ser invocada. En el lenguaje propuesto esto no es válido, de igual manera que utilizar variables para almacenar funciones o retornar funciones en sí. Lua también permite definir los parámetros de la llamada utilizando tablas o el tipo string, lo cual en el lenguaje propuesto no es permitido.

- **Definición de funciones:** A diferencia de Lua, el lenguaje propuesto no permite definir funciones locales, es decir, con la palabra clave ***local***. Tampoco es permitido en el lenguaje propuesto el uso de ***self***, el cual en Lua contiene la función en sí y se usa como parámetro de sí misma.

- **Convenciones léxicas:** Lua permite sentencias de escape en el tipo string, como lo son: ***'\n'*** y ***'\t'***. El lenguaje propuesto permite estos caracteres, pero no los interpreta sino que se toma como cualquier otro carácter. Además, el lenguaje propuesto utiliza la codificación UTF-8 con Unicode, la cual es permitida en Lua pero no es su codificación por defecto.

- **Manejo de errores:** El intérprete se encarga del manejo de errores, y lo hace diferente a como se hace en Lua. El intérprete propuesto ejecuta todo lo que puede, es decir, los errores semánticos no detienen la ejecución del programa e intenta terminar la ejecución completa. Sin embargo, los errores sintácticos son críticos para la ejecución del programa, pues el análisis sintáctico recibe como entrada el código completo y al terminar de manera exitosa genera el *AST*, de otro modo no genera la estructura necesaria para el análisis semántico y no es posible ejecutar el programa.

4.2 Fase de Despliegue de Gráficos

Esta fase requirió dos incrementos, los cuales están muy relacionados entre sí, sin embargo con el fin de ofrecer una aplicación modular y más entendible, se muestran separados.

1. **Despliegue Gráfico:** Se explica paso a paso la inicialización de la escena y los procesos que conlleva. Se abordan aspectos de importancia al invocar procesos durante la ejecución del despliegue gráfico.

2. **Modelos de Escena:** Se detallan aspectos de los modelos/objetos de escena y además se explica cómo estos sirven de interfaz entre el *DG* y el intérprete.

4.2.1 Primer Incremento: Despliegue Gráfico

En este incremento se desarrolló un *DG* que utiliza el API WebGL para el despliegue de gráficos. El mismo se encarga de inicializar la escena, compilar los **shaders**, inicializar la cámara, desplegar los objetos/modelos y manejar transformaciones. Todos estos procesos serán abordados detalladamente en breve.

En primer lugar, se escogió utilizar el API WebGL sin el uso de frameworks ni bibliotecas externas para ofrecer *Plantillas* de código sin convenciones, es decir, *Plantillas* base para que el usuario tenga el control de todos los procesos que se llevan a cabo en su despliegue.

El *DG* realiza los siguientes procesos:

1.- Inicializar la Escena:

Para la inicialización de la escena el *DG* primero se encarga de instanciar un **canvas** de **html** con las medidas precisas que contiene el mismo. De esta manera se crea la ventana en la cual el usuario podrá visualizar el despliegue que desarrolle. Cabe destacar que a medida que la ventana del navegador cambie de tamaño, la ventana del *DG* o el **canvas** también lo hará para ajustarse a las nuevas medidas.

2.- Compilar los shaders:

Los **shaders** son almacenados en el código **html** para no perjudicar el rendimiento de la aplicación con la carga de más archivos. Estos son compilados y enlazados para crear un **program shader**, el cual será utilizado para el despliegue de cada objeto/modelo de escena. Posterior a su compilación, los **shaders** necesitan enlazar las variables que se les pasarán como argumentos para lograr el efecto deseado de despliegue. Una vez hecho el enlace de estas variables ya se encuentra disponible para su uso durante la ejecución.

3.- Inicializar la cámara:

Posteriormente, se inicializa la cámara, es decir, se le asigna una posición, una dirección hacia donde ver y un vector que definirá el rango de visión de la misma. Por último se le da la visión perspectiva a la escena con las medidas de la ventana y las medidas que definirán el tamaño de la escena a desplegar.

La cámara utilizada es una cámara libre, permitiendo al usuario navegar por la escena libremente e incluso atravesar objetos/modelos.

4.- Desplegar objetos/modelos de escena:

Una vez inicializada la escena, se procede a desplegar objetos/modelos y a aplicarles transformaciones, o incluso cambiar el **shader** en uso. Para el despliegue de los objetos/modelos el *DG* define una estructura que contiene en cada posición un objeto/modelo a desplegar (este objeto/modelo puede ser una luz), la misma es recorrida cada vez que se invoca al **display** y se llama a la función de despliegue de cada objeto/modelo con sus parámetros correspondientes (**shaders**, propiedades de material y matrices). Existe el caso de que un objeto modelo contenga una transformación que se deba hacer constantemente, la cual es almacenada y aplicada cada vez que la función **display** es invocada.

5.- Manejar transformaciones:

El manejo de transformaciones es simple. Toda transformación se aplica únicamente al último objeto/modelo desplegado en la escena, lo que equivale a la última llamada de despliegue de objeto/modelo en el código desarrollado por el usuario. De esta manera no es necesario llevar un control de qué objeto/modelo es el indicado, tampoco es necesario el almacenamiento de los objetos/modelos en objetos del lenguaje Lua por el hecho de que las transformaciones sólo necesitan ser colocadas después de la llamada de despliegue de un objeto/modelo.

6.- Otros procesos y aspectos de importancia:

La iluminación está basada en los modelos Blinn-Phong, Gouraud y Flat. Estos modelos utilizan los componentes ambiental, difuso y especular. Estos componentes pueden ser modificados en valor con ciertas funciones que se le ofrecen al usuario; las mismas ofrecen el manejo de colores en los rangos [0,1], [0,255] y también ofrecen una serie de colores predeterminados con nombres propios (http://www.w3schools.com/colors/colors_names.asp). El usuario también puede agregar o eliminar luces, y para esto los shaders son compilados de nuevo con el nuevo número de luces que serán utilizadas en el despliegue. Cabe destacar que las luces también son objetos y que son parte de la escena, por lo que pueden ser modificadas de igual manera que a cualquier otro objeto/modelo de la escena, ya sea mediante transformaciones, cambio de componentes, o parámetros específicos del tipo de luz. La aplicación soporta tres tipos de luces: puntual, direccional y concentrada.

4.2.2 Segundo Incremento: Modelos de Escena

Para este incremento se diseñaron diferentes objetos que contienen todos los parámetros necesarios para el despliegue de un objeto/modelo de escena. Estos contienen los vértices, vectores normales por caras, vectores normales por vértices, la matriz de modelo, la matriz normal, funciones para la inicialización de los búferes a pasar a los **shaders**, funciones de transformación y la función de despliegue que es invocada en el *DG*.

Estos objetos sirven de interfaz para comunicar el intérprete con el *DG*, ya que son instanciados por el intérprete y sus instancias son invocadas por el *DG*. Así, el intérprete se encarga de reunir la información que necesita cada objeto para su instanciación y el *DG* de invocar sus procesos, ya sean de despliegue o transformación, pasando como argumento el **program shader** actual y otros que pueden cambiar durante el despliegue.

Existen cinco modelos 3D predefinidos, los cuales se muestran en la Fig. 24, que provee la aplicación: cubo, cono, esfera, cilindro y grid. Estos modelos ya tienen todos los datos necesarios para ser desplegados inmediatamente. Por otro lado, la aplicación permite la carga de modelos en formato *obj*. Estos modelos se cargan con un módulo diseñado **ad-hoc**. Las normales por vértice son aproximadas mediante el promedio de las normales por cara.

Para pasar la información a desplegar a los **shaders** se utilizan búferes. En este trabajo se utilizó uno llamado **vertex buffer object** (*vbo*), en el cual se almacenan secuencialmente en posiciones del búfer los datos que se quieren utilizar en el **program shader**. Es posible ordenar los datos para procesarlos de manera que se adecue mejor al problema, pero en este caso no fue necesario, por lo que el orden en el que se reciben los datos del cargador de modelos/objetos es el mismo en el que se encontrarán almacenados en el búfer. Debido a que existen diferentes datos (vértices, normales, etc) se utilizaron varios *vbo*, específicamente uno para cada dato diferente que se pasará al **program shader**.

Los modelos poseen una propiedad que puede ser modificada por el usuario, y es la forma de despliegue, es decir, la figura usando triángulos, usando puntos y usando líneas. De esta manera los usuarios pueden visualizar la malla del objeto/modelo.

Además, todo objeto/modelo posee un material el cual da información a los **shaders** sobre el material que está hecho el objeto/modelo.

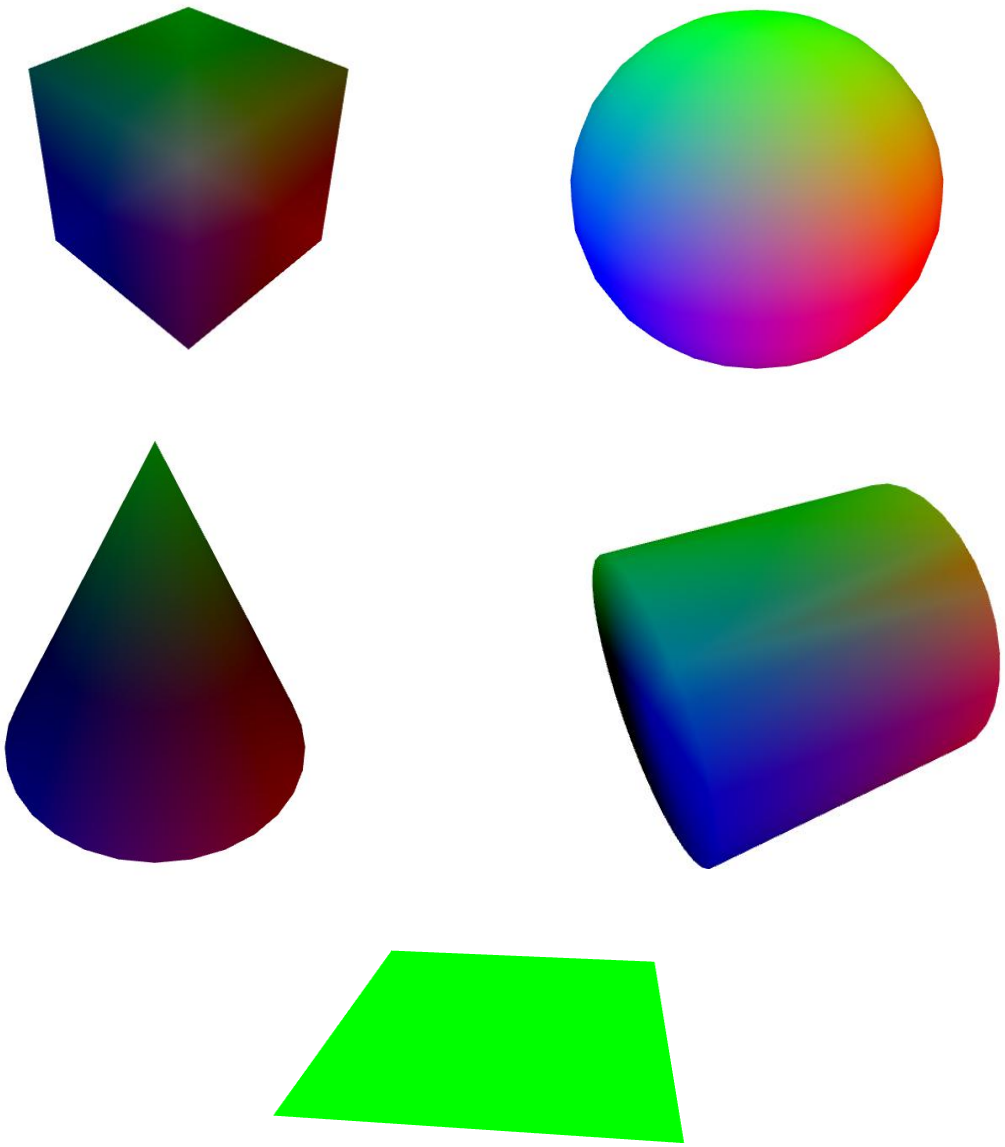


Figura 24. Modelos 3D predefinidos que provee la aplicación, cubo, esfera, cono, cilindro y plano (de arriba abajo y de izquierda a derecha).

4.3 Fase de Interfaz de Usuario

Para esta fase fueron necesarios dos incrementos referidos al control y la facilidad de uso del usuario con la aplicación.

1. **Interfaz Gráfica Web:** Se explica la estructura de la interfaz que permite visualizar los despliegues que se desarrollen, además de algunos accesos directos que facilitan el manejo de la aplicación.

2. **Generación de Plantillas:** Se explica cómo son las *Plantillas* de código, qué estructura tienen y cómo se generan.

4.3.1 Primer Incremento: Interfaz Gráfica Web

La interfaz web consiste en:

1. Un editor de código.
2. Una ventana para visualizar los gráficos.
3. Una consola de sólo lectura para los mensajes de la aplicación.
4. Una barra de menú con diferentes opciones.

En la Fig. 25 se muestran cada uno de los componentes de la interfaz.

Figura 25. Interfaz Gráfica Web.

La interfaz está basada en HTML5, CSS3 y JavaScript. Del mismo modo, la interfaz se basa en la técnica **responsive**, la cual ajusta los componentes de la interfaz a medida que la ventana del navegador cambia de tamaño, lo que la hace ajustarse a la pantalla de casi cualquier tipo de dispositivo.

Se diseñaron varias convenciones o accesos directos para facilitar el uso de la interfaz. Entre ellas están:

- Al presionar la tecla *Enter*, la aplicación interpreta el código que esté escrito en el editor de código.

- Al presionar el *scroll* del ratón se bloquea el editor de código y se activa la cámara libre sobre la escena desplegada.

Para escribir código utilizando la extensión propuesta se proporciona un editor de código. Este enumera las líneas de código, enmarca en color rojo la línea que contenga un error sintáctico, permite el uso de los accesos directos como copiado y pegado.

En la Tabla 1 se mencionan las instrucciones permitidas que generan una salida gráfica.

Nombre de la función	Argumentos	Descripción
DrawCube	String displayMode	Despliega un cubo en la posición (0,0,0) de la escena y si el argumento **displayMode** es válido lo utiliza para el despliegue del cubo. **displayMode** puede tomar los valores: “POINTS”, “LINES”, “LINE_STRIP”, “LINE_LOOP”, “TRIANGLES”, “TRIANGLE_STRIP” y “TRIANGLE_FAN”.
DrawSphere	String displayMode	Despliega una esfera con las mismas especificaciones de la función **DrawCube**.
DrawCone	String displayMode	Despliega un cono con las mismas especificaciones de la función **DrawCube**.
DrawCylinder	String displayMode	Despliega un cilindro con las mismas especificaciones de la función **DrawCube**.
DrawGrid	String displayMode	Despliega un **grid** con las mismas especificaciones de la función **DrawCube**.
DrawObject	String object, String displayMode	Despliega un objeto/modelo cargado de un archivo OBJ. El argumento **object** debe ser el nombre del archivo con la extensión y debe usar la sintaxis OBJ para poder desplegar correctamente el objeto/modelo en la escena. También utiliza las mismas especificaciones de la función **DrawCube**.
TranslateObject	Array vector	Traslada un objeto/modelo de escena utilizando como dirección el argumento **vector**. Este argumento sólo debe tener tres posiciones y estas deben contener números solamente.
RotateObject	Number angle ,Array vector	Rota un objeto/modelo de escena utilizando como dirección el argumento **vector** y como ángulo **angle**. Utiliza las mismas especificaciones que la función **TranslateObject**.
ScaleObject.	Array vector	Escala un objeto/modelo de escena utilizando el argumento **vector**. Utiliza las mismas especificaciones que la función **TranslateObject**
DrawPointLight	Array pos	Despliega una luz puntual en la escena y se utiliza por defecto el modelo Blinn-Phong. El argumento **pos** indica en qué posición de la escena será desplegada la luz y utiliza las mismas especificaciones que la función **TranslateObject**.
DrawDirectionalLight	Array pos, Array dir	Despliega una luz direccional en la escena y se utiliza por defecto el modelo Blinn-Phong. El argumento **pos** indica en qué posición de la escena será desplegada la luz y **dir** indica en qué dirección iluminará. Utiliza las mismas especificaciones que la función **TranslateObject**.
DrawSpotLight	Array pos, Array dir, Number cutoff,	Despliega una luz concentrada en la escena y se utiliza por defecto el modelo Blinn-Phong. El argumento **pos** indica en

	Number exponent	qué posición de la escena será desplegada la luz, **dir** indica en qué dirección iluminará, **cutoff** indica el ángulo de abertura y **exponent** indica la intensidad de la luz. Utiliza las mismas especificaciones que la función **TranslateObject**.
ChangeLighting	String model	Cambia el **program shader** actual por el que se pase como argumento. Esto sólo sucederá si el argumento es válido. Los valores válidos para el argumento **model** son: “Flat”, “Gouraud” y “Phong”.
AmbientalComponent	Array vector	Cambia el componente ambiental de un objeto/modelo por el que se pasa como argumento. Es posible usar números en el rango [0,1] y [0, 255] para el color. Se utilizan tres números para el color y se pueden utilizar los colores predeterminados mencionados en la fase de Despliegue de Gráficos.
DiffuseComponent	Array vector	Cambia el componente difuso de un objeto/modelo por el que se pasa como argumento. Utiliza las mismas especificaciones que la función **AmbientalComponent**.
SpecularComponent	Array vector	Cambia el componente especular de un objeto/modelo por el que se pasa como argumento. Utiliza las mismas especificaciones que la función **AmbientalComponent**.

Tabla 1. Instrucciones permitidas para generar una salida gráfica

4.3.2 Segundo Incremento: Generación de Plantillas

Se diseñó una estructura de código llamada *Plantilla* que contiene el código que genera el despliegue gráfico desarrollado por el usuario sin componentes de interfaz ni módulos de la aplicación que no se relacionen directamente con el despliegue de gráficos. Así, el usuario puede obtener un código para su uso sin ningún obstáculo como módulos innecesarios, código ofuscado u “oscurecido”. El usuario es capaz de mejorar o modificar a su voluntad el despliegue desarrollado rápidamente utilizando la solución propuesta.

El usuario tendrá disponible estas *Plantillas* en forma de un archivo comprimido, el cual contiene cada módulo de despliegue de gráficos por separado y un archivo con la información de cada objeto/modelo desplegado en la escena. Cada uno de estos módulos es una *Plantilla*, es decir, un código en lenguaje JavaScript utilizando el API WebGL organizado de manera que sea entendible para el usuario y con nombres de variables y funciones mnemónicos.

Un código en el lenguaje propuesto genera varias *Plantillas*, una por cada módulo de la aplicación que se utilice. Algunas *Plantillas* siempre serán generadas sin importar los módulos que se utilicen, como la *Plantilla* del *DG*. Así, no se generarán archivos innecesarios. Para que el usuario obtenga las *Plantillas* necesarias, debe utilizar la interfaz gráfica y descargar los archivos.

Es de importancia entender que no se ejecuta una traducción del código escrito en lenguaje Lua a lenguaje JavaScript, sino del despliegue desarrollado en la aplicación a un despliegue hecho en JavaScript con el API WebGL utilizando sólo lo necesario para su ejecución.

Capítulo 5
Resultados y Discusión

Para demostrar la efectividad de la solución propuesta en este trabajo de investigación se aplicaron las pruebas que se señalan a continuación:

- **Prueba de subconjunto del lenguaje Lua:** Esta demuestra que el subconjunto propuesto permite utilizar las funcionalidades que se plantearon en el alcance.

- **Prueba de esfuerzo programático para generar una salida gráfica:** Demuestra que el esfuerzo del usuario es menor para generar una salida gráfica, tomando como medida las líneas de código.

- **Pruebas de rendimiento:** En esta sección se encuentran tres (3) pruebas que evalúan el rendimiento de la aplicación y de la máquina. A continuación se mencionan dichas pruebas:

    - **Prueba de tiempo de despliegue de objetos/modelos de escena:** Se ven reflejados los tiempos de despliegue y se demuestra que la aplicación soporta diferentes objetos/modelos de escena desplegados al mismo tiempo.

    - **Prueba de tiempo desde el procesamiento del código hasta visualizar una salida gráfica:** Se demuestra que la carga adicional debida al procesamiento del código influye poco en el tiempo de visualización de una salida gráfica.

    - **Prueba de recursos consumidos por la aplicación:** Se ve reflejado el uso del CPU y de memoria de la máquina, y se demuestra que la aplicación puede ser utilizada aún cuando la máquina no posee muchos recursos.

- **Prueba de comparación entre la salida gráfica generada por la aplicación y la generada por las *Plantillas*:** Se demuestra que la salida gráfica generada por la aplicación es equivalente a la generada por la ejecución de las *Plantillas*.

5.1 Prueba de subconjunto del lenguaje Lua

Esta prueba toma en cuenta los aspectos necesarios para implementar las funcionalidades definidas en el alcance de este trabajo, y así justificar por qué el uso de un subconjunto.

Las funcionalidades propuestas en el alcance incluyen:

- Despliegue de figuras 2D y 3D, ya sean predeterminadas o modelos en formato *obj*.
- Modelo de iluminación Blinn-Phong, modelos de sombreado Phong, Gouraud y Flat.
- Cambio de componentes ambiental, difuso y especular tanto en modelos como en luces.
- Uso del formato RGB ya sea con rango de 0 a 1, de 0 a 255 o utilizando colores predefinidos con nombres definidos por Cascade Style Sheet, y soportados por todos los navegadores modernos.
- Transformaciones de traslación, rotación y escalado.

Para implementar estas funcionalidades se utilizaron funciones en lenguaje Lua, con el uso de parámetros para ajustar las características que tenga la funcionalidad. Por esto, es necesaria la inclusión de funciones con parámetros en la sintaxis. Esta inclusión requiere de once (11) reglas gramaticales. La Fig. 26 refleja el porcentaje de reglas incluidas sobre la totalidad del lenguaje, y la Fig. 27 refleja el porcentaje de reglas necesarias sobre la totalidad.

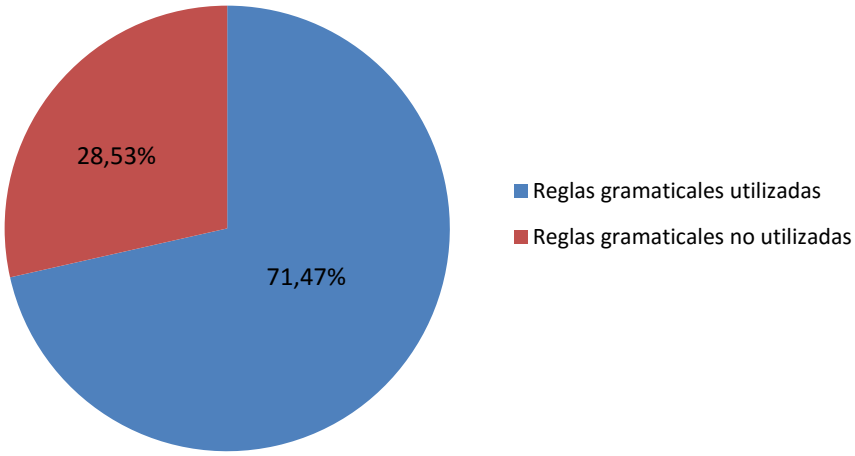


Figura 26. Reglas incluidas de la gramática de Lua.

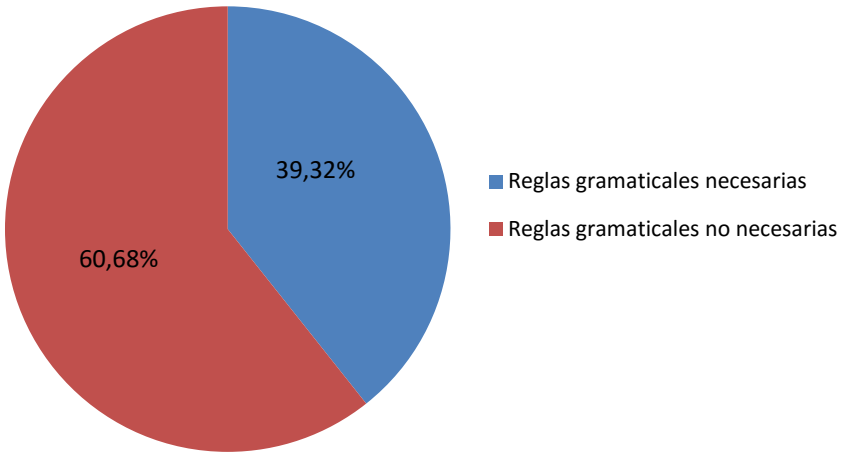


Figura 27. Reglas necesarias de la gramática de Lua.

Como se puede observar, sólo era necesario incluir un poco menos de la mitad de las reglas gramaticales para lograr los objetivos. También se puede observar que se incluyeron más reglas de las necesarias, tomando en cuenta éstas. De esta manera, el usuario posee más herramientas para desarrollar.

Las reglas gramaticales en cuestión se muestran en la Fig. 28.

```
1  func_call::= var args
2  args::= '(' [exp_list] ')'
3  exp_list::= exp {',' exp}
4  exp::= exp bin_op exp | un_op exp | '(' exp ')' |
5  'nil' | 'true' | 'false' | string | char_string |
6  numeral | var | func_call | hash_const
7  bin_op::= 'or' | 'and' | '==' | '~=' | '>=' | '<=' | '>' | '<' | '|' |
8  '~' | '&' | '>>' | '<<' | '..' | '-' | '+' | '%' | '//' | '/' | '*' | '^'
9  un_op::= '~' | '-' | '#' | 'not'
10 numeral::= int | hex_int | float | hex_float
11 hash_const::= '{' [field_list] '}'
12 field_list::= field {(';' | ',') field}
13 field::= '[' exp ']' '=' exp | name '=' exp | exp
14 var::= name {('.' name | '[' exp ']')}
```

Figura 28. Reglas gramaticales necesarias para lograr los objetivos planteados.

5.2 Prueba de esfuerzo programático para generar una salida gráfica

Se diseñaron diferentes casos de prueba, para comparar el esfuerzo que debe realizar el usuario al desarrollar los casos, utilizando la aplicación y sin utilizarla. Los casos evaluados fueron los siguientes:

1. Despliegue de un objeto predeterminado
2. Despliegue de un modelo cargado de un archivo con formato *OBJ*.
3. Despliegue de un objeto/modelo aplicando una transformación.
4. Despliegue de un objeto/modelo y una luz.
5. Despliegue de un objeto/modelo y una luz cambiando el modelo de iluminación.
6. Despliegue de tres (3) objetos/modelos y tres (3) luces.

La Fig. 29 muestra la cantidad de líneas escritas por el usuario para generar la salida gráfica correspondiente a cada caso de prueba.

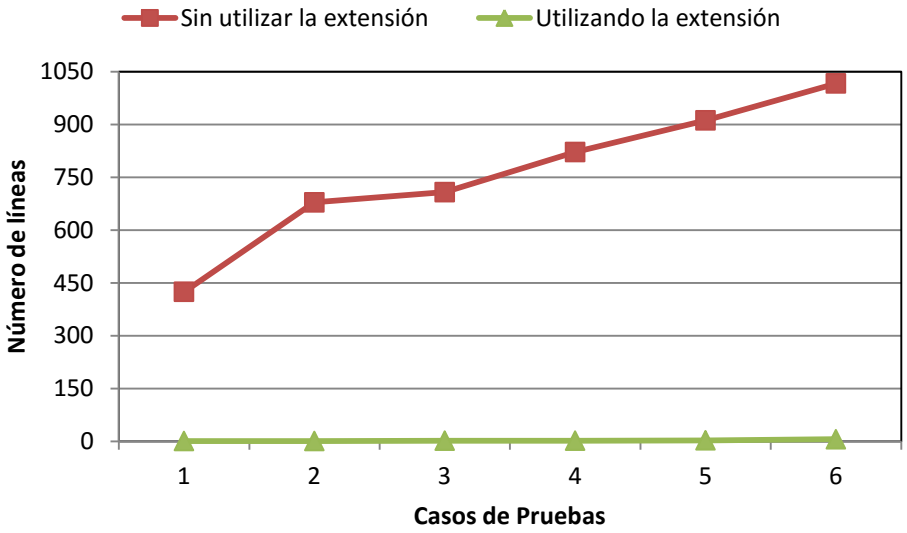


Figura 29. Cantidad de líneas escritas por el usuario.

En la Fig. 29 se puede observar que el usuario requiere una cantidad de líneas mucho menor al utilizar la aplicación, y por tanto requiere de un menor esfuerzo programático.

5.3 Pruebas de rendimiento

En esta sección se utilizaron tres (3) casos de prueba para todas las pruebas realizadas, los cuales se muestran en las Fig. 30, Fig. 31 y Fig. 32 respectivamente. Los casos de prueba son:

- **Despliegue de diez (10) esferas**
- **Despliegue de cien (100) esferas**
- **Despliegue de mil (1000) esferas**

```
local a
for a = 1, 10, 1 do
    DrawSphere()
end
```

Figura 30. Despliegue de diez esferas.

```
local a
for a = 1, 100, 1 do
    DrawSphere()
end
```

Figura 31. Despliegue de cien esferas.

```
local a
for a = 1, 1000, 1 do
    DrawSphere()
end
```

Figura 32. Despliegue de mil esferas.

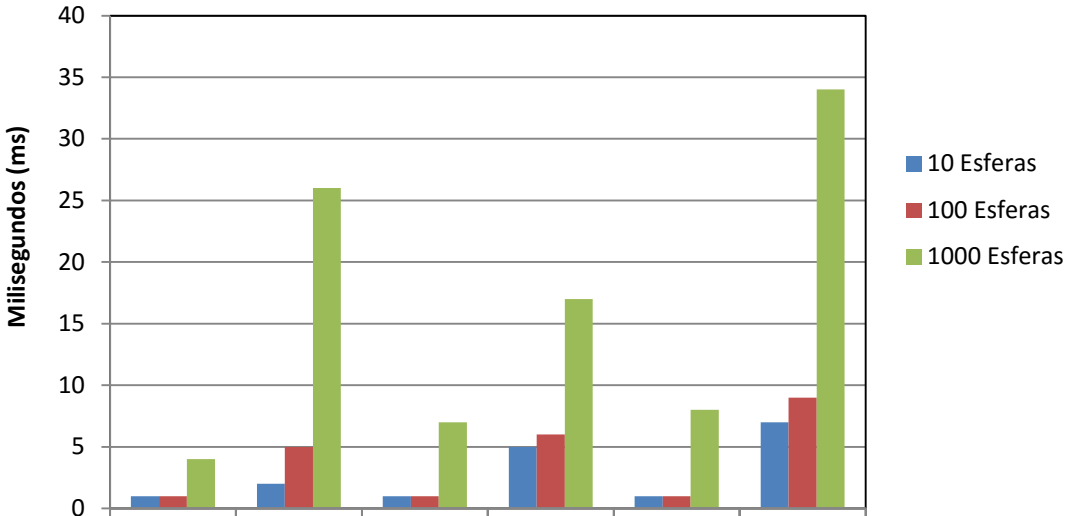
Donde cada esfera equivale a dos mil ochocientos ochenta (2880) triángulos. Estos casos de prueba fueron realizados en tres (3) navegadores para establecer una comparación entre estos, los mismos fueron **Google Chrome**, **Mozilla Firefox** y **Opera**.

Para estas pruebas se utilizó una máquina con los siguientes recursos:

- **Procesador:** Intel Core i5-3210M 2.50GHz x 4
- **Tarjeta Gráfica:** Intel Ivybridge Mobile
- **Memoria:** 15.4 Gb
- **Sistema Operativo:** Debian GNU/Linux 8 (jessie) 64-bit

5.3.1 Prueba de tiempo de despliegue de objetos/modelos de escena

Esta prueba toma en cuenta los tiempos de despliegue de objetos/modelos de escena. En la Fig. 33 se ven reflejados los tiempos cuando se genera la salida gráfica correspondiente a cada caso de prueba.



	Google Chrome		Firefox		Opera	
	Min	Max	Min	Max	Min	Max
10 Esferas	1	2	1	5	1	7
100 Esferas	1	5	1	6	1	9
1000 Esferas	4	26	7	17	8	34

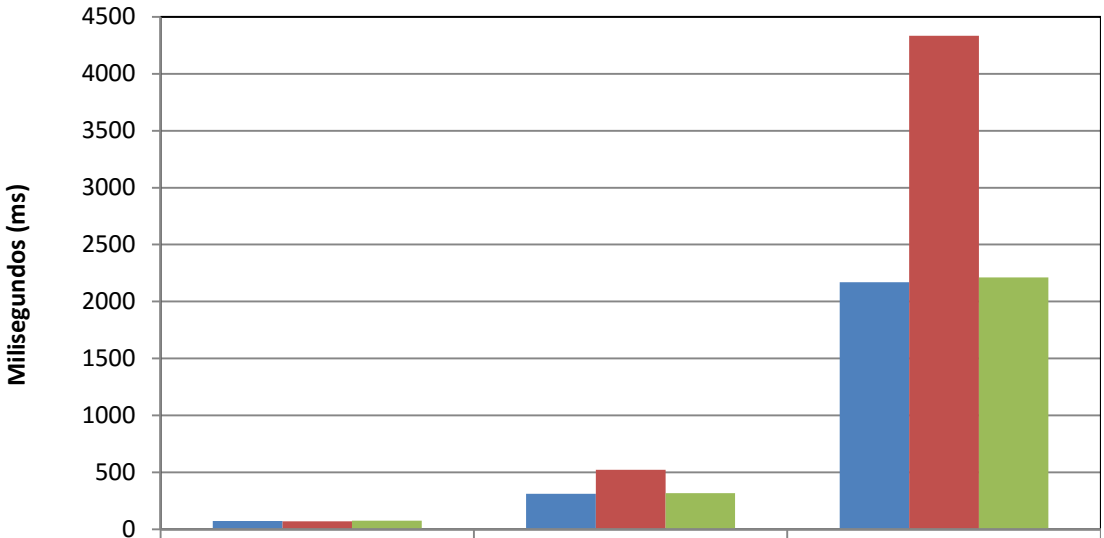
Figura 33. Tiempo de despliegue.

Debido al valor bajo de los tiempos y a la cantidad de muestras se decidió tomar el mínimo y máximo del tiempo. Se puede observar que entre los dos (2) primeros casos de prueba hay una diferencia baja de tiempo en cualquier navegador, y el último caso presenta una mayor tardanza. También se puede observar que el navegador **Opera** presenta una mayor tardanza en todos los casos de prueba.

5.3.2 Prueba de tiempo desde el procesamiento del código hasta visualizar una salida gráfica

Para establecer una comparación entre utilizar y no utilizar la aplicación se realizó la siguiente prueba, la cual toma en consideración el tiempo desde que se ordena ejecutar el código hasta que se genera la salida gráfica correspondiente. Luego son comparados los tiempos y se muestra la diferencia de tiempo entre ejecutar un despliegue y ejecutar un despliegue luego de hacer un procesamiento de un lenguaje.

La Fig. 34 muestra los promedios obtenidos de ejecutar cada caso de prueba con procesamiento de lenguaje tomando 5 muestras de tiempo. La Fig. 35 muestra la diferencia de tiempo entre los tiempos de despliegue y los tiempos desde el procesamiento del código hasta el despliegue.

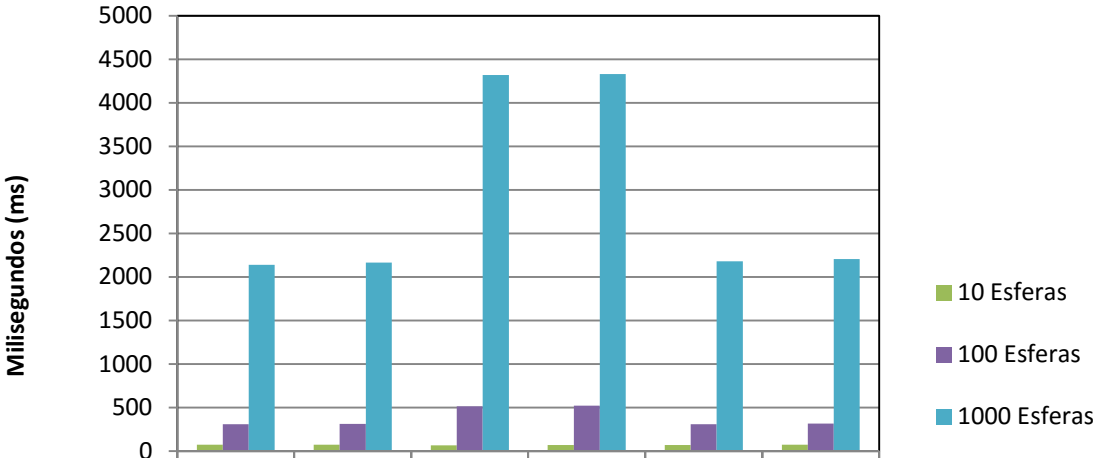


	10 Esferas	100 Esferas	1000 Esferas
Google Chrome	72,8	311,8	2169,0
Firefox	69,6	520,8	4334,6
Opera	74,0	315,8	2210,8

Figura 34. Tiempo promedio desde el procesamiento hasta la visualización

Se puede observar en la Fig. 34 que los tiempos entre casos de pruebas son muy diferentes. Además, se observa una tardanza mucho mayor en el navegador **Firefox** para todos los casos de prueba.

En la Fig. 35 se puede observar la gran diferencia de tiempo que añade el procesamiento del código en los casos de prueba. Sin embargo, este tiempo añadido sólo se presenta al interpretar el código. Una vez que el código haya sido interpretado, todos los **frames** a continuación consumirán sólo tiempo de despliegue, pues sólo se interpreta una vez el código y se almacena la información de la escena para ser desplegada periódicamente.



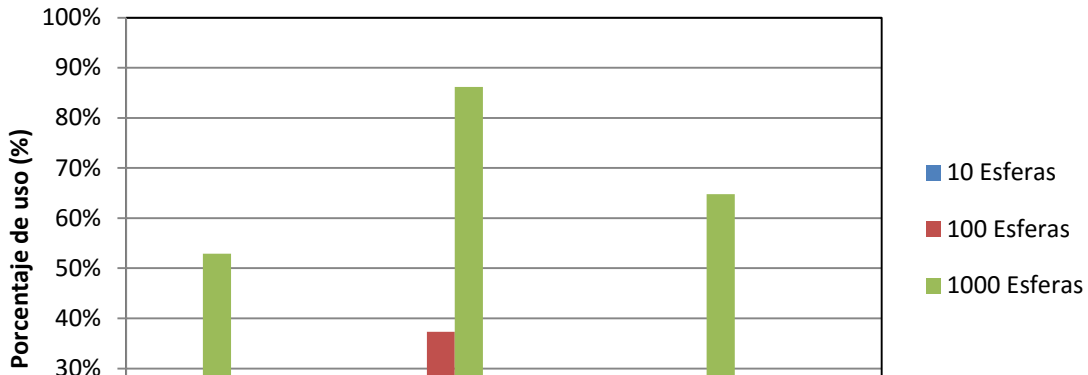
	Google Chrome		Firefox		Opera	
	Min	Max	Min	Max	Min	Max
10 Esferas	70,8	71,8	64,6	68,6	67	73
100 Esferas	305,8	310,8	514,8	519,8	305,8	315
1000 Esferas	2139,0	2165	4317,6	4327,6	2176,8	2203

Figura 35. Diferencia de tiempo entre el tiempo de despliegue y el tiempo total desde el procesamiento del código.

5.3.3 Prueba de recursos consumidos por la aplicación

El objetivo de la prueba es examinar la cantidad de recursos consumidos por la solución propuesta. Estos recursos se refieren a uso de la CPU y uso de memoria. La prueba toma en cuenta desde el momento de procesamiento del código en la extensión propuesta hasta la ejecución del primer **frame** del despliegue.

La Fig. 36 muestra los recursos consumidos con las características de la máquina anteriormente descritas.



	Google Chrome		Firefox		Opera	
	CPU	Memoria	CPU	Memoria	CPU	Memoria
10 Esferas	9,0%	0,7%	21,4%	1,6%	9,6%	0,6%
100 Esferas	12,0%	1,2%	37,3%	1,8%	11,6%	1,3%
1000 Esferas	52,9%	3,7%	86,2%	15,9%	64,8%	4,6%

Figura 36. Recursos consumidos.

Se puede observar que el porcentaje de recursos consumidos por la aplicación entre los dos (2) primeros casos de prueba no presenta una gran diferencia. Sin embargo, el último caso de prueba presenta un gran consumo de recursos tanto la CPU como memoria. Además, el navegador **Firefox** presenta un mayor consumo de recursos en todos los casos de prueba.

5.4 Prueba de comparación entre la salida gráfica generada por la aplicación y la generada por las *Plantillas*

Esta prueba mide los objetos/modelos y características de estos que son desplegados con la solución propuesta y las *Plantillas* generadas. Para esto se desarrollaron tres (3) casos de prueba:

1. Tomando en cuenta los objetos/modelos desplegados.
2. Tomando en cuenta las transformaciones.
3. Tomando en cuenta la iluminación y sombreado.

1. Tomando en cuenta los objetos/modelos desplegados:

En la Fig. 37 se muestra la salida gráfica generada por las *Plantillas* y por la aplicación al ejecutar el código que se ve reflejado en la Fig. 38.

Como se puede observar, la cantidad de modelos/objetos desplegados es la misma utilizando ambas herramientas. Además, los modelos/objetos desplegados por ambas herramientas son iguales. Así pues, se obtuvo un 100% de equivalencia entre la salida generada por la aplicación y la generada por las *Plantillas*.

Figura 37. Salida generada por la aplicación (lado izquierdo) y por las *Plantillas* (lado derecho), Tomando en cuenta los objetos/modelos desplegados.

```
1  DrawSphere()
2  DrawCone()
3  TranslateObject({-2, 0, 2})
```

Figura 38. Código utilizado para la ejecución de la prueba, Tomando en cuenta los objetos/modelos desplegados.

2. Tomando en cuenta las transformaciones:

La Fig. 39 muestra la salida gráfica generada por ambas herramientas, obtenida de la ejecución del código que se muestra en la Fig. 40. Como se puede observar, el modelo/objeto fue escalado al doble de su tamaño utilizando ambas herramientas. La salida gráfica es igual utilizando ambas herramientas, por lo que se obtiene un 100% de equivalencia en las transformaciones realizadas a objetos/modelos.

Figura 39. Salida generada por la aplicación (lado izquierdo) y por las *Plantillas* (lado derecho), Tomando en cuenta las transformaciones.

```
1  DrawSphere()
2  ScaleObject({2,2,2})
```

Figura 40. Código utilizado para la ejecución de la prueba, Tomando en cuenta las transformaciones.

3. Tomando en cuenta la iluminación y sombreado:

La Fig. 41 muestra la salida generada por ambas herramientas, obtenida de la ejecución del código que se muestra en la Fig. 42. Como se puede observar, se desplegó una esfera y una luz puntual. Como modelo de iluminación se utiliza Blinn-Phong y como modelo de sombreado Phong. También se puede observar que la iluminación y sombreado son iguales utilizando ambas herramientas, por lo que se obtiene 100% de equivalencia en la iluminación y sombreado.

Figura 41. Salida generada por la aplicación (lado izquierdo) y por las *Plantillas* (lado derecho), Tomando en cuenta la iluminación y sombreado.

```
1  DrawSphere()
2  DrawPointLight({3,3,3})
```

Figura 42. Código utilizado para la ejecución de la prueba, Tomando en cuenta la iluminación y sombreado.

Capítulo 6
Conclusiones y Trabajos Futuros

6.1 Conclusiones

En este trabajo se presentó una extensión para el lenguaje interpretado Lua. Esto trajo como consecuencia que el esfuerzo programático es reducido drásticamente, debido a la reducción en el número de líneas necesarias para obtener una salida gráfica.

La extensión facilita el desarrollo de escenas complejas, debido al uso de estructuras iterativas o de control que provee. Al mismo tiempo, se logró implementar las funcionalidades propuestas en el alcance con el uso de un subconjunto del lenguaje Lua como lo demuestran las pruebas realizadas.

La diferencia de tiempo añadido debido al procesamiento del código es aceptable, ya que el tiempo de espera por una salida gráfica sólo se ve afectado considerablemente al desplegar 2.880.000 triángulos o más. La aplicación consume una cantidad de recursos aceptable y sólo se ve afectada considerablemente a partir de 2.880.000 triángulos.

Gracias a las *Plantillas* generadas, se puede estudiar y utilizar el código que genera la salida gráfica de la escena, haciendo de este un proyecto de software libre que promueve la evolución de las técnicas para el despliegue de gráficos.

Las *Plantillas* generadas despliegan los mismos objetos/modelos con las mismas características que se desarrollaron en la aplicación, garantizando así una equivalencia entre la salida gráfica generada por la aplicación y por las *Plantillas*.

6.2 Trabajos Futuros

Actualmente la solución propuesta se sigue mejorando y se puede observar en el repositorio (https://github.com/ADDR2/Lua-Interpreter-for-Computer-Graphics). Como trabajos futuros, se plantean los siguientes:

- Extensión de las funcionalidades o herramientas del ámbito gráfico que provee la aplicación. Como por ejemplo, implementación de sombras.

- Adición de nuevos lenguajes conocidos para la generación de *Plantillas*, de manera que se pueda exportar el despliegue desarrollado a diferentes lenguajes.

Bibliografía

-K. C. Louden, K. A. Lambert, “Programming Languages Principles and Practice”, 3era ed, 2011.

-R. W. Sebesta, “Concepts of Programming Languages”, 10ma ed, pp. 05-07, 2012.

-A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, “Compiladores principios, técnicas y herramientas”, 2da ed, 2012.

-J. Yee, “A Survey Of Graphics Programming Languages”, 2004. [En línea]. Disponible en: https://classes.soe.ucsc.edu/cmps203/Fall04/finalreports/ProjectPaper_JerryYee.pdf

-I. Stephenson, “Essential RenderMan”, 2da ed, 2007.

- ----, “Babylon.js: a complete JavaScript framework for building 3D games with HTML 5 and WebGL”, 2013. [En línea]. Disponible en: http://blogs.msdn.com/b/eternalcoding/archive/2013/06/27/babylon-js-a-complete-javascript-framework-for-building-3d-games-with-html-5-and-webgl.aspx

- R. Cabello, “Three.js Documentation”, 2010. [En línea]. Disponible en: http://threejs.org/docs/#Manual/Introduction/Creating_a_scene

-M. D. Benedetto, F. Ponchio, F. Ganovelli, R. Scopigno, “SpiderGL: A JavaScript 3D Graphics Library for Next-Generation WWW”, 2010. [En línea]. Disponible en: http://vcg.isti.cnr.it/Publications/2010/DPGS10/spidergl.pdf

- Z. Carter, “Jison your friendly JavaScript parser generator”, 2009. [En línea]. Disponible en: http://zaach.github.io/jison/

- K. Matsuda, R. Lea, “WebGL Programming Guide: Interactive 3D Graphics Programming with WebGL”, 2013.

- E. Angel, D. Shreiner, “Interactive Computer Graphics: A Top-Down Approach with WebGL”, 7ma ed, 2014.

- D. Shreiner, G. Sellers, J. M. Kessenich, “OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3”, 8va ed, 2013.

- J. F. Hughes, A. V. Dam, M. Mcguire, D. F. Sklar, J. D. Foley, S. K. Feiner, K. Akeley, “Computer Graphics: Principles and Practice”, 3era ed, 2014.

- ----, “Lua 5.3 Reference Manual”, 2015. [En línea]. Disponible en: https://www.lua.org/manual/5.3/

- w3schools, “Color Names Supported by All Browsers”. [En línea]. Disponible en: http://www.w3schools.com/colors/colors_names.asp