CHAPTER 1

INTRODUCTION

1.1 Background

Lua is designed, implemented, and maintained by a team at PUC-Rio, the Pontifical Catholic University of Rio de Janeiro in Brazil. Lua was born and raised in Tecgraf, formerly the Computer Graphics Technology Group of PUC-Rio. Lua is now housed at LabLua, a laboratory of the Department of Computer Science of PUC-Rio. "Lua" (pronounced **LOO-ah**) means "Moon" in Portuguese. As such, it is neither an acronym nor an abbreviation, but a noun. More specifically, "Lua" is a name, the name of the Earth's moon and the name of the language. Like most names, it should be written in lower case with an initial capital, that is, "Lua". Please do not write it as "LUA", which is both ugly and confusing, because then it becomes an acronym with different meanings for different people.

1.2 Introduction

Lua is a powerful, efficient, lightweight, embeddable scripting language. It supports procedural programming, object-oriented programming, functional programming, data-driven programming, and data description. Lua combines simple procedural syntax with powerful data description constructs based on associative arrays and extensible semantics. Lua is dynamically typed, runs by interpreting byte code with a register-based virtual machine, and has automatic memory management with incremental garbage collection, making it ideal for configuration, scripting, and rapid prototyping.

1.2.1 Characteristics

- Lua is a proven , robust language

  Lua has been used in many industrial applications (e.g., Adobe's Photoshop Lightroom), with an emphasis on embedded systems (e.g., the Ginga middleware for digital TV in Brazil) and games (e.g., World of Warcraft and Angry Birds). Lua is currently the leading scripting language in games. Lua has a solid reference manual and there are several books about it. Several versions of Lua have been released and used in real applications since its creation in 1993. Lua featured in HOPL III, the Third ACM SIGPLAN History of Programming Languages Conference, in June 2007. Lua won the Front Line Award 2011 from the Game Developers Magazine.

- Lua is fast

  Lua has a deserved reputation for performance. To claim to be "as fast as Lua" is an aspiration of other scripting languages. Several benchmarks show Lua as the fastest language in the realm of interpreted scripting languages. Lua is fast not only in fine-tuned benchmark programs, but in real life too. Substantial fractions of large applications have been written in Lua.

- Lua is free

  Lua is distributed in a small package and builds out-of-the-box in all platforms that have a standard C compiler. Lua runs on all flavors of Unix and Windows, on mobile devices (running Android, iOS, BREW, Symbian, Windows Phone), on embedded microprocessors (such as ARM and Rabbit, for applications like Lego MindStorms), on IBM mainframes, etc.

1.3 Problem Statements

Lua is a brilliantly economical tool for solving many programming problems. Unfortunately, its economy and flexibility of design can confuse the newcomer: they may find a clumsy solution to their problem, or worse, not see one at all, when there is a simple and elegant solution waiting to be found. Unlike users of most languages, who simply program in them, Lua programmers will often want to embed, interface to, or even change Lua. Various libraries and tools have grown up to meet many of these needs, such as to lua, CGILua and Lua Socket. However, some needs are more abstract, and cannot easily be met by a tool or library questions such as:

1.4 Motivation

Lua is distributed in a small package and builds out-of-the-box in all platforms that have a standard C compiler. Lua runs on all flavors of Unix and Windows, on mobile devices (running Android, iOS, BREW, Symbian, Windows Phone), on embedded microprocessors (such as ARM and Rabbit, for applications like Lego Mind Storms), on IBM mainframes, etc. A fundamental concept in the design of Lua is to provide meta-mechanisms for implementing features, instead of providing a host of features directly in the language. For example, although Lua is not a pure object-oriented language, it does provide meta-mechanisms for implementing classes and inheritance. Lua's meta-mechanisms bring an economy of concepts and keep the language small, while allowing the semantics to be extended in unconventional

ways.

1.5 Existing Game Engines

Lua is used in a lot of game engines due to its simple language structure and syntax. The garbage collection feature is often quite useful in games which consume a lot of memory due to rich graphics that is used. Some game engines that use Lua includes -

- **Corona SDK**
- Gideros Mobile
- ShiVa3D
- Moai SDK
- LOVE
- CryEngine

Each of these game engines are based on Lua and there is a rich set of API available in each of these engines. Each of these Game SDKs/frameworks have their own advantages and disadvantages. A proper choice between them makes your task easier and you can have a better time with it. So, before using it, you need to know the requirements for your game and then analyze which satisfies all your needs and then should use them.

1.6 Standard Libraries

Lua standard libraries provide a rich set of functions that is implemented directly with the C API and is in-built with Lua programming language. These libraries provide services within the Lua programming language and also outside services like file and db operations.These standard libraries built in official C API are provided as separate C modules. It includes the following −

- Basic library, which includes the coroutine sub-library
- Modules library
- String manipulation , Table Manipulation and Math Library.
- File Input and output

CHAPTER 2

LITERATURE SURVEY

Lua was born in 1993 inside Tecgraf, the Computer Graphics Technology Group of PUC-Rio in Brazil. The creators of Lua were Roberto Ierusalimschy (then an assistant professor of the Department of Computer Science of PUC-Rio), Luiz Henriquede Figueiredo (the post-doctoral fellow at IMPA and later at Tecgraf), and Waldemar Celes (then a Ph.D. student at PUC-Rio).

All three were members of Tecgraf, working on different projects there before getting together to work on Lua. Roberto was a computer scientist whose main interest was already in programming languages.

Luiz Henrique was a mathematician interested in software tools and computer graphics. Waldemar was an engineer interested in applications of computer graphics.

Tecgraf is a large research-and-development laboratory with several industrial partners.

During the first ten years since its creation in May 1987, Tecgraf focused mainly on producing basic software tools to enable it to produce the interactive graphical programs its clients needed.

Accordingly, the first Tecgraf products were drivers for graphical terminals, plotters, and printers; graphical libraries; and graphical interface toolkits.

From 1977 until 1992, Brazil had an official market reserve policy for computer hardware and software. There was a general feeling that Brazil could and should produce its own hardware and software. In that atmosphere, access to existing software and to high-end hardware was very difficult in Brazil.

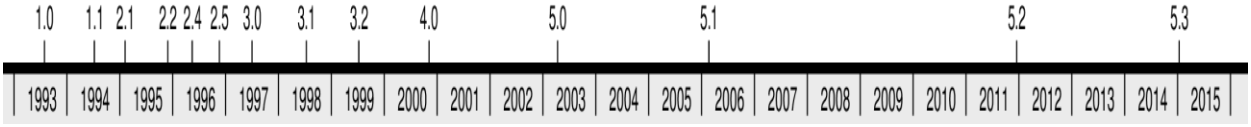
Tecgraf's clients could not afford, either politically or financially, to buy customized software from abroad: they would have to go through a complicated process of proving that their needs could not be met by the Brazilian industry.

Added to the natural geographical isolation of Brazil from other research centers, those reasons led Tecgraf to implement from scratch the basic tools it needed.

One of Tecgraf's largest partners was(and still is) Petrobras,the Brazilian oil company. Several Tecgraf products were (and still are) interactive graphical programs for engineering applications at Petrobras.

By 1993, Tecgraf had developed little languages for two of those applications: a data-entry application and a configurable report generator for lithology profiles. These languages, called DEL and Sol, were the ancestors of Lua.

Version History



Lua 1.0 was never released publicly, but it was up and running on 28 Jul 1993, and most probably a couple of months before that.

Lua 1.1 was released on 08 Jul 1994. This was the first public release of Lua and is described in a conference paper. Lua 1.1 already featured powerful data description constructs, simple syntax, and a bytecode virtual machine. Lua 1.1 was freely available for academic purposes; commercial uses had to be negotiated, but none ever were.

Lua 2.1 was released on 07 Feb 1995. Its main new features were extensible semantics via fallbacks and support for object-oriented programming. This version was described in a journal paper. Starting with Lua 2.1, Lua became freely available for all purposes, including commercial uses.

Lua 2.2 was released on 28 Nov 1995. Its main new features were long strings, the debug interface, better stack tracebacks, extended syntax for function definition, garbage collection of functions, and support for pipes.

Lua 2.3 was never released publicly; it only existed as a beta version.

Lua 2.4 was released on 14 May 1996. Its main new features were the external compiler *luac*, an extended debug interface with hooks, and the "getglobal" fallback.

Lua 2.5 was released on 19 Nov 1996. Its main new features were pattern matching and vararg functions.

Lua 3.0 was released on 01 Jul 1997. Its main new feature was tag methods as a powerful replacement for fallbacks. Lua 3.0 also introduced auxlib, a library for helping writing Lua libraries, and support for conditional compilation (dropped in Lua 4.0).

Lua 3.1 was released on 11 Jul 1998. Its main new features were anonymous functions and function closures via "upvalues". (Lua 5.0 brought full lexical scoping and dropped upvalues.) This brought a flavor of functional programming to Lua. There was also support for multiple global contexts; however, the API was not fully reentrant (this had to wait until Lua 4.0). Lua 3.1 also saw a major code re-organization and clean-up, with much reduced

module interdependencies. Lua 3.1 also adopted double precision for the internal representation of numbers.

Lua 3.2 was released on 08 Jul 1999. Its main new features were a debug library and new table functions. The last release was Lua 3.2.2, released on 22 Feb 2000. There will be no futher releases of Lua 3.2.

Lua 4.0 was released on 06 Nov 2000. Its main new features were multiples states, a new API, "for" statements, and full speed execution with full debug information. Also, Lua 4.0 no longer has built-in functions: all functions in the standard library are written using the official API. The last release was Lua 4.0.1, released on 04 Jul 2002. There will be no futher releases of Lua 4.0.

Lua 5.0 was released on 11 Apr 2003. Its main new features were collaborative multithreading via Lua coroutines, full lexical scoping instead of upvalues, and metatables instead of tags and tag methods. Lua 5.0 also introduces booleans, proper tail calls, and weak tables. Other features are better support for packages, new API for loading Lua chunks, new error handling protocol, better error messages, and much more. Lua 5.0 was the first version to be released under the new license. The last release was Lua 5.0.3, released on 26 Jun 2006. There will be no futher releases of Lua 5.0.

Lua 5.1 was released on 21 Feb 2006. Its main new features were a new module system, incremental garbage collection, new mechanism for varargs, new syntax for long strings and comments, mod and length operators, metatables for all types, new configuration scheme via luaconf.h, and a fully reentrant parser. The last release was Lua 5.1.5, released on 17 Feb 2012. There will be no futher releases of Lua 5.1.

Lua 5.2 was released on 16 Dec 2011. Its main new features are yieldable pcall and metamethods, new lexical scheme for globals, ephemeron tables, new library for bitwise operations, light C functions, emergency garbage collector, goto statement, and finalizers for tables. The last release is Lua 5.2.4, released on 07 Mar 2015.

Lua 5.3 was released on 12 Jan 2015. Its main new features are integers, bitwise operators, a basic utf-8 library, and support for both 64-bit and 32-bit platforms. The current release is Lua 5.3.4, released on 30 Jan 2017.

Lua in Games

Features that make Lua special are important to games:

Portability: Many games run on non-conventional platforms, namely game consoles like Microsoft XBox and Sony PlayStation. 8 Languages designed by large committees tend to be too complicated and never quite fulfill the expectations of their sponsors. Most successful languages are raised rather than designed. They follow a slow bottom-up process, starting as a small language with modest goals. The language evolves as a consequence of actual feedback from real users, from which design flaws surface and new features are identified.

Easiness of embedding: Games are demanding applications. They need both performance, for its graphics and simulations, and flexibility for the creative staff. Not by chance, many games are coded in (at least) two languages, one for scripting and the other for coding the engine. Within that framework, the ease of integrating Lua with another language (mainly C++, in the case of games) is a big advantage.

Simplicity: Most game designers are not professional programmers. For them, a language with a simple syntax and a simple semantics is particularly important.

Efficiency and small size: Games are demanding applications, and game consoles are less powerful than typical microcomputers. Therefore, the script interpreter should be parsimonious with resources. Besides those points, other characteristics of Lua have made it specially attractive to games: Unlike most other software enterprises, games production involves little evolution. Once a game is released, there are no updates or new versions, only new games. So, it is easier to risk using a new language in a game. Game developers do not care whether the language will evolve or how it will evolve. All they need is the version they used in the game. The Lua license is also convenient for games. Most big games are not open source (some game companies even refuse to use any kind of open-source code). The competition is hard, and game companies tend to be secretive about their technologies. For them, a liberal license like the MIT license is quite convenient. (Prior to version 5, Lua used a MIT-like license; since version 5 it uses the MIT license.) Finally, Lua has features that should be really attractive to games but we do not have first-hand knowledge that they are indeed used in games: co routines, as a means to implement multi-tasking in games; and Lua data files, which can replace textual data files with many benefits.

CHAPTER 3

ANALYSIS

Corona is a Lua-based framework that provides a large set of APIs and plugins, allowing developers to quickly and easily make apps that can run on multiple types of devices. With the built-in Corona Simulator, you can preview how your app will appear and respond on many types of devices, all without having to compile or deploy it for testing. When you're ready, you can conduct real-time device testing to see how your app performs on actual devices.In addition to the core framework, Corona includes many plugins which add specific functionality and help speed up the development of your app.

What is Corona?

Founded in 2008, Corona Labs Inc. is a company based in Palo Alto with extensive technical and commercial experience across several platforms; mobile ecosystem, software platforms, authoring tools, runtimes and cloud services. His previous experience includes leadership positions at companies such as Adobe, Apple, Macromedia and Microsoft.

Notably Corona Lua is developed using an easy to learn programming language, along with the API, which allows you to add features similar to Facebook with just a few lines of code.

APIs for Corona

The Corona´s API library has over a thousand APIs, enabling the creation of applications for commercial, electronic books and games for any subject supported platforms. You will see the changes instantly in the simulator and are able to adjust quickly to Lua changes, a language designed for quick and easy programming (easy to learn too).

Corona allows you to publish to iOS, Android, Kindle Fire and the Nook with a single code base and soon windows 8 too as they plan to incorporate it due to the rise in popularity this platform has had.

Developer Community

Corona has earned its place in the market in part thanks to the support given by its developer community, as their forums are full of enthusiastic developers offering advice, sharing code

and generally helping out. Corona also promotes regional or local groups to collaborate and network, which had let to the existence of hundreds of guides, tutorials, videos and sample projects to make life easier for fellow developers.

This tool provides a space for experimentation and study on its official website and goes even further by offering discounts on their licenses for educational purposes.

A Small Tutorial to Start Using Corona

Underneath is a short tutorial on getting started with Corona

Step 1 – Download and install Corona SDK

Once installed you can launch the Corona Simulator, where you're able to create new projects or use the sample projects that are a perfect starting point to learn.

Step 2 – Start a new project

To create a new project, you need only click on the "New Project" option. In the New Project window you enter the name of the new application, game or scene. You can set the default resolution and whether you want the application to be displayed vertically or horizontally.

Step 3 – Saving and opening in the Simulator

After choosing the where you want to save your application you can open the device simulator and chose the base application and configuration files. These files are:

· **main.lua.** This is the main project file. It is the first thing you'll need to run after starting the application.

· **config.lua.** This is the configuration file for your project. Here you're able to alter the settings for different devices.

· **build.settings.** This is the configuration file using Corona to create the application on different platforms, such as application permissions for Android.

To begin creating your application all you need to do is add the code of your app within the main.lua file.

The complexity of your application or game is set by your code, but the basic steps are the ones explained above.

Text Editors

In addition to Corona, you will need a **text editor**. There are several editors available and choosing one is much like choosing a car — everybody has their own preferences and you should explore which one suits you best.

If you don't already have a favorite text editor, the following options are recommended:

Editor	Add-On Package	macOS	Windows
Atom	autocomplete-corona	✓	✓
Xcode	Corona Plugin for Xcode	✓	
Visual Studio Code	Corona Tools	✓	✓
Sublime Text	Corona Editor	✓	✓
Vim		✓	✓
ZeroBrane Studio		✓	✓
Notepad++			✓
TextWrangler		✓	
TextMate		✓	

Pros

- In general, it takes significantly less code to implement a given functionality with Corona than the Java equivalent. In particular, dealing with REST services and OpenGL are greatly simplified by Corona.
- The Corona simulator runs circles around the Android emulator Google ships with its SDK. It is very easy to change the skin of the Corona simulator to represent a wide array of Android and iOS devices.

- Corona scales graphics brilliantly. The system used by Corona for choosing the best asset for a particular device, as well as its ability to auto-scale, seems more intuitive than the native Android approach, and it seems to work better as well.
- Setting up the development environment takes just minutes and is much less convoluted than installing Android's Eclipse/SDK/plugin environment.
- **A Corona app's biggest advantage:** It compiles for both iOS and Android.

Cons

- Debugging Corona applications can be painful. I have not found a good IDE or debugger, and the result is troubleshooting even simple issues can be tedious.
- Lua's loose syntax is ugly. Even an experienced and disciplined programmer will have to work very hard not to end up with spaghetti code. Corona's storyboard API helps when it comes to organizing the code, but compared to a strictly typed and scoped Java application, it leaves a lot to be desired. With Corona, it is very easy to introduce memory leaks; plus, it can be a bear to find those leaks.
- Some seemingly simple user interface elements have zero or crude support. A good example is popping up and using the keyboard on a device with a standard input field. This is a pretty common occurrence for a large number of applications, yet the limited support for this in Corona can result in a clunky or non-standard user interface.
- **Corona's biggest drawback:** Google's Android SDK and Apple's iOS development kits are free. There is free edition of the Corona SDK, but you really need to shell out $600 for the Pro version to get the full set of features. More than that, Corona works on a subscription model, so you have to take that $600 hit every year to keep your subscription current.

CHAPTER 4

APPLICATIONS

Creating an App

We are going to make a simple tapping game to keep a balloon in the air. Each time the balloon is tapped, we will "push" it a little higher .

Working of the Game:

We need to create a new project using the Corona simulator

There are three images :- background , balloon and the platform .

Load the Background , Platform and the balloon.

Add Physics to the game for bouncing the balloon as and when it is tapped .

Here the platform is static .

Functions:

- At this point in the game , the balloon drops on to the platform and bounces slightly off . This is no fun since we require the tap function .
- On tapping the balloon each time we need to push it higher for certain moment and then come back again and drop on the platform .
- When applied to a dynamic physical object like the balloon, it applies a "push" to the object in any direction. The parameters that we pass tell the physics engine how much force to apply (both horizontally and vertically) and also where on the object's body to apply the force.
- If you apply the force at a location which is not the center of the balloon, it may cause the balloon to move in an unexpected direction or spin around. For this game, we will keep the force focused on the center of the balloon.
- Events are what create interactivity and, in many ways, Corona is an event-based framework where information is dispatched during a specific event to an **event listener**.
- Whether that event is the user touching an object/button, tapping the screen, or (in this game) tapping the balloon, Corona can react by triggering an event.
- A tap Count is also given , to count the number of times the balloon is tapped .

The Three Images :

Fig 4.1: balloon

Fig 4.2: platform

Fig 4.3: Background

The three images combined together :

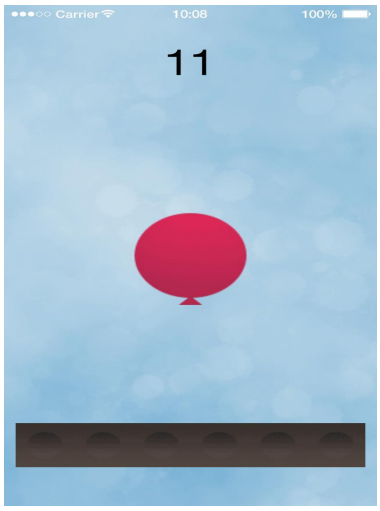


Fig 4.4: Game

Pop The Bubble : Application

The Game Logic Is Very Simple , “We need to pop the Required bubbles using the bullets to complete the level”.

Note: “The bullets are given only once and all we need to do is to tap on the screen to get them”.

Fig 4.5: Pop the Bubble Game

Requirement:

- Open the **Corona Simulator**.
- Click **New Project** from the welcome window or select **New Project...** from the **File** menu.
- For the project/application name, type Pop the bubble and ensure that the **Blank** template option is selected. Leave the other settings at default and click **OK** (Windows) or **Next** (Mac). This will create the basic files for your first game in the location (folder) that you specified. This is also the folder in which you'll place all of your app files/assets, including images, program files, etc.
- Include the images that needs to be used in order to develop the game .
- Differentiate the images using static and dynamic objects .

- Add the physics engine to the game .
- Add the functionality and events for the logic of the game .

Working :

- There are 4 levels to begin with , each level having its own level of difficulty .
- On opening the application , there are two buttons , Play and Credits .
- After Clicking the Play button a page appears which shows different levels available in the game .
- The level page displays the “4 Levels” .
- Each Level has its own set of difficulties within .
- In each level we get only 4 bullets and they spread apart 90˚ from the point of tap.
- Level 1: There are 5 bubbles and the required number of bubbles to pop is 4.
- Level 2: There are 4 bubbles and the required number of bubbles to pop is 4.There is an additional obstacle that disturbs the flow of the bubbles and not the bullets. The obstacle is smaller here .
- Level 3: There are 5 bubbles and the required number of bubbles to pop is 5.There is an additional obstacle that disturbs the flow of the bubbles and not the bullets. The obstacle is bigger than level 3.
- Level 4: There are 4 bubbles and the required number of bubbles to pop is 4 with two obstacles of same size.
- The Level is Completed only if the required number of bubbles are popped , else the level is failed .
- A Level Completed message occurs once the level is completed and Level Failed message occurs if the required number of bubbles is not popped.

CHAPTER-5

ADVANTAGES AND DISADVANTAGES

ADVANTAGES:

- Small: 20000 lines of C code that can be built into a 182K executable interpreter (under Linux).
- Portable: builds on any platform with an ANSI C compiler. You can see it running on almost anything from microcontrollers and Lego Minstorms NXT, to game engines, to mobile toolkits, to game consoles, to a browser (translated to JavaScript).
- Embedded and extensible language that provides a straightforward interface to/from C/C++.
- Sufficiently fast: performs well comparing to other languages and has a JIT compiler that noticeably improves performance on many tasks; those who may still be not satisfied with the performance, can implement critical parts in C and, given the ease of integration with C, still benefit from other good aspects replaced shootout results that are no longer available with benchmarks game.
- Well documented: reference manual, book, wiki, 6-page short reference and more.
- Friendly and enthusiastic community. Between the excellent documentation, the wiki, the mailing list, and Stack Overflow, I didn't have any issues finding answers to my questions.
- Clean and simple syntax suitable for beginners and accessible to non-programmers. Lua has borrowed most of its control syntax from Modula, the descendent of Pascal, which was widely used in education as an introductory language. I still remember using early versions of Philippe Kahn's fast and elegant Turbo Pascal IDE.
- Integrated interpreter: just run lua from the command line.
- Native support for coroutines to implement iterators and non-preemptive multi-threading.
- Incremental garbage collector that has low latency, no additional memory cost, little implementation complexity, and support for weak tables.
- Lexical scoping.
- Functional programming with first class functions and closures.
- Tail calls: return functioncall().
- Recursive functions don't need to be pre-declared: local function foo() ... foo() ... end; note this doesn't work with local foo = function() ... foo() ... end.

- Semicolon as a statement separator is optional (mostly used to resolve ambiguous cases as in a = f; (g).x(a)).
- Overloading using meta tables.
- Meta programming to do things from getting and modifying an abstract syntax tree to creating a new syntax for your DSL.
- Simple yet powerful debug library.
- Fast and powerful JIT compiler/interpreter (LuaJIT) which includes FFI library and is ABI-compatible with Lua 5.1 (this means that it can load binary modules compiled for Lua 5.1).

DISADVANTAGES:

- Tables and strings are indexed from 1 rather than 0.
- Assigning nil as a value removes the element from a table. This is consistent with returning nil for non-existing element, so it makes no difference whether the element does not exist or exists with a value of nil. a = {b = nil} produces an empty table.
- No integers as a separate numeric type; the number type represent real numbers. The next version of Lua (5.3) may change that.
- No classes; object-orientation is implemented using tables and functions; inheritance is implemented using the metatable mechanism.
- Method calls are implemented using object:method(args) notation, which is the same as object.method(object, args) notation, but with object evaluated only once.
- nil and false are the only false values; 0, 0.0, "0" and all other values evaluate as true.
- Non-equality operator is ~= (for example, if a ~= 1 then ... end).
- not, or, and keywords used for logical operators.
- Assignments are statements, which means there is no a=b=1 or if (a=1) then ... end.
- No a+=1, a++, or similar shorthand forms.
- No continue statement, although there is an explanation and a number of alternatives, like using repeat break until true inside the loop to break out of or a goto statement introduced

in Lua 5.2.

- No switch statement.
- Brackets may be required in some contexts; for example, a = {}; a.field works, but {}.field doesn't; the latter needs to be specified as ({}).field.
- A control variable in a loop is localized by default and is not available after the loop.
- Limit and step values in the numeric for loop are cached; this means that in for i = init(), limit(), step() do ... end all three functions init, limit, and step are called once before the loop is executed.
- Conditionals and other control structures require no brackets.
- No class/object finalizers. Lua provides finalizer functionality through the gc metamethod, but it is available only for userdata types (and not tables) and doesn't match the functionality provided by other languages, for example, DESTROY and END methods in Perl. There is an undocumented newproxy feature in Lua 5.1 that allows implementation of finalizers on tables; Lua 5.2 removed that feature as it added support for gc metamethod on tables.
- No yielding between Lua and C code: coroutine.yield call across Lua/C boundary fails with attempt to yield across metamethod/C-call boundary. I happened to come across this error several times as I was doing async programming with luasocket and coroutines, but solved it using the copas module. This has been addressed in Lua 5.2.
- No built-in bit operations in Lua 5.1. This is addressed in LuaJIT (BitOp) and Lua 5.2 (bit32), which both include bit libraries.
- No built-in bit operations in Lua 5.1. This is addressed in LuaJIT (BitOp) and Lua 5.2 (bit32), which both include bit libraries.

CHAPTER-6

RESULTS AND DISCUSSION

The main page that appears when the application is opened . There are two buttons namely Play and Credits .

Fig 6.1: Main Page

After Clicking the Play button this page appears which shows different levels available in the game .

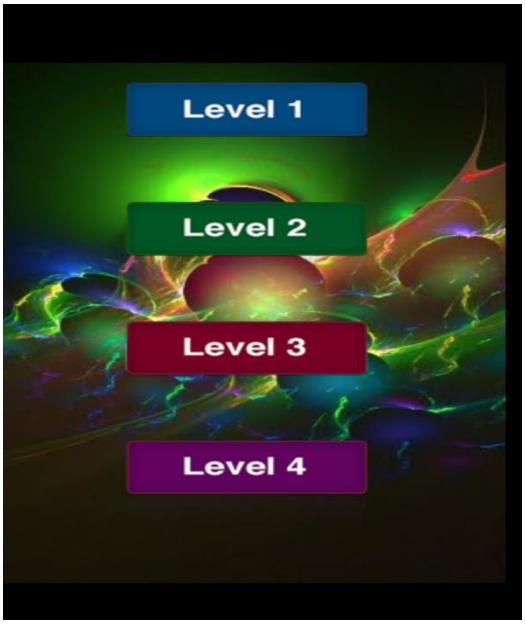


Fig 6.2: Level Page

When Level 1 is opened the game begins and there are 4 bubbles to popped with the 4 bullets available .

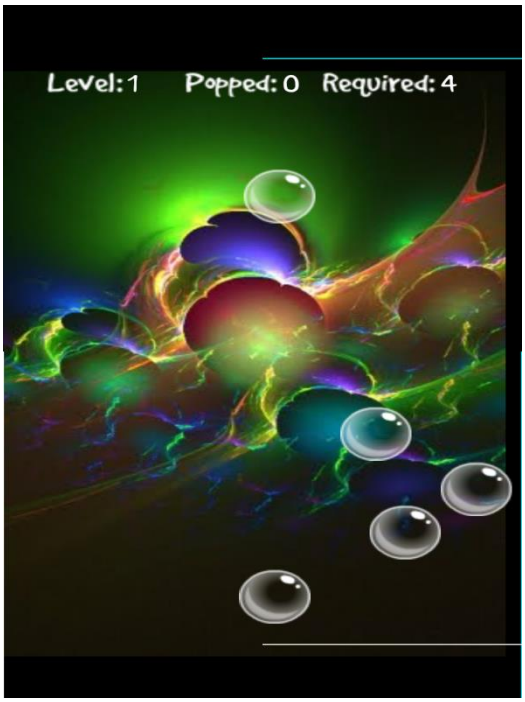


Fig 6.3: Level 1 Page

Bullets fired and the projection that it goes can be seen and they are trying to hit the randomly moving bubbles to complete the level.

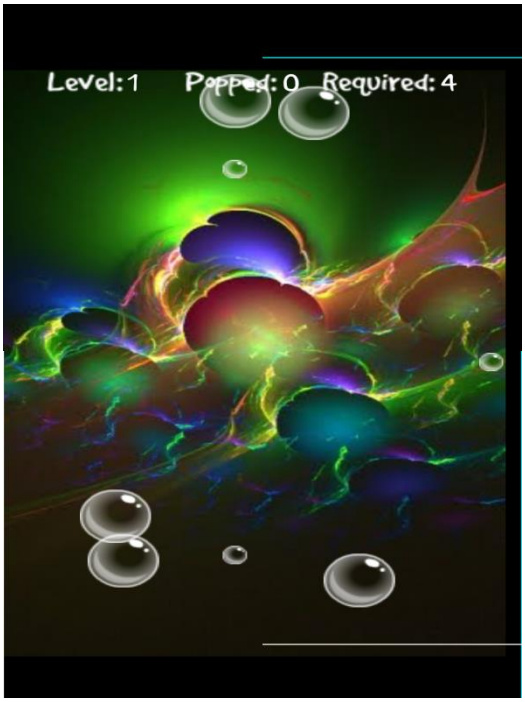


Fig 6.4: Level 1 Page

If the required bubbles are not popped then and alert pops up and shows that Level Failed message.

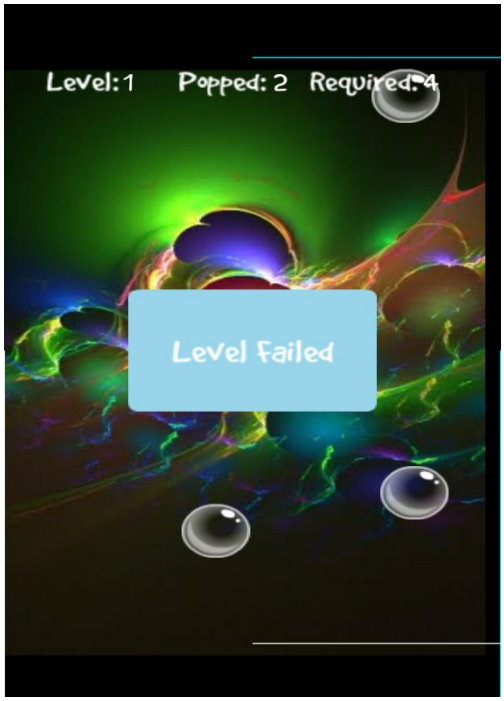


Fig 6.5: Level 1 Page with Level Failed Message

This is Level 2 game , which shows us more number of bubbles are there compared to Level 1 and even the number of required bubbles to pop is 5 .

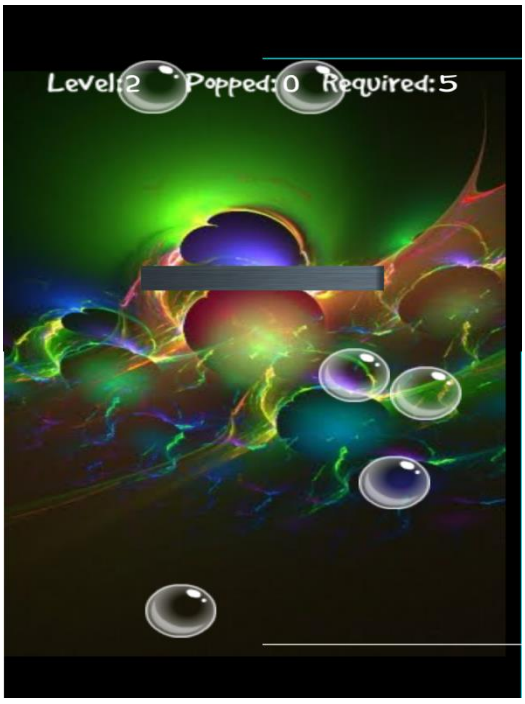


Fig 6.6: Level 2 page

Once the Required number of bubbles are popped in the Level , this means that the Level is Completed and a message alert is shown as Level Complete.

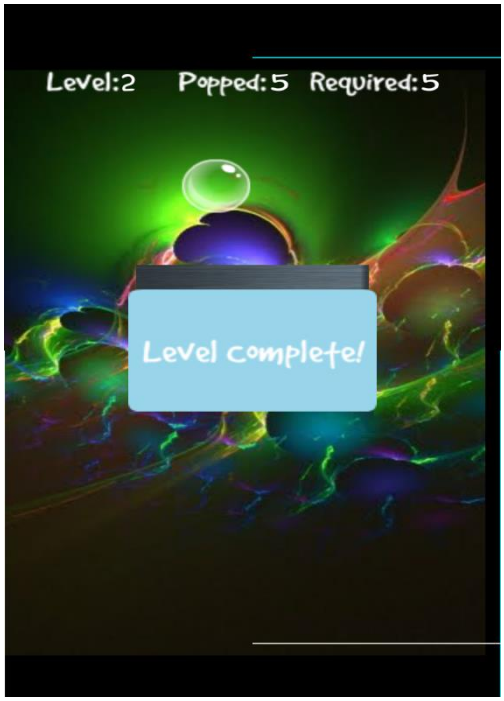


Fig 6.7: Level 2 Page with Level Complete Message

The Level 3 game showing 1 bubble popped and the required is same as that of Level 2 . The obstacle is much bigger than Level 2 and it disturbs the flow of bubbles and not the bullets .

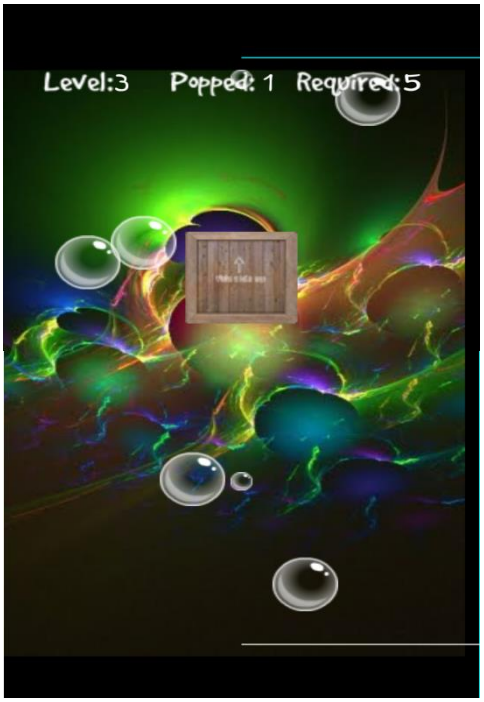


Fig 6.8: Level 3 page

The Level 4 game where there are two obstacles present which obstructs the flow of bubbles .

Fig 6.9: Level 4 page

The level 4 beginning part that tells us the required number of bubbles to pop.

Fig 6.10: Level 4 page

CHAPTER 7

CONCLUSION AND FUTURE WORK

Lua is a scripting language which started as a project of a small group for their personal projects but eventually became famous amongst programmers due to its ease of use, power, speed of execution of commands, portability as well as small size and little over-head.
Lua provides incredible integration with other leading languages though its API libraries and have become a really good extension language due to that fact. Many famous programs, games and game engines have adopted it.
It is really useful for game developers as it is a very secretive industry and having Lua in their game allows modding community to mod the games without even knowing the source code of the actual game which in turn increases the lifetime of that game, web browsers use Lua for extensions and plugins, video games use Lua for their AI logic.
Languages like Lua gives people with no programming background an easier way to get themselves into the field of programming without much investment of time and need of knowing the actual background processes and provides them with a platform to begin their programming career.
All this, comes along with multiple merits and uses which not many languages are able to seize.
The game only has single page to be played on. Additional feature like transition from one page to another needs to be added. More levels needs to be designed and upgraded as per required.
Marketing of the game needs to be learned , so as to put the game in play store .

REFERENCES

- Lua – an extensible extension language by R. Ierusalimschy, L. H. de Figueiredo, W. Celes,
  Software: Practice & Experience **26** #6 (1996) 635–652. [doi]
- The evolution of Lua by R. Ierusalimschy, L. H. de Figueiredo, W. Celes, Proceedings of *ACM HOPL III* (2007) 2-1–2-26.
- https://www.lua.org/docs.html
- Lua: an extensible embedded language by L. H. de Figueiredo, R. Ierusalimschy, W. Celes,
  Dr. Dobb's Journal **21** #12 (Dec 1996) 26–33.
- https://coronalabs.com/resources/tutorials/getting-started-with-corona/
- Passing a language through the eye of a needle by R. Ierusalimschy, L. H. de Figueiredo, W. Celes,
  ACM Queue **9** #5 (May 2011) 20–29.
- https://www.lua.org/versions.html
- Programming in Lua by Roberto Ierusalimschy Lua.org, third edition, January 2013