ARCHITECTURE & DESIGN - GENERAL

Authors

Name	Position	Main responsibility	Email
Simon Hellbe	Architect	Chapter 1, 2 and 5 (App 3)	simon.hellbe @gmail.com
Nima Zafari	Lead Designer	Chapter 3 (App 1) and 4 (App 2)	nimaezafari@ gmail.com

Updates

Sprint	Version	Date	Modification	In charge	Verified by
1	0.1	2014-10-09	First revision done	Simon	Nima
2	0.2	2014-11-07	Document template updated and lots of new info added	Simon	Nima
2	0.3	2014-11-18	Merged all architecture documents	Simon	Nima
2	0.4	2014-11-18	Added header numbering and some information	Simon	Nima
3	0.5	2014-12-02	Updated architecture for apps 1 & 2 to meet quality requirements	Nima	Simon
3	0.6	2014-12-02	Updated architecture and information for app 3	Simon	Nima
3	0.7	2014-12-05	Finalised architecture updates for apps 1 & 2 and some formating changes	Nima	Simon

Table of Contents

1. Introduction

1.1 Terminology

List of terminology used in the document. Updated continuously.

- STB - Set-top Box. Harware unit with TV-tuner and possible other functionality for displaying media on a TV screen or other display device. STB in the document refers to the environment the apps are run in.
- RCU - Remote Control Unit. The input device for the STB.
- Self-contained application. Apps with no external communication. Eg: gaming apps
- One-way communication application. Only receiving information. Eg: newsfeed apps.
- Two-way communication application. Both sending and receiving information. Eg: forum apps

1.2 License & Prerequisites

- The apps will be released under a open source license of TAP interactive's choice.
  - MIT license or GPLv3?
- The boxes have not IP connection within the Lua environment. The app that requires internet service uses a mock-up service provider within the box.
- The Lua environment in the box supports all standard built in libraries except for IP communication.

1.3 Rationale

This section covers main decisions that have been taken and the effects of those. During the development of the apps no major decisions that affects the architecture and design. Some of the restrictions in LuaEngine has affected the configuration and design of the app, though only on a lower level and not the overall architecture and design. Information about these minor issues can be found in technical documents and code comments.

2. General Architecture

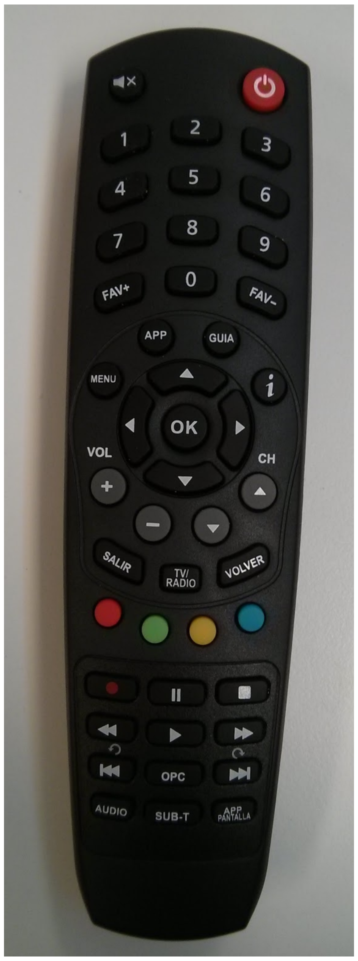
2.1 Hardware

This chapter contains information regarding the Set-top box hardware.

In general, the hardware in STBs provides poor performance in several aspects. Storage on the devices is generally limited, especially on old boxes which only have about 128 mb of storage. Working memory (RAM) is also limited. For reference the total size of an app's code base should be in the range 500 KB - 1000 KB, but could be larger if the app is graphics-heavy.

Remote Control Unit

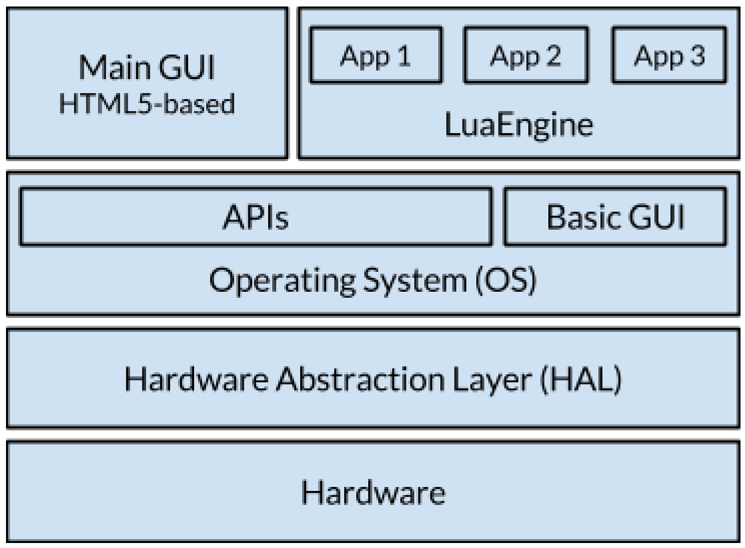
The only input method available is the remote control unit. The remote control has the following buttons:



2.2 Zenterio OS

The Set-top Box Architecture

The STB runs Zenterio's Linux based operating system called Zenterio OS (ZOS). Within the operating system's application layer Zenterio has implemented a Lua environment in which Lua applications can be run in a container. Lua has been selected because of its resource efficiency which is beneficial on the low STB hardware performance.



2.3 LuaEngine API

API

This section describes the operating system API that is available in the Lua environment when the application code is executed.

Execution

After the initial evaluation of the code, the execution is completely event-driven. The following events exist:

onKey(key, state)
key:The following keys are defined (as strings):

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ok, up, down, left, right, red, green, yellow, blue, white, info, menu, guide, opt, help, star, multi, exit, pause, toggle_tv_radio, record, play, stop, fast_forward, rewind, skip_forward, skip_reverse, jump_to_end, jump_to_beginning, toggle_pause_play, vod, backspace, hash, back, ttx, record_list, play_list, mute

state: is either 'down', 'repeat' or 'up'. A keypress generates one 'down', zero or more 'repeat' and exactly one 'up'.

Timer event
User-specified function, taking the timer object as argument.

onStart()
Called once after loading the script into a new environment at start

Surface Data Types

An area (pixmap) in graphics memory. Format is 32-bit RGBA.

surface:clear([color, [rectangle]])
Fills the surface with a solid color, using hardware acceleration. Surface transparency is replaced by the transparency value of <color>. Default color is {0, 0, 0, 0}, that is black and completely transparent. Default rectangle is the whole surface. Parts outside the rectangle are not affected.

surface:fill(color, [rectangle])
Blends the surface with a solid color, weighing alpha values (SRCOVER). Uses hardware acceleration. Default rectangle is the whole surface. Parts outside the rectangle are not affected.

surface:copyfrom(src_surface, src_rectangle, [dest_rectangle, [blend_option]])
Copy pixels from one surface to another, using hardware acceleration. Parts or all of <src_surface> can be copied. Scaling is done if dest_rectangle is of different size than src_rectangle. Areas outside of source or destination surfaces are (will be) clipped.

If <src_rectangle> is nil, the whole <src_surface> is used. If <dest_rectangle> is nil or omitted, x=0, y=0 are assumed and width and height are taken from <src_rectangle>. If <dest_rectangle> doesn't specify width or height, these values are also taken from <src_rectangle>. If <blend_option> is true, copying is blended using the alpha information in <src_surface>. If false, the alpha channel is replaced by the values in <src_surface>. Default is false.

surface:get_width()
Returns the pixel width (X axis) of the surface

surface:get_height()
Returns the pixel height (Y axis) of the surface

surface:get_pixel(x, y)
Returns the color value at position <x>, <y>, starting with index (0, 0). Mostly for testing, not optimized for speed.

surface:set_pixel(x, y, color)
Sets the pixel at position <x>, <y> to <color>. Mostly for testing, not optimized for speed.

surface:premultiply()
Changes the surface pixel components by multiplying the alpha channel into the color channels. This prepares some images for blending with transparency.

surface:destroy()
Frees the graphics memory used by this surface. The same is eventually done automatically by Lua garbage collection for unreferenced surfaces but doing it by hand guarantees the memory is returned at once. The surface can not be used again after this operation.

color
A Lua table of the color and alpha components, in range 0-255.

Valid forms are:

- Specify R, G, B. Alpha is 255 (opaque): {0, 0, 0}
- Specify R, G, B, A: {0, 0, 0, 0}
- Specify components by short name. Omitted components default to -- 0. At least one component must be specified: {r=0, g=0, b=0, a=0}
- Specify components by long name. Omitted components default to -- 0. At least one component must be specified.
- {red=0, green=0, blue=0, alpha=0}

rectangle
A Lua table specifying x, y, width and height. In some cases, width and height are optional. Negative width or height are not allowed. At the moment, negative x or y are not allowed either.

Valid forms are:

- {x=0, y=0, w=0, h=0}
- {x=0, y=0, width=0, height=0}

timer data type

A recurring timer that triggers a configurable callback at a specific interval.

timer:set_interval()
Changes the interval

timer:stop()
No more callbacks are generated, even if a timer timeout already happened and was waiting for dispatching.

timer:start()
The timer starts generating callbacks at the prescribed interval, if previously stopped.

sys module

sys.new_timer(interval_millisec, callback)
Creates and starts a timer that calls <callback> every <interval_millisec>. <callback> can be a function or a string. If a string, the global variable of this name will be fetched each time the timer triggers and that variable will be called, assuming it is a function. In this way, callbacks can be replaced in real-time.

The callback function is called with signature: callback(timer) where <timer> is the timer object that triggered the event

sys.time()
Returns the time since the system started, in seconds and fractions of seconds. Useful to measure lengths of time.

sys.stop()
Terminates the execution of the script. The rest of the currently executing code will be run, but all timers are stopped and the current script environment will never be called again.

gfx module

gfx.set_auto_update(bool)
If set to true, any change to gfx.screen immediately triggers gfx.update() to make the change visible. This slows the system if the screen is updated in multiple steps but is useful for interactive development.

gfx.new_surface(width, height)
Creates and returns a new 32-bit RGBA graphics surface of chosen dimensions. The surface pixels are not initialized; clear() or copyfrom() should be used for this. <width> and <height> must be positive integers and each less than 10000. An error is raised if enough graphics memory cannot be allocated.

gfx.get_memory_use()
Returns the number of bytes of graphics memory the application currently uses. Each allocated pixel uses 4 bytes since all surfaces are 32-bit. A limit of gfx.get_memory_limit() is enforced.

gfx.screen or screen
The surface that shows up on the screen when gfx.update() is called. Calling gfx.set_auto_update(true) makes the screen update automatically when gfx.screen is changed (for development; too slow for animations)

gfx.get_memory_limit()
Returns the maximum bytes of graphics memory the application is allowed to use.

gfx.update()
Makes any pending changes to gfx.screen visible.

gfx.loadpng(path)
Loads the PNG image at <path> into a new surface that is returned. The image is always translated to 32-bit RGBA. Transparency is preserved. A call to surface:premultiply() might be necessary for transparency to work. An error is raised if not enough graphics memory can be allocated.

gfx.loadjpeg(path)
Loads the JPEG image at <path> into a new surface that is returned. The image is always translated to 32-bit RGBA. All pixels will be opaque since JPEG does not support transparency. An error is raised if not enough graphics memory can be allocated.

2.4 Usage of LuaEngine

LuaEngine is the Zenterio OS application that executes Lua scripts for advertisements.
The application maintains a Lua environment, containing the complete code and state of the current script. Active timers and key input is sent to this environment.

Control

Login to the STB using telnet. You will get a root shell terminal. Run ZacCmdClient to enter the command mode for applications.

The commands available for LuaEngine are:

startapp LuaEngine LuaEngine [script]
Starts LuaEngine. Optionally give a script to run. The execution will be sandboxed to the directory of the script.

sendcmd LuaEngine exit or sendcmd LuaEngine quit
Exits LuaEngine altogether.

sendcmd LuaEngine stop
Stops any timers and clears the Lua environment. Equivalent to Lua sys.stop(). Requires LuaEngine to already be running.

sendcmd LuaEngine run <filename>
Clears the current environment and loads and runs <filename>. Runs onStart(). The execution will be sandboxed to the directory of the script. Requires LuaEngine to already be running.

sendcmd LuaEngine addfile <filename>
Loads <filename> into the current environment without stopping or clearing the environment. Requires LuaEngine to already be running. Doesn't change the script sandbox (paths will still be relative to existing sandbox).

sendcmd LuaEngine eval <expression>
Evaluate a Lua expression in the current environment. Requires LuaEngine to already be running. Doesn't change the script sandbox (paths will still be relative to existing sandbox).

Telnet

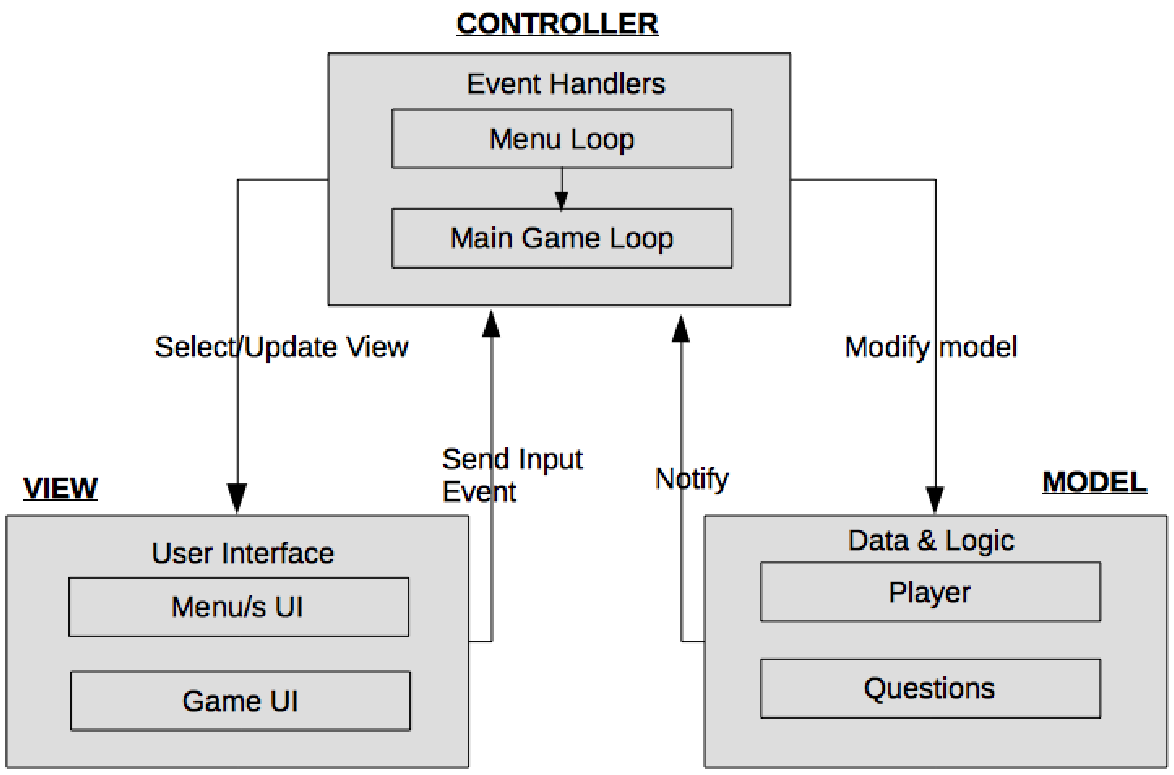
For development, the application listens for a telnet connection on port 10001 when running. The telnet session presents an interactive Lua prompt (REPL) in the environment of the current script. Globals can be inspected and changed, and the full API available to Lua scripts can be used.

The line "quit" exits the telnet session without affecting the run-time environment.

3. App 1 - Quiz Game

3.1 General Architecture

This game utilises the MVC(model-view-controller) architectural pattern. This allows the seperate handling of game entities(enemy,player) from the main game loop(controller) and the user interface(view). The communication between these 3 logical segments is shown in the diagram below.



(please see next page for descriptions)

3.2 View

This logical segment contains a folder with all in game views(questions, answers, characters, sprint lanes, finish line) as well as all complete menu views used within the game(start page, game page etc). The user input is captures by the controller which then selects the approriate view based on the specific input.

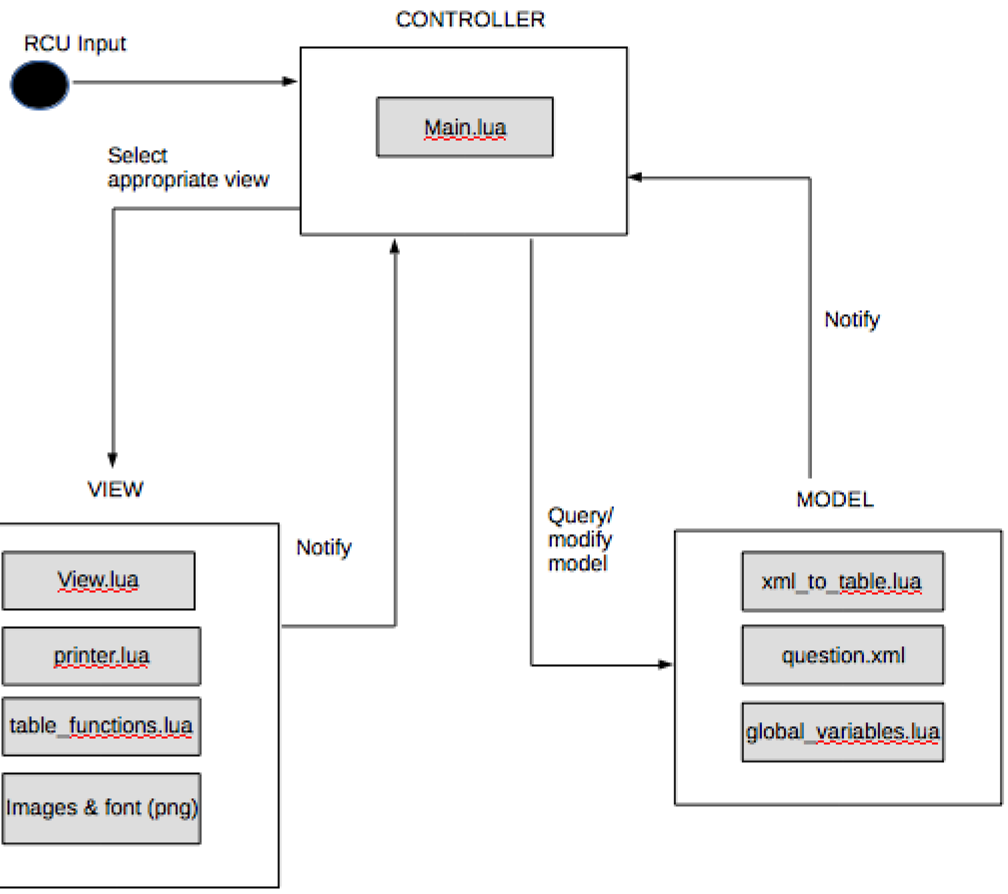
3.3 Controller

The controller is the logical segment which handles & processes events in the application, hence its the name 'controller'(all events are based off user input). There are two main loops which the controller runs: the menu loop(while the user is navigating through the menus) and the main game loop(which is invoked once the user 'starts' a game level, and runs until the end of a level is reached). The controller receives user input from the view, and uses it to run queries on the model(which contains the main data & logic of the game). For example, the user input received is the 'OK' button on the RC while the 'Start Game' button is highlighted. The controller will invoke the game loop and modify the model(based on how many players were selected & the question pack). It will then update the View to show the main game screen. Assume the next input event is player 1 submitting the answer for question 1. The controller will receive this input, modify the model(to check the answer), receive the notification from the model(right/wrong answer) and proceed to update the view.

3.4 Model

The model is the main logical segment and is responsible for housing & driving the data & logic behind the quiz game. Representations of players, questions & answers are kept in the model, and are modified based on requests from the controller. Once changes have been made, notifications are returned to the controller which will then utilise the changes to update the view.

3.5 Implementation View



3.6 File Structure & Descriptions

- /lib/
  - During development, contained all of the emulator API files. On the box, this is replaced by the Zentorio API
- /src/
  - Contains the main class files of the entire program
    - main.lua - the main game controller which creates/destroys game sessions and queries/modifies models in order to select the correct views while the game is running.
    - global_variables.lua - variables used throughout the entire program which includes players, game settings and variables used from UI positioning
    - printer.lua - module which is used to print text on the screen

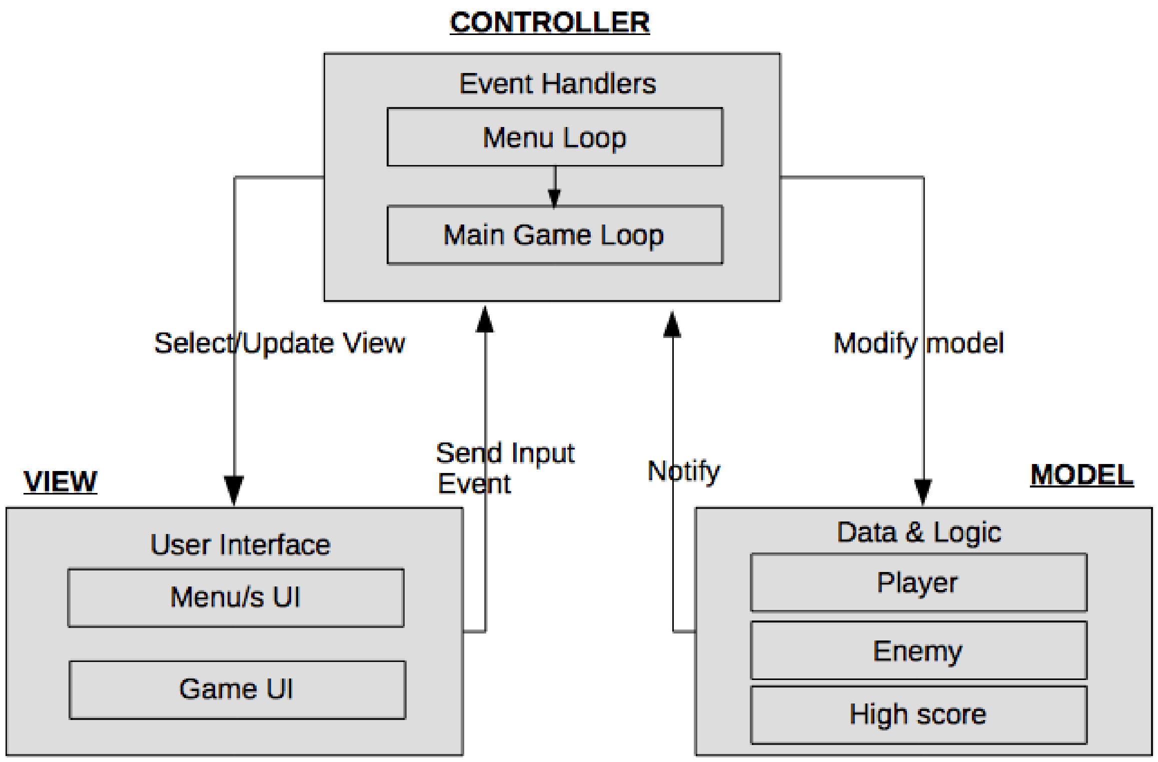
        - view.lua - Handles all funcitonality related to displaying images and surfaces on the screen.
        - table_functions.lua - Used to calculate the number of elements in a table. Used by the printer.lua module
        - xml_to_table.lua - Checks if the wanted text is on the same row as the xml tag.

- **/images/**
    - Contains all of the image which are loaded by view.lua - entire UI package of the app
- **/font/**
    - Contains a collection of png files, one for each letter of the font. These are printed by the printer.lua module
- **/test/**
    - Contains LuaUnit unit tests for each of the program files in the /src/ folder.
- **run-runit-tests.lua**
    - Launch script for running LuaUnit tests
- **start-emulator-linux.sh**
    - Starts the app in emulator on Linux (with Löve on system)
- **start-emulator-mac.sh**
    - Starts the app in emulator on Mac (with Löve on repo)
- **start-emulator-windows**
    - Starts the app in emulator on Windows (with Löve on repo)

4. App 2 - Blobs Game

4.1 General Architecture

This game utilises the MVC(model-view-controller) architectural pattern. This allows the seperate handling of game entities(enemy,player) from the main game loop(controller) and the user interface(view). The communication between these 3 logical segments is shown in the diagram below.



4.2 View

This logical segment contains a folder with all in game views(blobs,slobs,stars,background etc) as well as all complete menu views used within the game(menu, start page, game page etc). This is the portion of the application which users directly interact with. User Input(starting a game, moving in 2 dimensions etc) is received from the RCU and processed in the custom_onkey.lua file, which then sets the state of the game. The controllers contain the various bits of code which determine program path at runtime, depending on the relevant game state.

4.3 Controller

The controller is the logical segment which handles & processes events in the application, hence its the name ‘controller’(all events are based off user input). There are two main loops which the controller runs: the menu loop(while the user is navigating through the menus) and the main game loop(which is invoked once the user ‘starts’ a game level, and runs until the end of a level is reached). In addition, the game controller also requires the file collision.lua which controls collision events and the savegame.lua file which creates profiles. These 2 controllers were initially a part of game, but were split off to make the design more logical and modular.

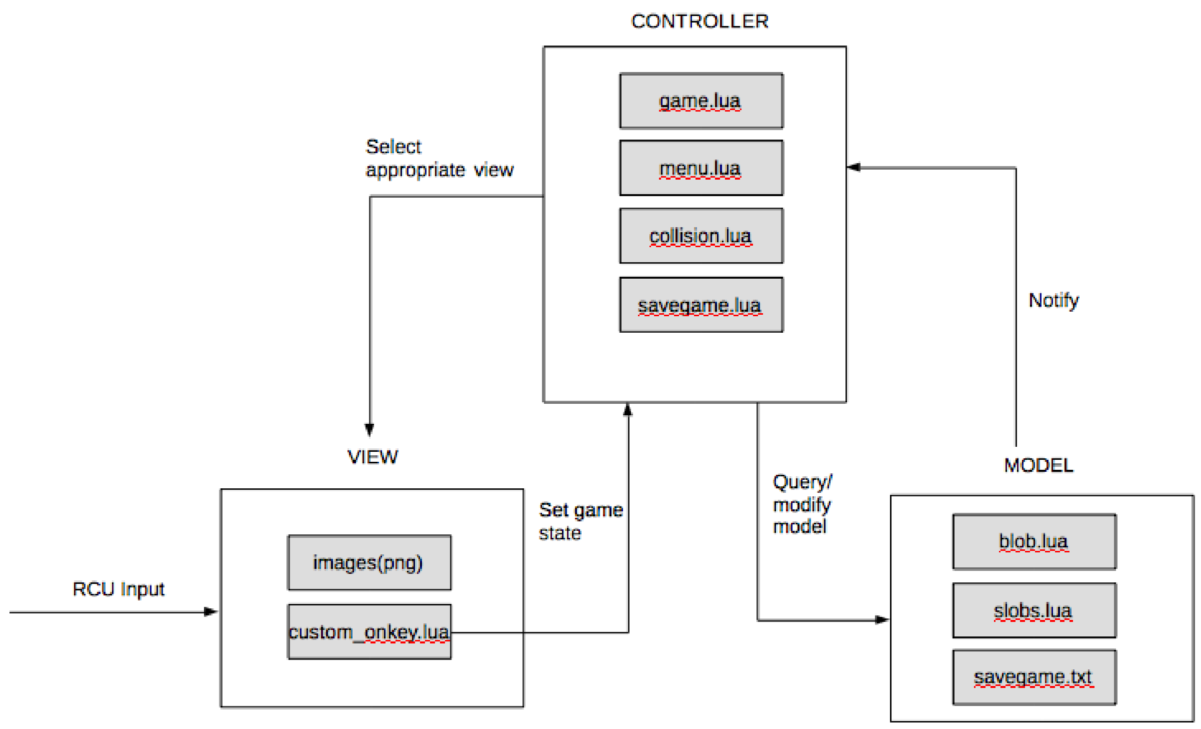
The controller receives user input from the view(specifically custom_onkey.lua), and uses it to run queries on the model(which contains the main data & logic relating to the main game elements - slobs & blobs).

For example, the user input received is the ‘UP’ button on the RC while the game_state is set to run. The controller will modify the blob.lua model to update its position. It will then update the View to represent this change in coordinates on the screen.

4.4 Model

The model is the main logical segment responsible for housing the data models & their logic. The attributes and methods of the player, enemies and high scores are modified based on requests from the controller. Once changes have been made, notifications are returned to the controller which will then utilise the changes to update the view.

4.5 Implementation View



4.5.6 File Structure & Descriptions

- /lib/
    - During development, contained all of the emulator API files. On the box, this is replaced by the Zentorio API
- /src/
    - Contains the main class files of the entire program
        - game.lua - the main game controller which creates/destroys game sessions and queries/modifies models in order to select the correct views while the game is running. Also contains global variables used throughout the whole program
        - menu.lua - controls the flow throughout all menu pages in the app(level select, avatars, tutorial, high scores)
        - collision.lua - module for detecting collisions(methods are called upon in game.lua)

        - savegame.lua - manages the save profile & load profile functions
        - blob.lua - data & logic which control & represent the various states of the blob(player)
        - slobs.lua - data & logic which control & represent the various states of the slobs (enemies)
        - savegame.txt - profiles are saved in this text file

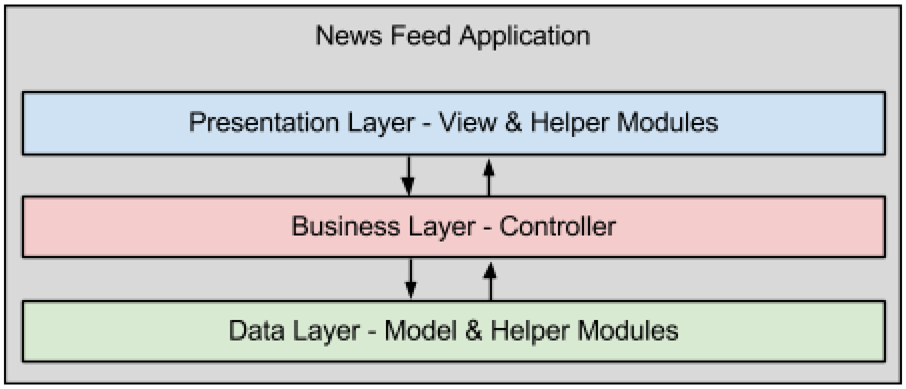
- **/images/**
    - Contains all of the UI elements of the app including menus, backgrounds, characters etc. These various elements are loaded by game.lua and the subsequent modules which are referenced
- **/test/**
    - Contains LuaUnit unit tests for each of the program files in the /src/ folder.
- **run-runit-tests.lua**
    - Launch script for running LuaUnit tests
- **start-emulator-linux.sh**
    - Starts the app in emulator on Linux (with Löve on system)
- **start-emulator-mac.sh**
    - Starts the app in emulator on Mac (with Löve on repo)
- **start-emulator-windows**
    - Starts the app in emulator on Windows (with Löve on repo)

5. App 3 - News Feed

This section describes the architecture and design of application number 3. It is a news feed application that displays news with text and images which are fetched from an online news API.

5.1 Main Architecture Pattern

The app is structured within three main layers separating presentation (user interface), application logic and data storage. Communication flow works as below. The philosophy is based on the MVC pattern, though it the controller is more tightly coupled with the view (they work in pairs).


News Feed Application
Presentation Layer - View & Helper Modules
Business Layer - Controller
Data Layer - Model & Helper Modules

5.1.2 Presentation Layer

This is where the application user interface lives, it contains everything that manages how things look.

Components and modules in this layer:

- View classes - Manages a content to be displayed on the screen
    - Main View - The main screen
    - Article View - For a single article
- Graphics module - Helper functions the views
- Font loader - Helper functions for text printer module
- Text printer - Helper function for printing text on the screen
- Log module - Writes log messages to console and/or log.txt

5.1.3 Business Layer

This is where the application logic lives, it contains everything that manages coordination, how things work, what is done when, etc.

Components and modules in this layer:

- Controller classes - Manages user input and business logic
    - Main Controller - The main screen
    - Article Controller - For a single article

5.1.4 Data Layer

This is where the application data lives, it works as the collective memory of the app.

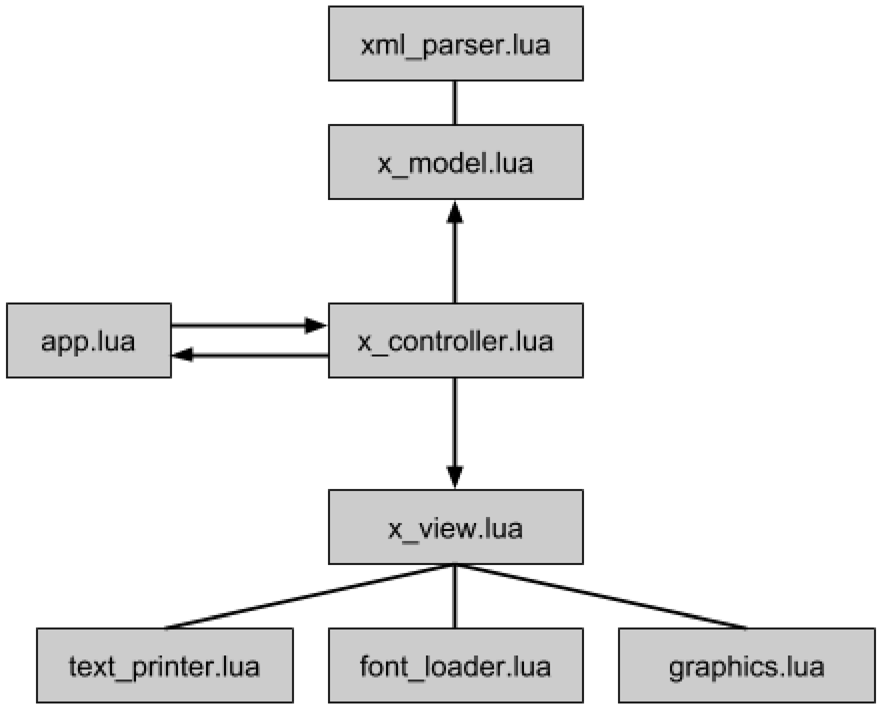
Components and modules in this layer:

- Model classes - Manages data and data logic
    - News model - For news data
- XML parser module - Parsing RSS files to Lua tables
- Download module - Function for downloading content from the news API

5.2 Implementation View

The app.lua file contains is the launcher script for the app. The role of app.lua is to initialize the app and its resources and to receive callbacks from LuaEngine (such as user input from the RCU) and forward the information to the current controller.

5.2.1 File structure and relations between classes and modules



5.2.2 File structure of the app

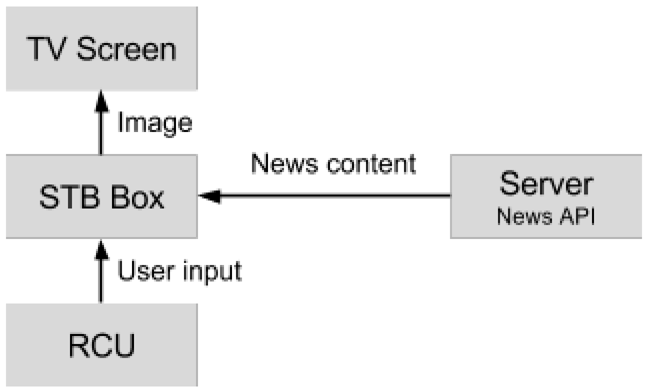
In the root folder the following items can be found:

- **/emulator/**
  - Contains modules that emulates the LuaEngine API on a development PC using the Löve 2D game engine
- **/lib/**
  - Contains helper modules files
- **/src/**
  - Contains the main class files
- **/test/**
  - Contains LuaUnit unit tests for the application
- **app.lua**
  - The main launcher script for the app
- **conf.lua**
  - Settings for the emulator
- **config.ld**
  - Configuration for LDoc generation
- **main.lua**
  - Launch script for the emulator
- **README.txt**
  - Instructions for running the app on a STB
- **run-runit-tests.lua**
  - Launch script for running LuaUnit tests

- **start-emulator-linux.sh**
    - Starts the app in emulator on Linux (with Löve on system)
- **start-emulator-mac.sh**
    - Starts the app in emulator on Mac (with Löve on repo)
- **start-emulator-windows**
    - Starts the app in emulator on Windows (with Löve on repo)

5.3 Deployment View

How the app is deployed on the STB and communicates with other hardware units.


TV Screen
Image
STB Box
News content
Server
News API
User input
RCU