Game Automation

- Vinay Chippa

Introduction

When we talk about game testing, every successful game the key lies in bringing flawless astonishing user experience. The major difficulty is ensuring this factor with the numerous platforms and various device configurations available in the market today. Why should we ensure this? The answer is that, a significant portion of revenues generated for all major marketplaces are from gaming industry. Here comes the absolute need to automate these game components to its most feasible level.

Challenges

Major challenges faced in mobile game segment are that it is fiercely competitive and short attention span from the users. The user lifetime value is heavily dependent on additional game content, collaboration with the user etc., resulting in various game updates. To support these updates there will be several application backend updates needed over a period of time. This is what will help recover the initial marketing investments.

One of the main complaints about automated UI tests is that they stop working when you make major changes to your game. There is some myth to this, most modern testing tools work directly with controls rather than being based on-screen coordinates. They are much more robust and don't break as easily as early UI testing tools.

Frameworks

When you contemplate automated testing tools, it's hard to imagine using them for something like ad-hoc testing for video games. In reality, test automation is rarely used in video game testing at all, we can think of a few used cases in which you could use it for data-driven testing if the game uses databases or for certain regression tests.

But that doesn't mean you can't use test automation for gameplay, it just might now be the most effective way since user interaction with a game is the most important aspect of game that you will need to test to. In recent past, the game automation is taking its recognition with evolution of different technologies. Many such test automation frameworks developed becoming a common platform between the game development practices and evolving technologies to automate the game.

Another implementation is the automation tools with AI logic where the tool learns how to play the game. By simulating every probable action and monitoring the results from the application backend / frontend UI and next performable action depending on the results identified in the background application logs to the server. This will help the AI logic to understand and perform the action in the front end accordingly. This way at least a most common user path is automated.

Example

This is the case with TestComplete, where the script for clicking on a button looks something like:

```
CustomerForm.ButtonOK.Click();
```

and entering text in a text control looks like

```
CustomerForm.edName.Text = 'John';
```

Breakdown: Android Test Automation Frameworks

	Robotium	Ulautomator	Espresso	Appium	Calabash
Android	Yes	Yes	Yes	Yes	Yes
iOS	No	No	No	Yes	Yes
Mobile Web	Yes (Android)	Limited to x, y clicks	No	Yes (Android & iOS)	Yes (Android)
Scripting Language	Java	Java	Java	Almost any	Ruby
Test creation tools	Testdroid Recorder	UI Automator Viewer	Hierarchy Viewer	Appium.app	CLI
Supported API Levels	All	16=>	8, 10, 15-19	All	All
Community	Contributors	Google	Google	Active	Pretty quiet

Comparison of test automation frameworks

Robotium is an Android test automation framework that fully supports native and hybrid applications. Robotium makes it easy to write powerful and robust automatic black-box UI tests for Android applications. With the support of Robotium, test case developers can write function, system and user acceptance test scenarios, spanning multiple Android activities.

Ulautomator, by Google, provides an efficient way to test UIs. It creates automated functional test cases that can be executed against apps on real Android devices and emulators. It includes a viewer, which is a GUI tool to scan and analyze the UI components of an Android app.

Espresso, by Google, is a pretty new test automation framework that got open-sourced just last year, making it available for developers and testers to hammer out their UIs. Espresso has an API that is small, predictable, and easy to learn and built on top of the Android instrumentation framework. You can quickly write concise and reliable Android UI tests with it.

Calabash is a cross-platform test automation framework for Android and iOS native and hybrid applications. Calabash's easy-to-understand syntax enables even non-technical people to create and execute automated acceptance tests for apps on both of these mobile platforms.

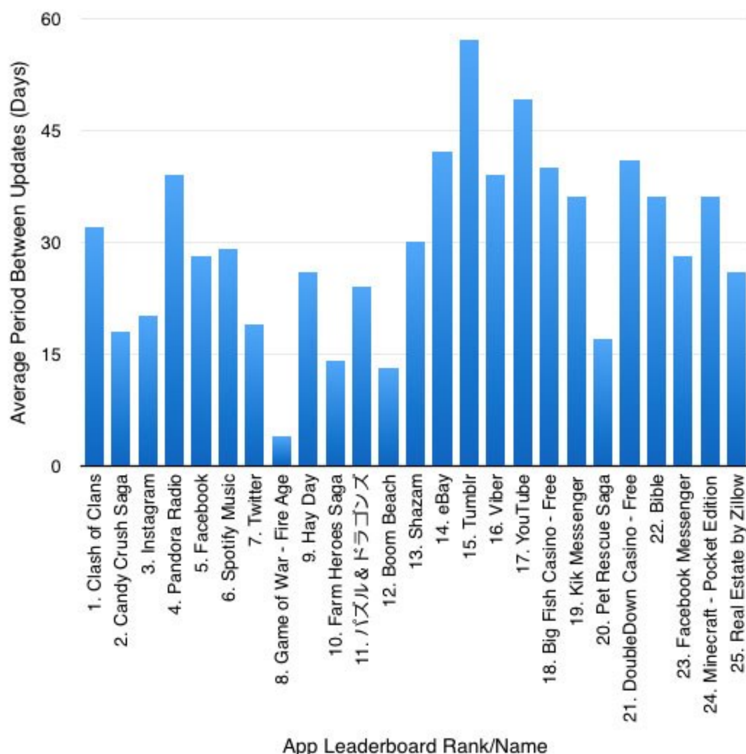
Appium, In a nutshell, Appium is a mobile test automation framework (and tool) for native, hybrid and mobile- web apps for iOS and Android. It uses JSONWireProtocol internally to interact with iOS and Android apps using Selenium's WebDriver. In fact, Appium is a pretty good choice for both apps and games because, in many cases, apps and games tend to be identical (or at least very similar) on platforms, Android and iOS - and so the same test script can be applied to both. Another significant benefit of Appium is that users can write tests using their favorite development tools, environment and programming language, such as Java, Objective-C, JavaScript, PHP, Ruby, and Python or C #, among many others.

Appium enables users to execute tests on mobile devices regardless of OS. This is possible because the Appium framework is basically a wrapper that translates Selenium's WebDriver commands to UIAutomation (iOS), UIautomator (Android, API level 17 or higher) or Selendroid (Android, API level 16 or lower) commands, depending on the device's type.

Frequent releases are the nature of the gaming market:

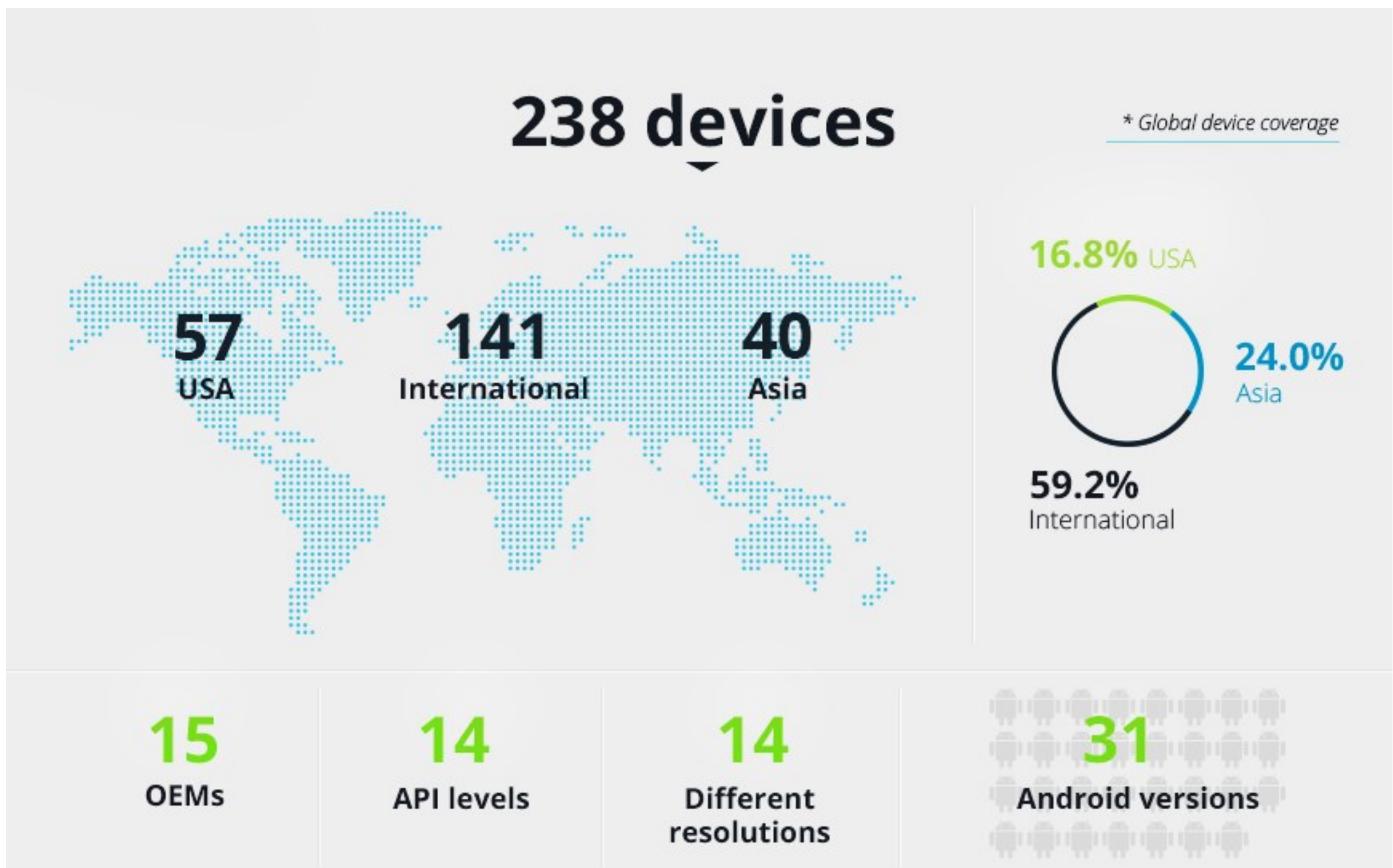
The competitive gaming market mandates frequent game releases to the market on the one hand and on the other hand games involve actual money bids it is of utmost important to ensure highest standard of testing with full coverage of all OS.

Average Number Of Days Between Their Version Updates For 25 Top iOS Apps



Games tend to be on the lower end of the spectrum, with more frequent updates. The exceptions are the casino games, which have a longer time between updates, presumably because they do not change much. Apps that are updated less frequently are social media apps like YouTube and Tumblr and messaging apps like Viber and Kik Messenger.

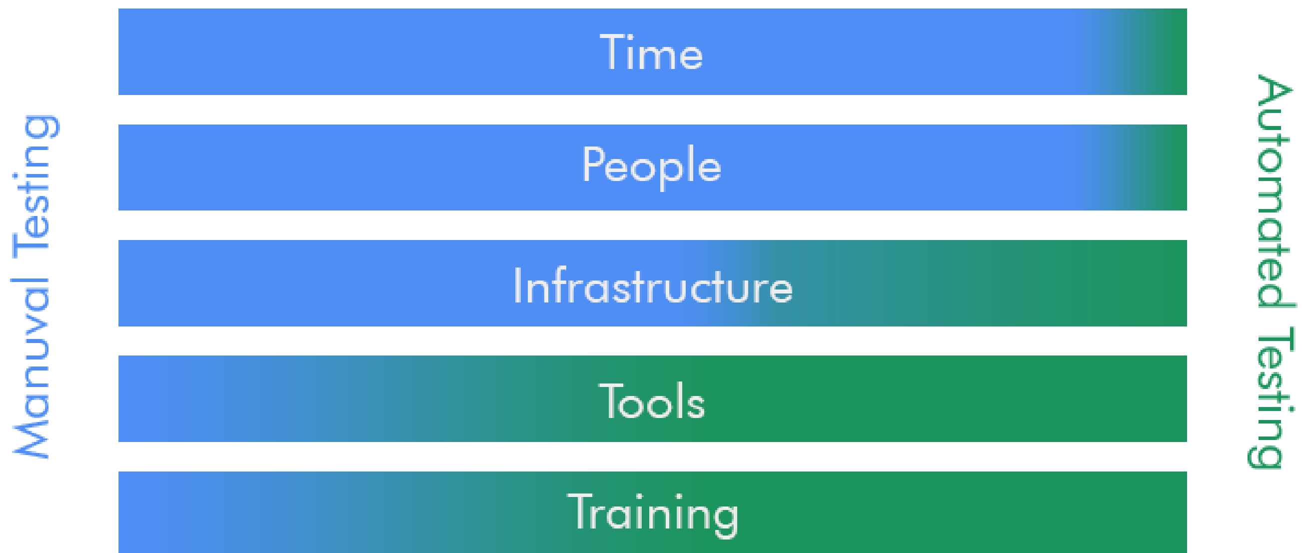
Coming to Mobile games testing, a majority of parties think that manual testing is the only effective way to move forward. The focus is towards identifying all the issues and getting it fixed before the game is published. Integrating test automation into the process focusing on every regression cycle and advancement can give considerable amount of results providing a game ready for publishing. The fact being, manual testing can't promise a 100% bug free game. This is because of many factors: such as too much effort, time, verification and validation are needed. This happens only in the superficial part of the game leaving the application backend untested. To debug the backend components, you will need a team of programmers. This is where test automation can be implemented. It can deliver 24/7 without any of the manual efforts. Test automation can deliver better test coverage and test results by getting deep into the game application ensuring stability, compatibility (devices) and so on.



Above image brings us the information on how many different devices are available globally to perform adequate test coverage.

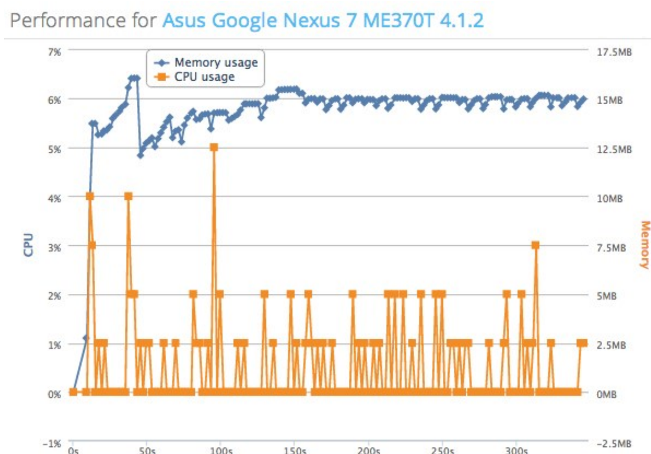
Costs, Assets and Time to market

The test automation for mobile has been considered a critical factor for big mobile app companies but for some reason it is often thought to be too expensive or difficult to adopt for smaller companies. Probably due to historical reasons the cost factor is the first consideration when deciding whether company wants to use automation or stick with manual testing in their development projects. Regardless of if you select manual and automated testing, you'll need the following assets and resources – and those will cost you money: Time, People, Infrastructure, Tools, and Training. Challenges



The primary challenge to automate testing for these games is the usage of OpenGL or ActiveX by passing the OS level services. This leads to a problem - all the native mobile test automation frameworks become useless with Mobile games. This limits us to use only the X&Y clicks without much feedback or validation about the internal state of the game.

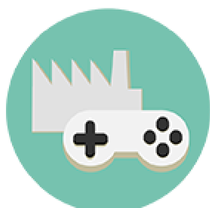
Secondary challenge is performance, which is the key factor for great user experience and can only be observed on real hardware. Frame rates do matter a lot, and the richness in graphics on different mobile devices counts. Massive games come with heavy binaries reaching up to 3GB which demands for heavy memory, GPU, Battery and CPU. All the above needed to be considered for actual user experience. It should not to be affected due to different hardware configurations. Majority of the games are integrated with different hardware components like sensors, mic, speakers, camera, GPS etc. The consistency of gameplay largely varies based on the quality of these components in different mobile devices available in the market.



Performance of an application using the device hardware to ensure the stability and latency w.r.t its active state

For the past few years we have been working with major game publishers, testing their games on various physical devices with wide variety of tools, practices, methods resulting in huge a framework. Regardless of the game engine, the game developers use a simple form of image recognition method. This is as simple as capturing a screenshot and evaluating it automatically or manually while it can be fully automated gameplay. The gameplay can be driven by the test script and output data stored as logs for further analysis. By combining these 2 approaches, identifying certain graphical elements from real-time screenshots and comparing those to pre-set graphical assets and progressing through the game using the test script.

Black box testing approach using different test automation frameworks is evolving in gaming industry. For example; Appium which is a cross-platform framework for android and iOS devices gets the work done really well. The reason being; the games on both these platforms work the same way. Appium bridges the gap between image recognition and handling assets within the game. Appium is capable of many things - from installing the builds to performing necessary actions to managing the test automation sessions by becoming high level interface between the test scripts and game. Combining this with OpenCV makes our work even easier by accessing the game from outside and validating the screen buffer with its image library directing the Appium script to perform action at X&Y co-ordinates. It can recognize images that are stretched or at an angle. The Idea is to simplify writing the scripts, which basically performs 2 tasks i.e. clipping the reference images and define the action or click when the match is found in the screenshot. Complexity of scripts is reduced to a major extent by this implementation.



Take your game to the next level with our Game Testing Services

Click Here

Example: We have used Appium to automate the most popular slots game

In this example we are using the most popular slots game. It's a fantastic game and I bet many of you have played it so you should be pretty familiar how the game looks and so on. We're also going to use Appium as a selected test automation framework to basic clicking-through Basic game play of this game.

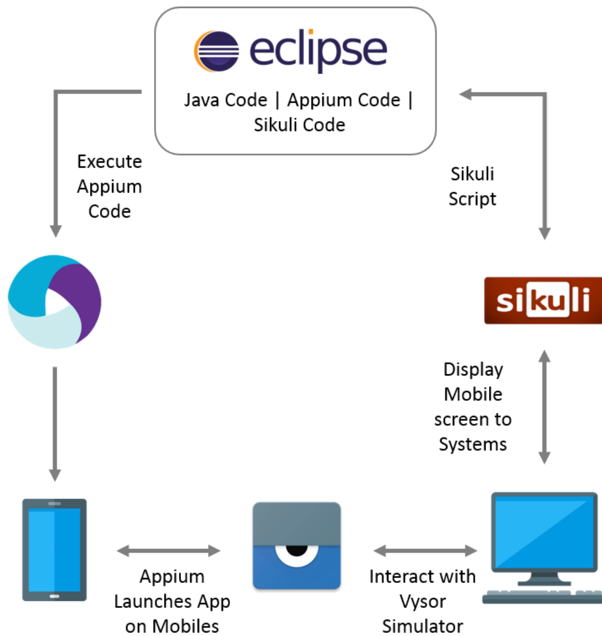
```
##
## Example script that tests the basic game play
##
## Works on multiple platforms irrespective any device attached to the machine.
##
//Launching the application
public class PropertyFile
{
WebDriver driver;
public void setup() throws FindFailed, InterruptedException, IOException {
File app = new File("E:/apks/slots.apk");
DesiredCapabilities capabilities = new DesiredCapabilities();
capabilities.setCapability("noReset", "true");
capabilities.setCapability("fullReset", "false");
capabilities.setCapability("AutomationName", "Appium");
capabilities.setCapability("platformName", "Android");
capabilities.setCapability("platformVersion", "6.0.1");
capabilities.setCapability("deviceName", "Galaxy S6");
capabilities.setCapability("newCommandTimeout", 2000);
capabilities.setCapability("app", app);
capabilities.setCapability("appPackage", "com.package_name");
capabilities.setCapability("appActivity", "com.activity_name");
driver = new AndroidDriver(new URL("http://127.0.0.1:4723/wd/hub"), capabilities);
}
}
```

```
// Lobby 1 Settings button and build check
//@Test(priority = 1)
public void lobby1Testcase() throws InterruptedException, FindFailed, IOException {
Screen sc = new Screen();
gc.motdHandle();
// Waiting for Settings Button on Home Screen to click
sc.wait("D:/ project/New Images/Home_screen/Home_setting.png", waitTime);
Assert.assertTrue(exists("D:/ project/New Images/Home_screen/Home_setting.png", sc),
"Home_setting.png not found");
// Clicking Settings Button
sc.find("D:/ project/New Images/Home_screen/Home_setting.png").click();
// Waiting for Settings Close Button on Home Screen to click
sc.wait("D:/ project/New Images/Home_screen/Setting_Closing.png", waitTime);
Assert.assertTrue(exists("D:/ project/New Images/Home_screen/Setting_Closing.png", sc),
"Setting_Closing.png not found");
// Clicking Settings Close Button
sc.find("D:/ project/New Images/Home_screen/Setting_Closing.png").click();
// Waiting for Settings Button on Home Screen to click
sc.wait("D:/ project/New Images/Home_screen/Home_setting.png", waitTime).click();
Assert.assertTrue(exists("D:/ project/New Images/Home_screen/Home_setting.png", sc),
"Home_setting.png not found");
// Clicking Settings Button
sc.find("D:/ project/New Images/Home_screen/Home_setting.png").click();
Thread.sleep(3000);
// Waiting for Sound Option
Settings st = new Settings();
st.MoveMouseDelay = 1;
sc.wait("D:/ project/New Images/Home_screen/Soun_click.png", waitTime);
sc.dragDrop("D:/ project/New Images/Home_screen/Soun_click.png",
"D:/ project/New Images/Home_screen/dragDropDest.png");
Thread.sleep(2000);
```

```
// Waiting for about option and click
st.MoveMouseDelay = 0;
sc.wait("D:/ project/New Images/Home_screen/home_about.png", waitTime);
Assert.assertTrue(exists("D:/ project/New Images/Home_screen/home_about.png", sc),
"home_about.png not found");
sc.find("D:/ project/New Images/Home_screen/home_about.png").click();
Thread.sleep(5000);
// Waiting for about page to load
sc.wait("D:/ project/New Images/Home_screen/about.png", waitTime);
Assert.assertTrue(exists("D:/ project/New Images/Home_screen/about.png", sc), "about.png not
found");
// Taking screen short for build
gc.takeSnapBuild();
// Waiting for about close option and click
sc.wait("D:/ project/New Images/Home_screen/about_close.png", waitTime).click();
}
// Room 1 checks
//@Test(priority = 2)
public void room1TestCase() throws InterruptedException, FindFailed, IOException {
Screen sc = new Screen();
sc.wait("D:/ project/New Images/Room1/room1_reference.png", waitTime);
// Setting room1 region
Location room1RegionLoc = sc.find("D:/ project/New Images/Room1/
room1_reference.png").getCenter();
Region room1Region = sc.newRegion(room1RegionLoc, 600, 600);
// Waiting for home page to load
sc.wait("D:/ project/New Images/Room1/Room1_refrence.png", waitTime);
room1Region.wait("D:/ project/New Images/Room1/Room1_entry", waitTime);
// Clicking room 1 entry button
room1Region.find("D:/ project/New Images/Room1/Room1_entry").click();
Thread.sleep(2000);
// handling addon
gc.addonHandle();
```

```
// Spinning inside room
sc.wait("D:/ project/New Images/Room1/Spin_room1.png", waitTime);
Assert.assertTrue(exists("D:/ project/New Images/Room1/Spin_room1.png", sc),
"Spin_room1.png not found");
sc.find("D:/ project/New Images/Room1/Spin_room1.png").click();
Thread.sleep(5000);
// gc.progresion();
// Room1 settings
sc.wait("D:/ project/New Images/Room1/Room1_setting.png", waitTime);
Assert.assertTrue(exists("D:/ project/New Images/Room1/Room1_setting.png", sc),
"Room1_setting.png not found");
sc.find("D:/ project/New Images/Room1/Room1_setting.png").click();
// Room1 paytable
sc.wait("D:/ project/New Images/Room1/Paytable_room1.png", waitTime);
Assert.assertTrue(exists("D:/ project/New Images/Room1/Paytable_room1.png", sc),
"Paytable_room1.png not found");
sc.find("D:/ project/New Images/Room1/Paytable_room1.png").right(2000)
.find("D:/ project/New Images/Room1/click_paytable.png").click();
Thread.sleep(3000);
// room1 settings close
sc.wait("D:/ project/New Images/Room1/Room1_setting_close.png", waitTime);
Assert.assertTrue(exists("D:/ project/New Images/Room1/Room1_setting_close.png", sc),
"Room1_setting_close.png not found");
sc.find("D:/ project/New Images/Room1/Room1_setting_close.png").click();
// return to lobby
sc.wait("D:/ project/New Images/Room1/Return_home.png", waitTime);
Assert.assertTrue(exists("D:/ project/New Images/Room1/Return_home.png", sc),
"Return_home.png not found");
sc.find("D:/ project/New Images/Room1/Return_home.png").click();
Thread.sleep(5000);
gc.motdHandle();
}
```

Architecture used



Tasks handled by the Framework:

- Screen Capturing of 'About' dialog box for build version check
- Screen Capture for test fail instances
- One level-up Progression
- Handling Add-On's dialogs
- Handling pop-up dialogs
    - Capturing the pop-up dialogs appeared during the game progression for manual reference after the test execution is completed. This is to validate desired pop-up dialogs are triggered during the gameplay.
- Reporting the test NG report Via email

Using Image Recognition

What we just went through in Appium + Sikuli example was the basic image recognition flow for enabling mobile game to be tested on real devices, regardless of any OS platform (Android and iOS). Here we had a limitation to recognize the elements as the application was developed in flash. In certain apps and majority of games, you will not be able to access the elements on the screen. When you load the app, and open the Appium GUI, you will be presented with a single Android View, rather than a layout with elements.

To work around this problem, I decided to implement OpenCV image recognition to enable finding elements on the screen using screenshots. To achieve that, we used SikuliX API. Due to the nature of SikuliX API, the flow isn't as straightforward as one would expect. You can't do a direct comparison against the device; you need to run a remote comparison against a screenshot of the device using a simulator.

Code Flow

The flow in above code is as follows:

- Take a screenshot of the device
- Compare the image of the element you want to find to the screenshot of the device
- If match is found, return coordinates of the center of the element
- Use those coordinates to tap on the screen
- The main methods in the OCR class:
    - clickByImage - Main method you should be using. It allows you to find and tap on the element on the screen by passing in the path to the screenshot of the element. It aggregates all the convenience method into a single, easy to use method.
    - takeScreenshot - convenience method that takes a screenshot and returns a Buffered Image for further processing.
    - getCoords - requires screenshot as buffered Image and the path to the image of an element we're looking for. If match is found, the coordinates are returned in a Point2D object.
    - elementExists - returns true if element is found on the screen.
    - waitUntillImageExists - Explicit wait using Image Recognition. Waits for specified duration until a match for the specified image is found.

Limitations:

Cannot validate the number format in the screen

- Cannot validate particular text
- Not possible to compare if there are any minor changes in preloaded screenshots taken from Sikuli
- Test data such as image objects need to be preloaded resulting in rework in any design UI design changes
- Limited to single user path or gameplay but not able to simulate all the probable user paths
- Multiple tests on different devices at a time will not be possible while using image recognition method, this is possible only if we will be able to identify the elements within the UI using selenium grid

Current areas of automation being viable

Game automation has entered gaming industry at a very few stages of game testing because of its limitations in simulating all the possible user paths. It's being helpful in performing basic level of testing like smoke testing, build verification tests, basic level progression, MMO grinding, support and maintenance, load testing etc. Because of the cost incurred in framework maintenance, as and when there is a design change, it has taken more time to develop the framework instead of manual testing. Game automation has taken remarkable changes and improving the contributions with evolving technology.

At the end of the day though, no automation framework is as adaptable as a real tester, so it usually requires substantial buy-in from development teams. The earlier you can start working with a specific team, the better the results you'll likely see. With each success you gain traction to change the culture. In some cases, Sony is working with developers to design in-test frameworks before development even starts.

Try to resist the temptation to start on only high-level integration tests such as UI or screenshot comparisons. These may be conceptually easy to understand and author, but they are also very fragile and not easily scalable. This could hold you back while you just lose time and money on maintenance.

Future of game automation

The world's oldest board game still has a few moves to play. Go, a game of strategy and instinct considered more difficult to master than chess, was created roughly in the same era as the written word. The game is uniquely human - at least, it used to be. Last year, a computer program called AlphaGo defeated an internationally ranked professional player.

Between the physical and mental spaces, there is another reality in need of double control. Augmented reality, or artificial reality, bridges the gap of actuality and imagination. Pokémon GO is a prime example, as people navigate the physical world to find fictional creatures with only experience as a guide. The parameters and goals shift with each new exposure

AI researchers at DeepMind (acquired by Google in 2014) have developed an intelligent software engine that can automatically learn to play and finish all of the Atari games by itself. You just need to feed-in the game and the rest will be done by the system. The machine built by the DeepMind team recently won one of the toughest Chinese games, AlphaGO, against world champion, Lee Sedol, by a score of 4-1. In another breakthrough, researchers from the University of Texas at Arlington (UTA) created a game engine that can empower a computer to play any game, for example Super Mario, without any human interference. The paper was published in the MIT Journal.

Nevertheless, the potential benefits of automation are huge. For example, Unity's ability to develop and release across so many platforms is almost solely attributable to automation. It enables them to more easily verify features across 15 to 20 platforms with a massive number of test cases run on every single commit.

Quite simply, automation is the latest, and potentially greatest, tool in the QA arsenal. Any mid-sized developer would be well advised to invest time in it as soon as possible.