크로스 플랫폼을 지원하는 GUI 프레임워크 구조 설계

김민수○ 이승형 이성원
경희대학교 컴퓨터공학과
betas@khu.ac.kr, shlee7@khu.ac.kr, drsungwon@khu.ac.kr

GUI Framework Structure Design for Cross-Platform

Min-Su Kim○ Seung-Hyung Lee, Sung-Won Lee
Department of Computer Engineering, Kyung Hee University

요 약

본 논문에서는 여러 기기들의 등장으로 분산된 Front-End GUI 개발환경을 하나로 통합하여 구현할 수 있는 Framework의 구조와 구현방법을 제안한다. 제안하는 프레임워크 에서는 1) 공통 컴포넌트를 이용한 Layout 정의, 2) 컴포넌트와 플랫폼을 이어주는 Native Adapter 정의, 3) 최종적으로 Platform 환경과 밀접하게 구동될 수 있도록 Layout Interface를 정의하였다. 최종적으로 기존 Hybrid방식의 앱 개발 방법의 단점인 Internal WebView를 이용하여 구동되어 전체적 성능이 하락한다는 단점을 Compiled 방식으로 보완하여 성능 향상을 꾀하고, Cross-Platform 개발이 용이하다는 기존의 장점을 합하고자 한다.

1. 서 론

최근 SI환경에서 가장 흔하게 일어나고 있는 개발은 Android, iOS와 같은 스마트폰 어플리케이션의 개발이나, HTML과 같은 웹 개발 등, 사용자에 밀접한 서비스 또는 제품을 구현하는 개발 이라고 할 수 있다. 그중 Back-End 개발은, 구현될 클라이언트의 플랫폼과는 독립적으로, 하나의 프로그래밍 환경으로 구성할 수 있지만, 대다수의 Front-End GUI 개발은 각각의 플랫폼에 종속적으로 구성할 수 밖 에 없다. 따라서 각각의 플랫폼에 맞는 개발자를 기업들이 채용하는 것은 중소기업이나 막 창업을 시작하는 스타트업들에게는 큰 부담으로 작용한다.

본 연구에서는 이런 개발환경을 하나로 통합하기 위해 1) Fragmentation된 Front-End GUI 개발 방법을 하나로 묶을 수 있는 개발방법을 제안하고, 2) 이를 구현 할 수 있는 System Structure를 구성하며, 3. 이를 구체적으로 각각의 플랫폼에 이식 할 수 있는 방법을 제안하고자 한다.

2. 관련연구

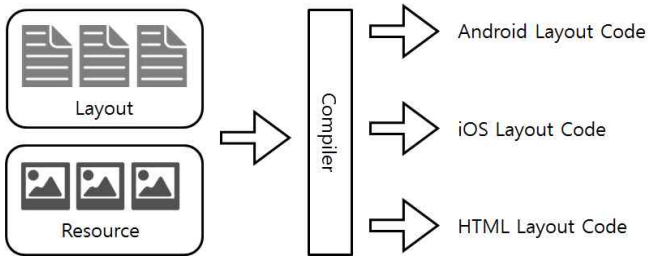
하이브리드 앱 구현 방식은 HTML을 이용하여 모바일 웹을 구현하고, 이를 각각 모바일 기기에 내장된 Internal Web View를 사용하여 기기에서 보여준다. 만약 기기와 상호작용(센서, 카메라 등)이 필요할 때에는 Javascript Interface를 이용하여 제어하는 방식으로 동작한다.

이 방식은 기존 모바일 웹의 코드로 Android, iOS등 환경에 구애받지 않고 똑같이 UI를 사용자에게 보여줄 수 있다는 장점이 있으나, 구동속도가 Internal WebView에 의하여 좌우되고, Image Resource등을 매번 네트워크를 통해 받아오기 때문에 완벽한 로딩에까지 걸리는 시간이 Native App보다는 느리다는 단점이 있다.

이러한 단점을 해결하기 위해 PhoneGap[1]과 Cordova[2]라는 각각 Adobe와 Apache에서 지원하는 크로스 플랫폼 앱 개발 환경이 등장하여 리소스 로딩을 매번 해야 한다는 단점을 "HTML5, CSS, Javascript"로 구현된 웹페이지를 디바이스에 직접 내장시키고, 해당 플랫폼에서 Internal WebView방식으로 구동 하는 방법으로 해결하였고, 각 플랫폼별 특이기능들은 미리 Javascript API형식의 라이브러리를 통해서 구현하여 기기 종속적이지 않는 코드를 작성하는 데에 초점이 맞춰져 있다.[3]

3. Native Emulation방식의 GUI 프레임워크 시스템 제안

본 논문에서는 위의 방식들과는 달리 Hybrid 방식에서 Internal WebView로 인하여 발생하는 성능 문제를 해결하고자 [그림 1]과 같이 공통된 레이아웃 코드와 리소스 들을 직접 컴파일 하여 Native Code로 변환 하는 방식으로 해결하고자 하였다.



[그림 1] 사용자가 정의한 리소스를
직접 타겟 레이아웃으로 컴파일 하는 방식의 개념도

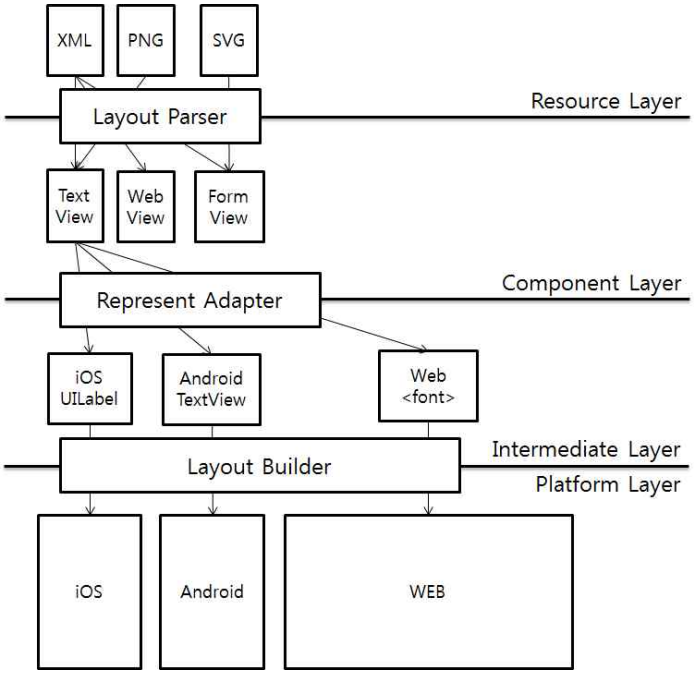
구현하고자 하는 시스템의 각 부분은 [그림 2]와 같은 몇 개의 Layer로 구분 할 수 있다.

Resource Layer 에서는 개발자가 정의한 XML으로 된 Layout과 Javascript로 구현된 View의 콜백 함수 구현(Constructor, Getter, Setter, 조작을 위한 Method 등), 비트맵, 벡터 이미지 리소스와 같은 내용을 파싱하고, 타겟 환경에 맞게끔 변환 하는 데에 목적이 있다.

Component Layer에서는 사용자가 구현한 Custom View들을 프레임워크에서 기본적으로 지원하는 Primitive View로 변환하고, 리소스 참조를 수행하여 중간코드로 바꾸어 주는 데에 목적이 있다.

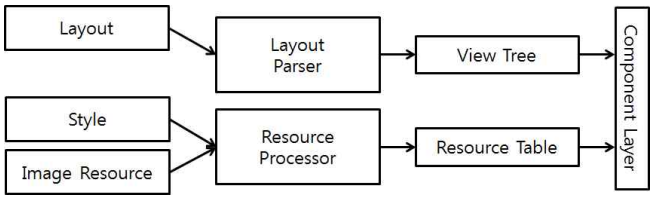
Intermediate Layer에서는 중간코드를 각 디바이스의 환경에 맞는 코드로 대응시켜 변환 하는 데에 목적이 있다.

최종적으로 Platform Layer에서는 컴파일 된 코드를 해당 디바이스 읽어와 Native View로 구현하기 위한 Native Library를 제공하는 데에 목적이 있다.



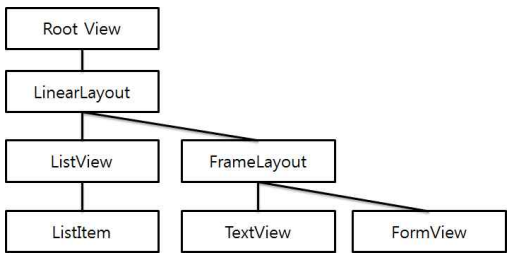
[그림 2] 전체 시스템 구성도

3.1 Resource Layer



[그림 3] Resource Layer Flow

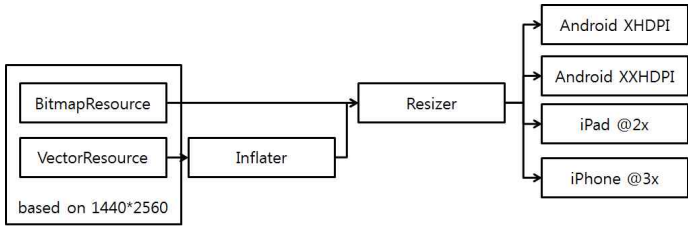
Resource Layer 에서는 [그림 3]과 같이 각 화면 또는 컴포넌트의 Layout을 구조적 문법인 XML 포맷을 이용하여 주종 관계와 어트리뷰트를 기술하면 Layout Parser가 [그림 4]와 같은 View Tree구조로 변환하고, 각 플랫폼에서 사용될 리소스를 참조하기 용이하도록 기기에 맞도록 변환하고, 참조테이블인 "Resource Table"로 변환하는 역할을 수행한다.



[그림 4] View Tree 구조 예시

3.1.1 Resource Processor

GUI를 구현하기 위해서는 필수적으로 1) 다른 레이아웃을 참조하거나, 2) 다른 View를 참조하거나, 3) 다른 이미지 또는 벡터를 참조하거나, 4) 사전 정의된 색상, 스타일, 크기 등을 참조해야 하는 상황이 발생한다. 따라서 Layout Parsing 과정에 앞서서 우선 이미지와 스타일을 컴파일 하는 과정을 거친다.



[그림 5] 이미지 프로세싱 과정

벡터방식의 이미지는 단일 사이즈로 여러 타겟 사이즈의 비트맵을 많이 만들어낼 수 있다는 장점이 있지만, 타겟 디바이스에 따라 지원이 될 수도, 안 될 수도 있다. 따라서 [그림 5]와 같이 공통적인 지원을 위해서 벡터 이미지를 각각 타겟 환경 Metric에 맞는 벡터로 1차적으로 변환하고 이미지를 생성하며 (Vector Inflate), 비트맵 이미지들도 각각의 Metrics에 맞도록 변환한다.(Mipmapping)

그 이후 XML을 바탕으로 정의된 스타일과 이미지의 메타데이터를 묶어서 [표 1]과 같은 구조 Resource Table로 정의하여, 이후 Layout Parser에게 제공한다.

구분	ID	값
Image	img_bg_info	$path/bg1.png
	btn_ok	$path/btn/1.png
Style	common_btn	사용자 정의 style 객체

[표 1] Resource Table 예시

3.1.2 Layout Parsing

레이아웃은 XML로 구현되어 이전에 설명되었던 [그림 4]와 같이 각각의 View끼리의 포함 관계를 정의할 수 있다.

또한, 각각의 View 내부에 XML Attribute로 Size, Margin, Padding, Style, Color, Value와 같은 Style Attributes를 정의할 수 있는데, 이때 다른 리소스나 레이아웃의 ID를 참조하게 되는 경우 선후관계에 의하여 참조하지 못하는 상황이 발생 할 수 있으므로, 우선 string 형식으로 객체에 저장 해놓고, 추후 Component Layer에서 [표 1]의 Resource Table에서 참조하여 구성 하게 된다.

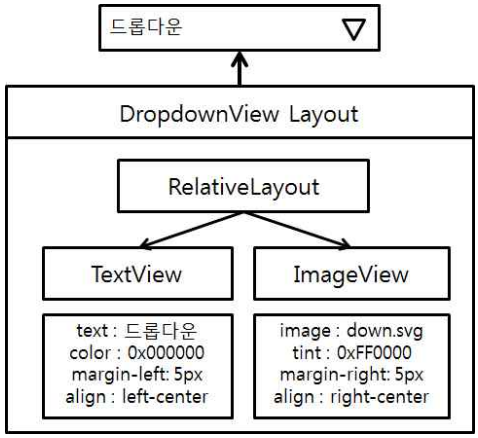
3.2 Component Layer

Layout Parser로부터 전달 된 "View-Tree"의 아이템을 깊이 우선 탐색을 통해 순회하면서 [표 2]에서 정의한 Primitive View로만 구성 될 수 있도록 View Tree를 생성하고, 이와 동시에 스타일과 컬러, 크기와 같은 사전 정의된 value들을 직접 어트리뷰트에 대입하여 파싱을 용이하게 만든다.

LinearLayout	TextView
RelativeLayout	TextFieldView
FrameLayout	ImageView
PageViewLayout	Button
ListLayout	DecorView
TabLayout	ListItemView
GridLayout	TabBarItemView

[표 2] Primitive View 목록 예시

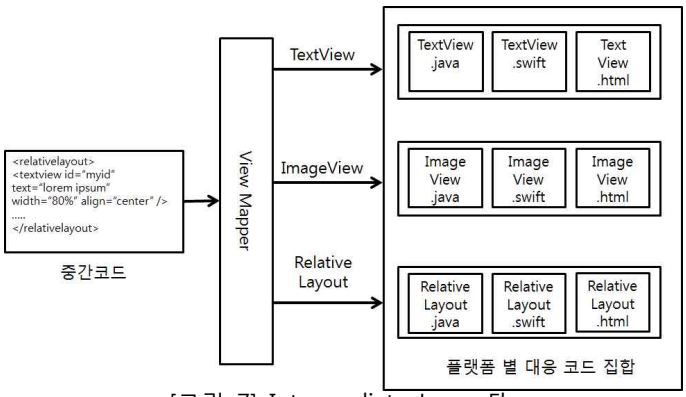
예를 들어 [그림 6]에서 정의한 바와 같이 Dropdown Button과 같은 커스텀 컴포넌트는 Primitive View를 이용하여 정의된 또 다른 레이아웃이다.



[그림 6] 드롭다운 버튼 처리 예제

최종적으로 위처럼 모든 레이아웃을 파싱하여 Primitive View로 구성하고 나면 모든 플랫폼에서 공통으로 사용할 수 있는 중간코드가 생성된다.

3.3 Intermediate Layer



[그림 7] Intermediate Layer Flow

Intermediate Layer에서는 작성된 중간코드를 읽어 들여서 1) 각각의 플랫폼에 구현되어 있는 기본 View 또는, 사용자 정의 View로 매핑하고, 2) 어트리뷰트로 부터 해당 View 객체의 값들을 설정한다. 3) 또한 Javascript로 정의된 View의 Interface함수나 생성자등을 해당 View객체에서 사용할 수 있도록 Binding 한다.

본 프레임워크에서 지원하는 Interface 함수는 [표 3]과 같다.

함수이름	설명
init(attr)	생성시 호출
click(view)	특정뷰의 클릭시 호출
delete()	소멸시 호출

[표 3] 기본 인터페이스 함수 예시

예를 들어 Dropdown Button에는 생성자에서 입력받은 텍스트와 드롭다운 이미지에 맞는 View구성이 필요한데, 이에 대한 내용을 Javascript에서 정의하면 실제 뷰가 구현되면서 해당 자바스크립트를 프레임워크 내부에 내장된 Javascript Engine으로 호출하게 된다.

3.4 Platform Layer

이 레이어 에서는 최종 완성된 컴파일 코드를 위하여 필요한 기능들을 해당 디바이스 환경에 맞는 코드로 구현하여 라이브러리로 제공하는 단계 이다.

이 과정 중 View를 직접 화면에 로드할 때, 또는 Interface 함수를 통해 사용자가 직접 구현한 View의 Javascript를 자체에 내장된 Javascript Engine을 통해 구동하여 결과 값을 제공 또는 처리해준다. 예를 들어 View가 로드되었을 때 특정 Javascript를 실행하여 View를 로드할시 동적으로 구성 할 수 있게끔 해주며, View의 특정 Property에 접근 할 수 있도록 Getter와 Setter를 제공해주기 위하여 별도 내장된 Javascript Engine을 통해 Bridging을 수행한다.

4. 결론

본 연구는 기존 Hybrid 개발 방식의 단점인 레이아웃 구현 과정에서 발생하는 성능이슈를 별도로 정의한 Primitive View를 이용하여 정의하고, 이를 컴파일 하여 타겟 코드로 직접 변환하는 데에 목적이 있다고 할 수 있다.

따라서 이에 필요한 Resource Layer, Component Layer, Intermediate Layer, Platform Layer 총 4가지의 레이어로 구분하고 각각 Layout Parser, Represent Adapter, Layout Builder의 역할을 하는 프로그램의 구조를 제안하였다.

본 연구를 통해 실제 1개의 코드로 크로스 플랫폼에서 구동이 가능하다면 기존 Native Mobile개발 (특히 Android)에 익숙한 개발자들이 iOS나 Web 기반의 GUI를 짜는 데에 도움을 얻을 수 있을 것으로 생각하며, 더 나아가 기업들이 이러한 구현방식을 채택함으로써 불필요한 자원의 낭비를 줄일 것으로 기대한다.

하지만 본 연구에서 제안한 방식은 PhoneGap, Cordova와는 달리 구체적인 사용자 인터렉션이나 장면전환, 네트워크 통신, 로직 등 을 구현하기 위해 여전히 플랫폼에 종속적인 Native Code를 사용해야 한다는 단점이 존재한다. 따라서 본 방법을 적용 하더라도 레이아웃에서의 성능향상은 기대할 수 있지만, 일정 부분은 각 플랫폼에 맞는 소스코드로 작성되어야 할 필요성이 분명히 있다. 따라서 추후 본 프레임워크를 구현하는 단계에서 해당 부분에 대한 해결 방안을 연구할 계획이다.

* "본 연구는 미래창조과학부 및 정보통신기술진흥센터(IITP)에서 지원하는 서울어코드활성화지원사업의 연구결과로 수행되었음" (R0613-16-1203)

5. 참고문헌

[1] Adobe PhoneGap (http://phonegap.com/)
[2] Apache Cordova™ (https://cordova.apache.org/)
[3] Pierre, Niyigena Jean, and Mukiza Octavien, "Review of PhoneGap APIs Accessing the Native Mobile Platform APIs.", *Lecture Notes on Software Engineering 4.1,* 2016