**Vision Blocks Documentation**

MIT Media Lab: Camera Culture Group

Nicholas Matuzita Mizoguchi

**Abstract**

[Type the abstract of the document here. The abstract is typically a short summary of the contents of the document.]

Table of Contents

Introduction 3

Project: vblocks-base 4

EngineActivity 4

Module 4

ModuleCollection 4

Sample 5

CommitableView 5

Project: Blocks 6

VBlocksApplication 6

MainActivity 6

PreviewActivity 7

PreviewMenu 7

MainView 7

BlockView 7

BlockGroupView 7

Creating a Block 8

Getting Started 8

Implementing methods 8

Execute 8

getName and getModuleName 8

getModuleTouchListener 9

onHandleIntent 9

Adding to the Menu 9

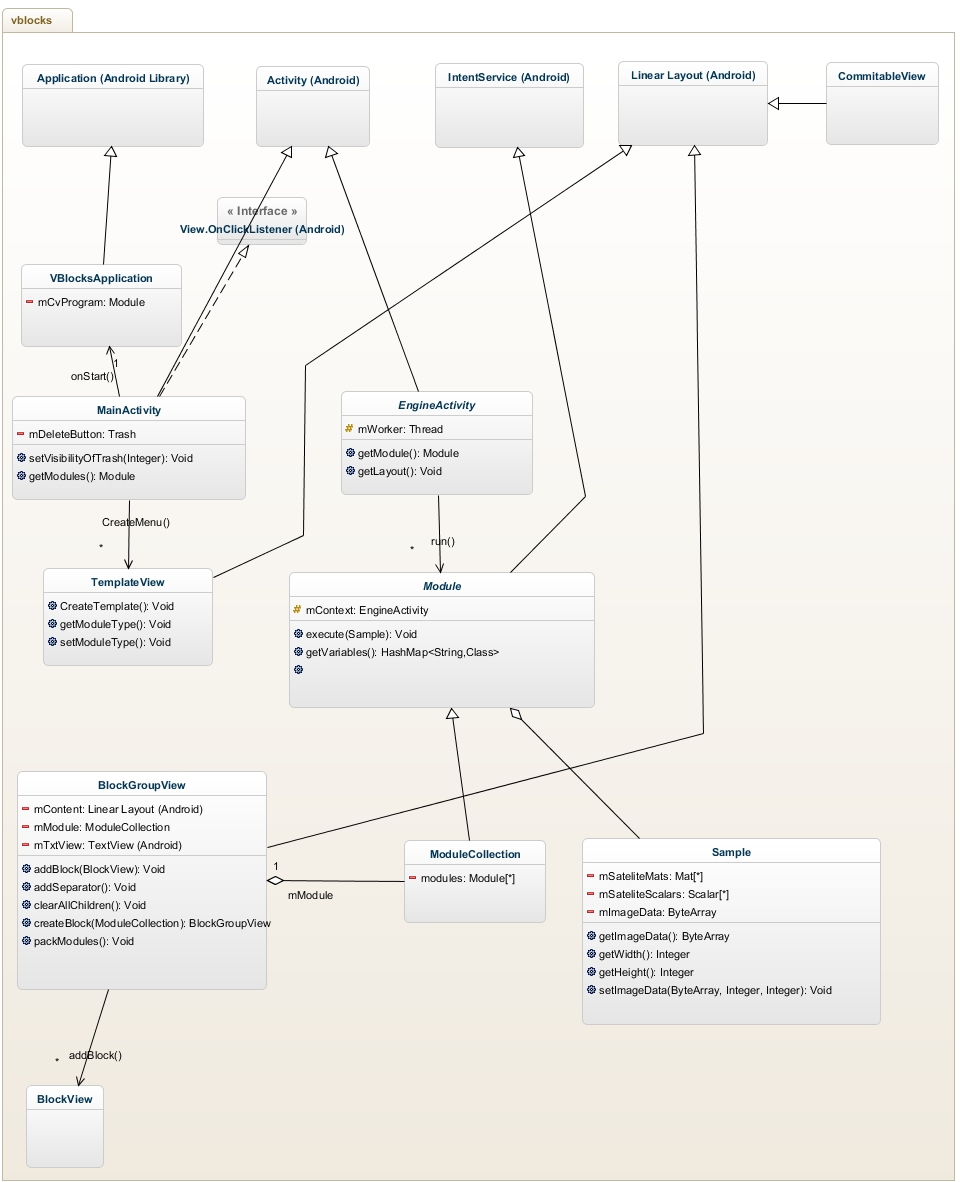
Configuration View 9

Drawer Menu 10

# Introduction

This document aims to show in a high-level language how the classes in Vision Blocks behave. The goal is to enable the reader to receive enough knowledge to implement a new *Module*, adding it to the project.

An UML diagram is shown below. It doesn’t represent all the classes in the project, but it will give you a better understanding of how the project is organized.

Diagram 1: UML class diagram of Vision Blocks.

# Project: vblocks-base

Represents the Vision Blocks engine. It has several abstract classes that serve as model for the Vision Blocks application.

## EngineActivity

Runs the activity related to the application devised by the client.

Uses the collection of modules selected by the client, creating a thread where the application will run.

It’s a class that extends *Activity* from Android SDK. Doesn’t have any accessible method.

## Module

Dictates the blocks behavior. It represents the shape of the blocks.

A sample from the camera is given to the block through the *execute* method as a Sample object.

This class extends *IntentService* from Android SDK. For more information on *IntentService* visit:

<http://developer.android.com/reference/android/app/IntentService.html>

* **Module(String serviceName);**Constructor: Defines the name of the module as serviceName.
* **CommitableView getConfigurationView(Context context);**
* **abstract ExecutionCode execute(Sample image);**
* **abstract getName();**
* **HashMap<String, Class> getVariables(Class classType);**

## ModuleCollection

A Collection of *Module*. It has methods from Collection class. Also behaves as a single *Module*, implementing the behavior of a Module, but with more instances. The class is devised this way because a module can contain modules inside it (just as a Camera module, or an if statement module).

It extends *Module* so it can behave correctly. This class is implemented the way it is because blocks can be a collection of blocks too.

* **ModuleCollection(String serviceName);**Constructor: Defines the name of the collection as *serviceName*.
* **boolean add(Module o);**
* **boolean addAll(Collection<? Extends Module> os);**…  
  All other methods from Collection  
  …
* **Module get(int i);**Search the <code>Module</code> in the i-th position.  
   Requires that the object has at least i elements.  
   @param i Index of the Module  
   @return Module in the i-th position
* **HashMap<String, Class> getVariables(Class classType);**

## Sample

Representation of an Image. It has the basic properties such as width, height, image data array, and matrices referent to the sample. It stores an image as a stream of bytes, and initially, has the YUV format (it’s the standard of the Android camera). In a lower level, the image is bypassed directly from the camera, once if you use the Android SDK you are not able to get a structured image to work with. In early versions, the Sample carries an already treated image in the RGB format as a Mat, to facilitate access and processing with OpenCV library. Is good to point that alterations in the Rgb\_Mat is propagated through all the blocks, and to view the output of this Mat, the user must put an *OutputModule* at the end, which will print the processed image on screen.

* **Sample();**Constructor: Initialize attributes with null and zeros.
* **Sample(byte [] data, int width, int height);**Constructor: Initialize attributes with their respective values.
* **Byte [] getImageData();**

...

Other getters

…

* **setImageData(byte [] imageData, int width, int height);**Set attributes, respectively.

## CommitableView

An abstract class that displays the module’s configuration view. Each module that needs a configuration view needs to have a *Config* class that extends *CommitableView*. The method *commit* is called when the user submit the configuration by pressing the *OK* button on screen. The OK button is defined outside the CommitableView, since it is shared code.

The way this class should be used is better explained in the [Block Creation](#_Module_Configuration) section.

# Project: Blocks

## VBlocksApplication

Android Application class that has *modules* in it.

It extends from *Application* from Android SDK. This is the application that will run when the Vision Blocks app starts. This application will start a *MainActivity* activity.

* **setProgram(Module sequence);**  
  Set the applications module to sequence.

## MainActivity

The Activity launched when Vision Blocks app starts.

It handles the interface where the user can select blocks, and devise its own Computer Vision enabled application. It is responsible for managing a menu where you can find all the blocks, the block organization, and all other elements that appear in the main screen.

It’s main layout is *activity\_drag\_drop*. The menu is made of several *TemplateView* instances, which can be selected and dragged to the *BlockGroupView* instance. The *BlockGroupView* is located in the middle of the screen. It is the place where the user will put the modules he wants to execute in his program. Once a *TemplateView* is placed in the *BlockGroupView*, it becomes an instance of *BlockView*. Each *BlockView* represents a module, and the module is instantiated at the same time as the BlockView’s *onCreate()*.

When the *start* button is pressed, a *PreviewActivity* object is instantiated, using the list of modules, which are extracted from the *BlockGroupView*.

*MainActivity* extends *Activity* from Android SDK. Is responsible for configuring the drag-and-drop layout, with the blocks, trash button, and start button. Handles the modules so they can be packed together, and sent to *PreviewActivity* to analyze and run them.

* **setVisibilityOfTrash(int v);**Set visibility of button <code>delete</code> according to <code>v</code>.  
  @param v VISIBLE, INVISIBLE, or GONE
* **CreateMenu();**Creates upper menu where you can select the blocks that will be used in client's program.  
  It populates the menu *LinearLayout* with *TemplateView*s. Each of these represents modules, and are draggable, so the user can put them in their program.
* **Module getModules();**Pack all modules present in <code>R.id.component</code> and return it.  
  @return Collection of all modules in the BlockGroupView.

## PreviewActivity

Responsible for running the custom application devised by the user. The application is defined by the sequence of modules in *mProgram*.

Since it extends *EngineActivity* in the *vblocks-base* project, it inherit all *EngineActivity* behaviors. In the *onCreate()* call, it initializes all the modules in *mProgram* by calling the *onCreate()* method of each module. In addition, it creates a new thread that is responsible for running the modules through the *executeModules()* method. It also configures the drawer layout, which has the drawer menu.

## PreviewMenu

Shows the blocks used to create the *PreviewActivity*. It is responsible for showing the options in the drawer menu, and also configuring listeners to it to handle touch events in the options of the menu. The menu has all blocks as options, and are shown inside a *ListView*. A single touch configures the selected block as the receiver of touch events (more information about how to get touch events in a block are later in this document), and a touch-and-hold action will call the *CommitableView* of the selected block.

## MainView

This view is set over all blocks when running the *PreviewActivity*. It is responsible for forwarding touch actions to a specific block, which is configured using the drawer menu. More about this drawer menu is located later in this document.

## BlockView

Represents each selected module that will be executed by *PreviewActivity*. These views are located in the middle of the screen, and sometimes they are configurable. It shows a menu when clicked, enabling the user to modify some properties of the module.

## BlockGroupView

A *GroupView* of *BlockViews*. It has instances of modules, which can be obtained by calling *packModules().*

An important thing of this class is that it is responsible for handling the option feature of each block. This means that *BlockGroupView* will get the module’s *CommitableView* by calling the method *getCommitableView*, bringing the configuration view to the client. When the client presses the *OK* button, the method *commit* in *CommitableView* will be called.

# Creating a Block

A block is a class that inherits from *module*. Each block brings a new function that the client can use in his program. It widens the realm of possibilities of applications that can be built. On the other hand, the module has to be made so it can interact with the existing blocks, which makes encapsulation is essential.

## Getting Started

First of all, create a new class in the *edu.mit.cameraCulture.vblocks.predefined* in the *Blocks* project. The class name must represent your block. Your class has to extend the *Module* abstract class, and therefore, it has to override the following methods:

* public ExecutionCode execute(Sample image);
* public String getName();
* protected void onHandleIntent(Intent intent);
* public static String getModuleName();

Don’t forget to implement your constructor as well. A good example is *Blur.java* or *Canny.java*, which are easy to understand.

## Implementing methods

### Execute

The *execute* method is responsible for all your block processing. It should perform all image processing, and then forward it to the next block. You receive your data from the image, which is a Sample object. We recommend the following steps in your execution:

* Check the image properties, and set them if necessary
* Do any conversions that your image needs.
* Apply your logic.

Right now, the implementation of all our code considers that image comes in the YUV format. Usually, we define a *Mat* (from *OpenCV*) to store it. The block checks the image width and height to know its orientation (also check this for initialization purposes too), and then we create the *Mat.*

The image processing result can be forwarded to the next module by adding the *Mat* into *Sample image*, or it can be printed on screen. Other options are available too, as printing *Views* on screen (for example, rectangles). It all depends on the *Module* functionality.

### getName and getModuleName

Both return a String that represents the *module’*s name. The reason why *getModuleName* exists is that it is static, so when we need to get its name (usually when creating the menu) we can get it without instantiating an object.

### getModuleTouchListener

You can override this method in case you want to get the touch actions performed by the user. This method should return a *ModuleTouchListener* object, which should have all the methods from this interface overridden. The main view of the *PreviewActivity* will use this listener to forward the touch events that occur over it. Once you override this method, the drawer menu will automatically recognize that this module can receive touch events, and will configure this listener when this block is selected.

The methods that the *ModuleTouchListener* object must implement are:

### onHandleIntent

It is required to implement this method once your module extends from *Intent* class from Android SDK. It is not necessary to do any changes in it for your module to work.

## Adding to the Menu

Go to the method *CreateMenu()* in the *MainActivity* class and search for which list you want to put your module in (e.g.: INPUT, FILTERS, etc.). Follow the example, modifying the arguments to match your class. For instance, when adding the *Canny* module, the following line was added in the FILTERS part:

filterItems.add(TemplateView.CreateTemplate(this, R.drawable.module\_filter\_button,R.drawable.module\_vision, Canny.class, Canny.getModuleName()));

Basically, add a new line, substituting Canny for your class name.

## Configuration View

After adding a block to the project, the client can click on it to obtain options for that specific object. For instance, the Blur block allows the user to customize the *Blur Block Size*.

First, create a *Config* class for your block (see *BlurConfig.java* for an example). Your *Config* class should extend *CommitableView*. You should put all your view creation into the generator method (build views to interact with the client). Also, you have to override the *commit* method, which is called when the client presses the OK button. In the *commit* method you should modify the configuration of your module by using the arguments obtained in the *Config* object.

When you define a *CommitableView,* you will automatically configure the Dialog that that opens the *CommitableView* when you press and hold the current module in the Block selection screen. Also, the same Dialog will appear when you press and hold the block inside the drawer menu, which you can access when running the application. More details about the drawer menu are found later in this document.

# Drawer Menu

This menu is located inside the *PreviewActivity*. It uses *DrawerLayout* from the Android Support Library. The layout of the *PreviewActivity* can be found inside the activity\_main.xml file. For more information about the *DrawerLayout*, visit the [Android Developer](http://developer.android.com/training/implementing-navigation/nav-drawer.html) website. Basically, this *DrawerLayout* configures the sliding pane located in the left, which loads an instance of *PreviewMenu,* which was previously described.

The advantage of this Drawer Menu is that you can set which block will receive the touch actions, and also can modify the parameters of the blocks on the fly. This means you don’t have to go back to the *MainActivity* again to configure the blocks anymore.