New Mexico State University

Master Final Report

Beat Blocks

Author:
Paolo Grignaffini

Supervisor:
Dr. Enrico Pontelli

May 6, 2019

To my Family,
to my Friends,
to my Future.

Contents

1 Introduction **1**
1.1 Motivation 1
1.2 Summary of the work done 2

2 Background **4**
2.1 Review of related work 4

3 Language Design **6**
3.1 Syntax and semantics of the language 7
3.2 Example of "program" 7

4 System Design and Implementation **10**
4.1 Top-Down description of the system 10
4.2 Blocks design 12
4.3 Implementation details 13
4.3.1 Aruco Markers 13
4.3.2 Camera calibration 14
4.3.3 Color Detection 15
4.3.4 Color calibration 16
4.3.5 Blocks Detection 20
4.3.6 Board Detection 22
4.3.7 Blocks logic 23
4.3.8 Loops 25
4.3.9 Configuration and Dependencies Files 27

4.3.10 SDL_mixer 29

5 Evaluation **31**

5.1 Test 31

5.1.1 Note test 33

5.1.2 Function test 34

5.1.3 Parallel test 35

6 Conclusions and Future work **37**

6.1 Future work 37

Acknowledgements **39**

Bibliography **40**

Chapter 1

Introduction

1.1 Motivation

The purpose of this project is to build a platform to let blind people/toddlers getting started with basic concepts of programming through the utilization of sounds and gestures.
The idea driving the implementations is simple: the user is able to define sound variables inside a Lego board, by using Lego blocks as syntactical entities, in order to produce melodies.
The semantics of the program is simple as well: the user is able to configure each block entity to contain a specific sound, and through the combination of such-defined blocks hear the resulting musical composition just by pointing with his fingers (or by clicking with the mouse) to the desired configuration.
It is possible to consider this platform as a modular musical instrument since there is basically no limit regarding what this tool can play and the possible musical configurations that can be built on it. Moreover, by letting the user to use his fingers to play, like a guitarist does with the strings of his guitar, the similarity becomes clear.
It is not only a tool for creating music though, it is also a learning platform. The main goal that the platform is trying to achieve is to give a taste of how programming feels like to people that cannot fully experience coding,

because of disease-related reasons (such as visual impairment) or because of young age.
I still remember from my undergraduate days how I felt when I was able for the first time to write down a working piece of code, the excitement of seeing the corrects results on the screen, and the realization that I was just walking my first steps in a world with endless possibilities for creation. From that moment I started to look at the programmer job differently, thinking of it as a creative job, where the developer assumes the resemblance of a pianist, that by pressing the correct buttons in the correct order, can produce something that is not material but still real, like music, and like software.
I want the user to feel the excitement of hearing the correct output, I want him to change variables' values to try different configurations and test his creativity through the most direct mean of communication in the world: music.
The aforementioned considerations have guided me during the whole development process of this project, which is discussed in the next sections.

1.2 Summary of the work done

The work done in this project can be summarized in three phases: (i) creation of the 3D models for the blocks, (ii) implementation of a mechanism to track objects through color detection, and (iii) testing.
Phase (i) took almost an entire semester to complete, due to the fact that in order to create the 3D models for the blocks in usage in the platform, a preliminary work of learning how to use Blender, which is an open-source toolset for graphics, needed to be done.
During the months of development, several different designs and block types have come into shape: a lot of these attempts have been discarded, while others have been modified during the process. Eventually, two types of blocks were chosen, the first one, which is a circle-shaped block, will be referred into the next sections as **Note** block, and the second one, which is a pentagon-

shaped block, will be referred into the next sections as **Function** block. For a detailed explanation of the **Blocks design**, please refer to subsection 4.2. The realization of phase (ii) and (iii) spanned over the time of two semesters, and it will be thoroughly discussed in (ii) section 4.3, and (iii) section 5.1.

Chapter 2

Background

2.1 Review of related work

Providing children affected by visual impairments (VI) with a framework for programming has been a challenge that several researchers from all around the world have tried to address during the past decades. The principal aim of these kind of platforms is to teach children Computational Thinking (CT) [1], which is a way of thinking that helps students to develop skills to think in an organized manner to solve a specific problem that can be expressed in an algorithm and be executed by a computer, and therefore not limited to the learning of writing programs [7]. Some of the most commonly-used approaches are Block-based programming [2], that allows children to program by connecting visual blocks, and Tangible programming [6], that makes use of manipulatives in which the students can relate concepts by using the sense of touch [3], [5], [4]. Unfortunately, most of the existing assistive technology are not compatible with programming environments used into introductory programming classes, and researchers are constantly looking for new methods of input, output, and feedback that are easier to use by students with VI.

This project introduces a new approach based on both Block and Tangible programming with the addition of musical feedbacks, that are used to provide

the output of the program to the user. It is not intended to provide an *extensive* framework for programming, since the block-based language designed is not (yet) Turing-complete, and it only stresses the concepts of functions, variables, and loops. However, the proposed platform offers a solid baseline for future developments targeted on improving the expressiveness of the language, in order to increment the set of functionalitis that this platform can offer to the students.

Figure 2.1: E-block

Figure 2.2: P-CUBE

Chapter 3

Language Design

Let's now take a closer look to the blocks, their positioning, and their meaning.

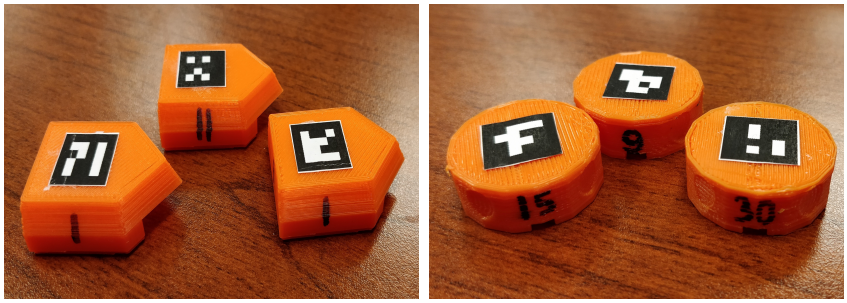

Figure 3.1: on the left: function blocks, on the right: note blocks

As we can see from Figure 3.1, the blocks present a label glued on them. This label is an **Aruco Marker**, which is utilized to give to each block a *unique* integer identifier, that is used for the **Block Detection**, discussed in sub-section 4.3.5. Further discussion about **Aruco Markers** and their usage inside the platform can be found in sub-section 4.3.1.
Returning to the blocks, we can notice that there is a slight difference between the two types: the **Note** blocks present numbers on their side, which correspond to their markers identifiers. On the opposite hand, the **Function** blocks present roman numbers on their side, which correspond to the

function that they refer to.

3.1 Syntax and semantics of the language

Note blocks are defined inside the platform through a *configuration* file that specifies the audio file contained in them.
Function blocks are defined using the content of the blocks placed on their right-hand, the resulting block will then contain the sounds represented by each entity concatenated.
Here we need to make a fundamental differentiation between **Function** calls and **Function** definitions.
A **Function** block, in order to be a definition, must be the first one to appear on the board, amongst all the other **Function** blocks with the same roman number. On the opposite hand, a **Function** block is considered a call when it presents a roman number that already appears on a **Function** block placed before it (considering the Origin placed on the top-left corner). Therefore, each **Function** block whose not a definition, will contain the sounds assigned to the first one defined.
Function blocks can be even defined using other **Function** blocks, the only constraint is that, in order to define a Function f_2 with another Function f_1, f_1 must be defined first.
The relationships between the different **Function** blocks are declared in the *dependencies* file, that specifies which identifiers, amongst all the ones belonging to **Function** blocks, refer to the same definition (have an identical roman number).

3.2 Example of "program"

In this section we consider an example to get a better understanding of the syntax and semantics of the blocks.
Let's then consider the following scenario:

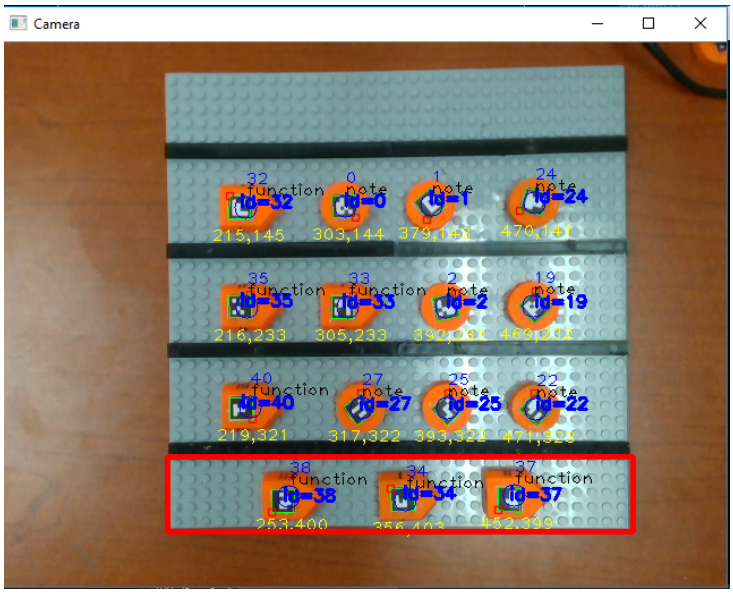

Figure 3.2: Example scenario

Which can be translated on paper into the following model:

$$+\!\!-\!\!-\!\!-

- the operator '&&' declares that the entities on its left and right-hand must be played in parallel

Every block inserted into the last line of the board is related to its neighbors with an '&&' operator, meaning that each block placed there will play in parallel (e.g. starts playing at the same time).
For this configuration, the *dependencies* file states that:

- Function 32 corresponds to Function 33 and 34 $\implies$ 32 == 33 == 34
- Function 35 corresponds to Function 37 $\implies$ 35 == 37
- Function 40 corresponds to Function 38 $\implies$ 40 == 38

Therefore, we can rewrite the proposed model as follows:

```
+-----------------+
+-----------------+
  32 = 0 + 1 + 24
+-----------------+
  35 = 32 + 2 + 19
+-----------------+
  40 = 27 + 25 + 22
+-----------------+
  40 && 32 && 35
+-----------------+
```

Which will output the sounds contained in Functions 40, 32 and 35 in parallel.

Chapter 4

System Design and Implementation

4.1 Top-Down description of the system

The diagram represented in Figure 4.1 offers an overview of the system from the perspective of the user. The **Blocks** and the **Board** are depicted as inputs for the user, which employs them to produce a configuration. The configuration, in turn, is fed into the program, that outputs the sounds defined by the blocks and their positioning over the board. The user is also able to dynamically change the blocks position while the program is running.

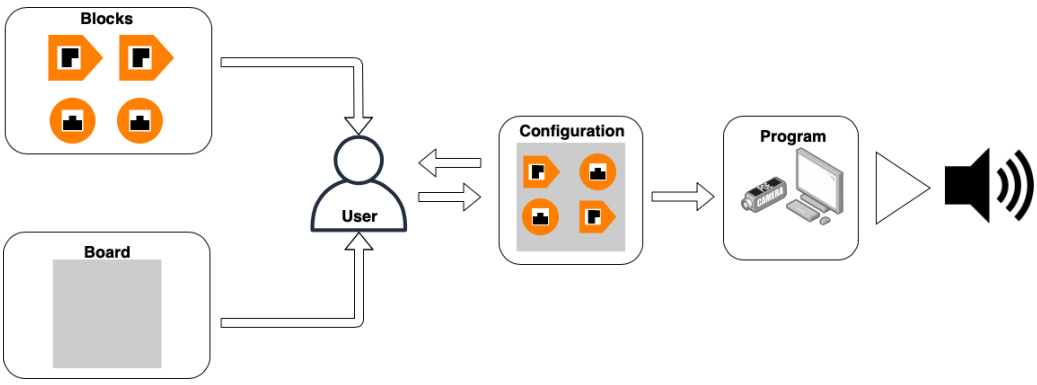

Figure 4.1: System overview

The diagram illustrated in Figure 4.2 offers an overview of the functioning of the **Program**. The **Program** starts by acquiring a **frame** from the camera, that is filtered through **HSV** to obtain the thresholds for both blocks and board.

The **Blocks threshold** is used to find the contours of the blocks, needed by the *tracking* algorithm to create the objects representation inside the program, namely *tBlocks*. The methodology used to accomplish this task is the topic of the discussion provided in section 4.3.5.

Next, the objects inside *tBlocks* are differentiated on the basis of the values defined in a *configuration* file, which splits them in two groups: **Notes** and **Functions**. Configuration and dependencies files are thoroughly discussed in section 4.3.9

On the other hand, the **Board threshold** is used to calculate the *convexHull* of the board, which returns its area and its corners. The implementation details can be found in section 4.3.6

Finally, the **Notes**, the **Functions**, and the **Board** characteristics are fed into the **Logic** to output the sounds.

For a full-detailed explanation of the **Logic**, please refer to section 4.3.7.

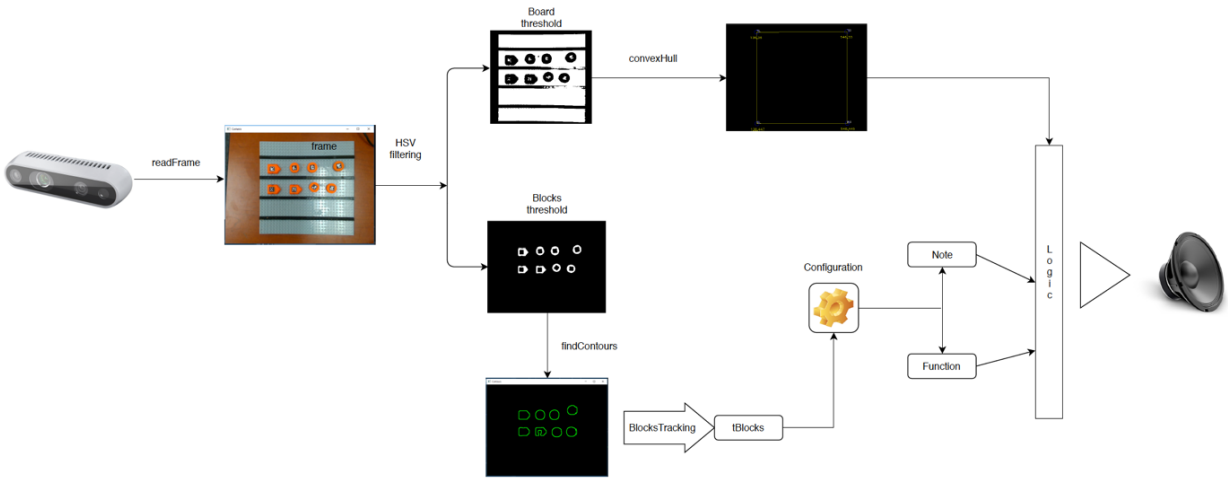

Figure 4.2: Program overview

4.2 Blocks design

While creating the models for the blocks, the main goal was to try to keep the design as simple as possible without sacrificing functionalities.
The original models for the blocks come directly from the official Lego website printabrick.org which is a terrific resource for finding Lego block designs that are freely available for the user to download and print.
The project started with 5 types of blocks, but after some reasoning and several attempts, it was decided that only two different types were essentially needed. As a writer only needs letters and an empty page to write them down, a composer only needs notes and a structure to contain them, and for this purpose, a pentagram fits best. Following this intuition, two types of blocks were realized: the first one, that is a circular block, represents the abstract concept of note. In this context the word "note" assumes a wider meaning, in fact it is not limited to a single sound as it is in music, but it can have an audio file configured in it, which means that it could even contain a 2 minutes' beat.
The second one is a pentagon-shaped block (**Function** block), which represents the pentagram. Its main purpose is to store different **Note** blocks to let the user play them in a given order, exactly like a music-sheet. Over the Lego board several pentagrams can be defined using the **Note** blocks and played in parallel to obtain the final composition.

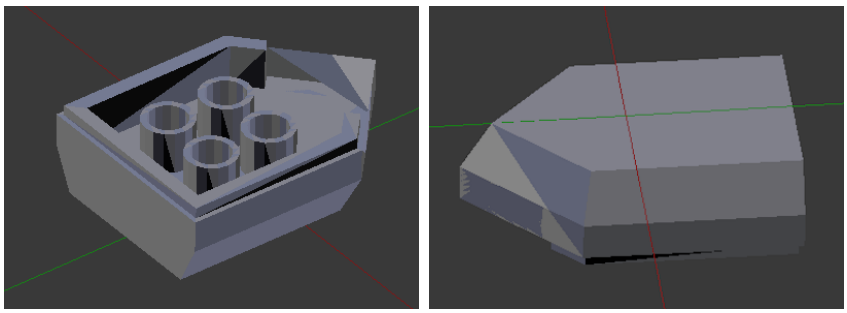

Figure 4.3: Function block 3D model

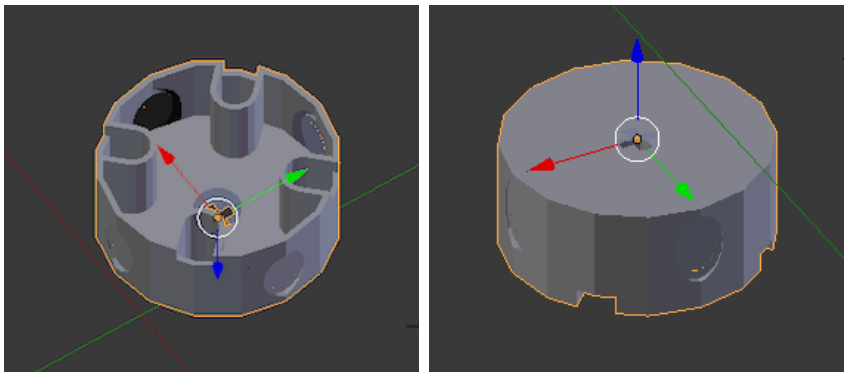

Figure 4.4: Note block 3D model

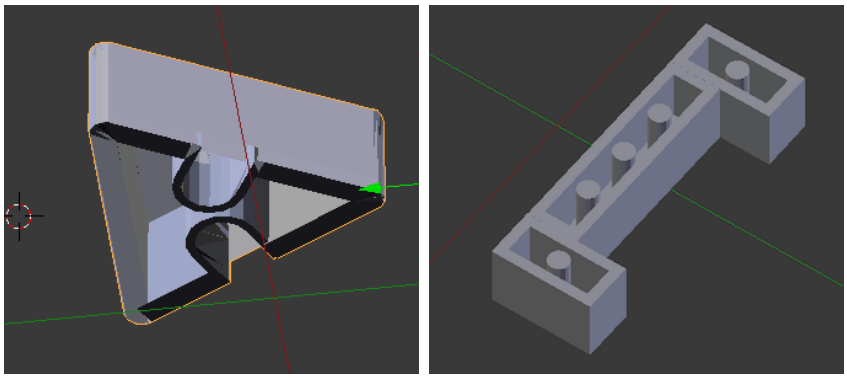

Figure 4.5: Some of the 3D models discarded

4.3 Implementation details

4.3.1 Aruco Markers

```
#include <opencv2/aruco.hpp>
```

An Aruco Marker is a binary square fiducial marker composed by a wide black border and an inner binary matrix which determines its identifier.

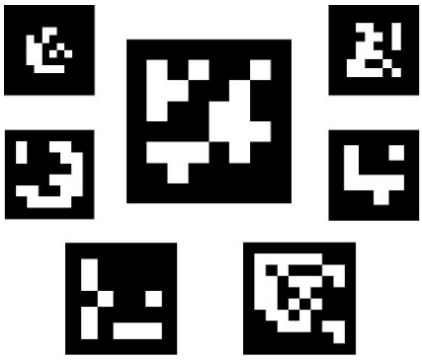

Figure 4.6: Examples of Aruco Markers

An unique Aruco marker is assigned to each block by physically gluing it on the surface of the blocks.
The Aruco module offers many types of markers to use, for this implementation have been used markers belonging to the Dictionary: DICT_4X4_50, meaning that they are implemented with a binary matrix of size 4x4 bits, and the range of values that they can represent goes from 0 to 49. This implementation choice limits the maximum total number of blocks that can be used to 50. If in the future this constraint will represent an obstacle, the user will need to choose a different type of Dictionary that can better fit his needs. This can be accomplished by changing the Dictionary used in the *BlockDetector::findCorners*(...) method*.
All of the different Dictionaries differ by the size of their internal binary matrix and the range of values that they can represent.
For a detailed description on how to properly select a Dictionary, please refer to: Aruco

*the new markers generated by the new Dictionary need to be printed and glued on the blocks

4.3.2 Camera calibration

Unfortunately, **Aruco Markers** do not come out of the box ready to be identified, but some preliminary steps need to be taken to obtain an accurate identification. The process of camera calibration for **Aruco Markers** can be really tedious, but it becomes pretty basic once you have coded the

functions to generate a configuration file for your camera that the Aruco module will use to set its inner values to discriminate the markers. Sadly, the Aruco module doesn't contain such functions, and therefore they must be coded separately. Luckily, the online community already took care of this issue, providing several methods to generate the configuration, which can all be found in the *CameraCalibration* class.
For a full-detailed description of the camera calibration process and the code contained in the *CameraCalibration* class, please refer to the playlist "OpenCV Basics" by George Lecakes, especially the sections regarding camera calibration. Link
One good thing about this whole camera calibration process is that it needs to be done only once for a given camera, thus once a configuration file is successfully generated, we can completely forget about it.

4.3.3 Color Detection

Now that we have the identifiers, we need to logically glue them on the blocks. What that means is, right now, we only have a bunch of numbers floating around the board, without any logical connection to their corresponding blocks, even if they are physically glued on them. This behavior is expected since, in the program, we never assert that "the identifier in the position p = (x,y) represents the block whose area contains p". To do so, we need to be able to identify the block in the space, calculate the portion of space that its area covers, and check which identifier matches it (the identifier position is automatically calculated by the Aruco module).
In order to identify a block in the space, Color Detection has been used.
The Color Detection technique used in this implementation is based on filtering HSV values.
The HSV (hue, saturation, value) model is an alternative representation of the RGB color model, which basically accomplishes the same task in a different way.
Thus, in order to identify any object through HSV filtering, it is enough to

map its color into HSV values. This mapping will create a threshold matrix which represents the image currently processed by the camera, normalized over the HSV values, meaning that everything in the image that is of the chosen color will correspond to a white pixel in the window, while all the remaining colors will be rendered in black. Given the threshold matrix to operate with, it becomes pretty easy to differentiate each object by considering each closed shape as a separate object, and then calculating its area and center of mass through the information given by the points lying on its contours.
So far, so good, but how do we determine the HSV values for the blocks (and the board too)?
Of course we could rely on predefined values found on the internet for a given color, but that would cause some inaccuracy in the detection that could be easily avoided by inserting the exact values for the given object. Therefore, implementing a tool to easily filter HSV values on-the-fly was much needed, and it is included in the driver class of the program: *trainedBlocksTracking.cpp*.

4.3.4 Color calibration

To invoke the HSV calibration tool from the driver class, it is sufficient to change the boolean property *calibrationMode* from false (its default value) to true. The implementation code can be found in the *ColorCalibration* class.
Once the *calibrationMode* is activated, this is the interface that is presented to the user.

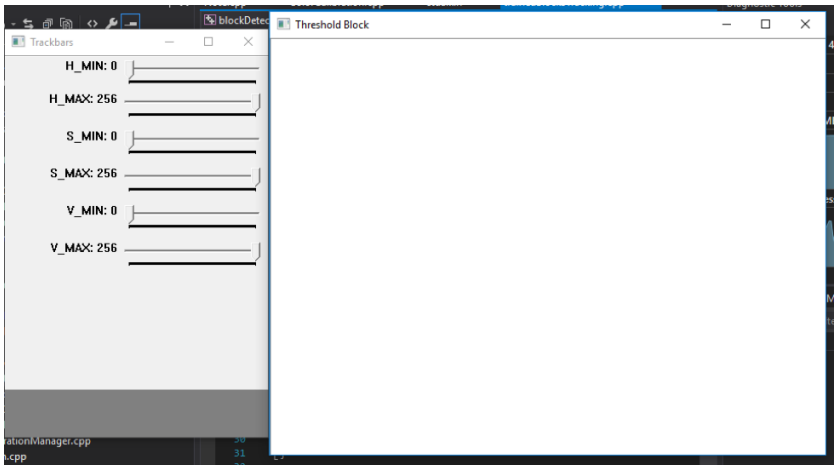

Figure 4.7: Color calibration with no values set

As you can see from Figure 4.7, when no custom values for HSV are set in the window "Trackbars", the returning threshold matrix (displayed in the right window) renders a blank image. To obtain the desired threshold matrix, we need to play with the trackbars on the left to get a clear representation of our objects in the window on the right.
For this implementation blocks of orange color and a board of grey color have been used, and their corresponding HSV values have been obtained by using the *ColorCalibration* tool to get their threshold, as you can observe from the following images.

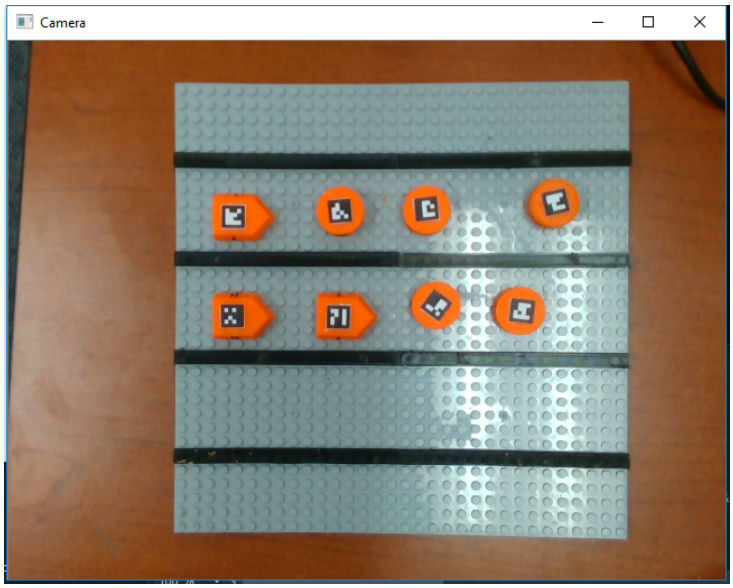

Figure 4.8: Original image

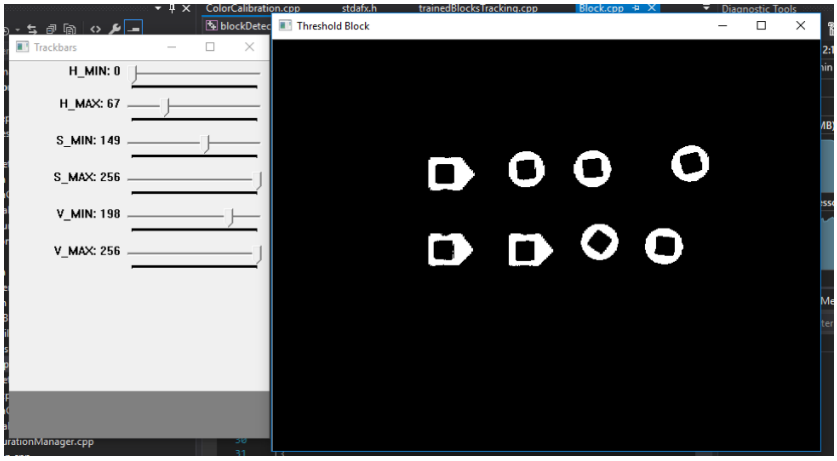

Figure 4.9: HSV values for orange blocks

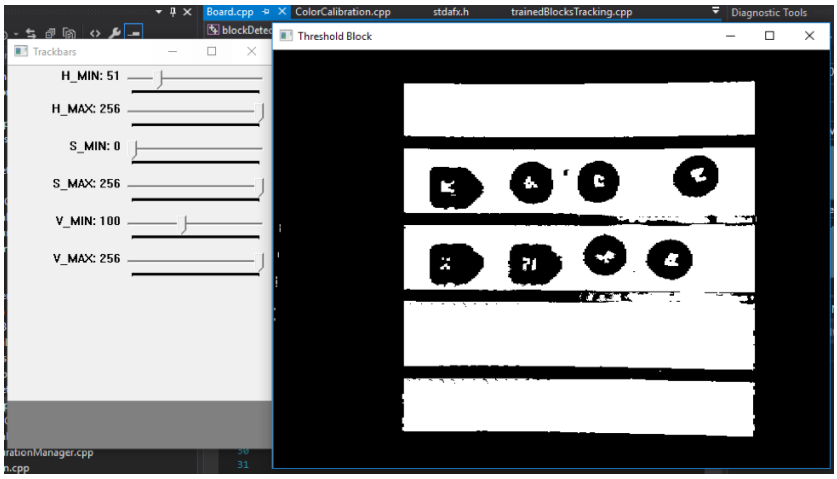

Figure 4.10: HSV values for grey board

As we can see from Figure 4.9, the HSV values corresponding to the orange color of the blocks are: H = (0, 67), S = (149, 256), V = (198, 256).
On the other hand, the values corresponding to the grey color of the board are: H = (51, 256), S = (0, 256), V = (100, 256).
The HSV values calculated can then be set to fill the properties of the corresponding object, which are defined in the constructor.
For example, the following snippet includes the code for the blocks' constructor which is defined in *Block.cpp*

```
Block::Block()
{
    //if we need to detect a different color we need to change the HSV values accordingly
    setHSVmin(cv::Scalar(0, 150, 200)); //hardcoded orange color
    setHSVmax(cv::Scalar(65, 256, 256));
    setXPos(0);
    setYPos(0);
    setID(-1);
    setType("X");
}
```

Figure 4.11: Blocks' constructor

From Figure 4.11, we can see where to call the methods *setHSVmin* and *setHSVmax* with the values calculated. The same applies for *Board.cpp*.

In this way, if one day the color of our blocks changes, we only need to calculate the new HSV values for the color through the ColorCalibration tool and set the values in the constructor accordingly.
Unfortunately, because of poor lightning in the room used for testing, the HSV filtering wasn't able to get a clean shot of the board, in fact the resulting threshold still contained some noise. In order to get a better representation, some morphological operations (such as dilatation and erosion) are applied to each matrix to get the final result.

Figure 4.12: Thresholds after morphological operations

Note: the hole in the middle-right part of the white square is caused by the direct light from one of the lamps over it, this condition can sometimes cause some inaccuracy on the marker identification.

4.3.5 Blocks Detection

Now that we have the threshold matrices for the objects that need to be detected, we must programmatically create those entities. This task is accomplished by the method *BlockDetector::findObject(...)*, which given a fixed threshold, finds the contours of each shape and check whether the given

shape is likely to be a block or not. What that means is, since the threshold matrix could still contain some noise, the algorithm only considers shapes with an area bigger than 20px by 20px.
The output matrix "contours" is then scanned to find the coordinates of the considered object inside the window using the moment method.
The information regarding the position and the contours of the block are given as input to the function *BlockDetector::update(...)*, which updates the status of the blocks.
The status of each blocks is saved inside a map, namely:

```
unordered_map<int , trainedBlock*> tBlocks
```

This structure contains all the blocks identified by the camera, an *unordered_map* has been used to easily access the information contained in a block given its identifier.

The function *BlockDetector::update(...)* works in the following way.
First off, given the coordinates of the block and an array of the identifiers detected, it checks which one corresponds to the block position and assigns the correct id to the block. This operation is accomplished by the function *BlockDetector::findIdentifier(...)*, which implements the logical gluing of the markers to the block.
Secondly, before adding the block to *tBlocks*, there are some properties that need to be checked. First thing to do is to check whether the block taken into consideration has already been identified or not.
Let's examine the latter case first.
If the block has not yet been inserted into *tBlocks*, it means that it is a new block that came into play. Therefore, it is sufficient to set its position and id and add it to the map. An implementation detail that wasn't mentioned before is that the *Block* class and the *trainedBlock* class (which is a derived type from Block) contain a property called type, which represents the type of the block identified, that can identify a **Note** (in the program: type == "note") or a **Function** (in the program: type == "function"), as discussed

in the **Introduction** chapter. Thus, the type of the block is also set before adding it to the container, and the type of each block is defined inside a configuration file that get parsed by the *ConfigurationManager* class, which will be discussed in section 4.3.9.
Now let's examine the case where the identified block is already inside *tBlocks*. In this case, the first thing to do is to update the block's position, if it has changed from the last iteration.
Then, some properties that regards the block status are checked.
In this context, the word "trained" referred to a block means that its type is defined in a configuration file, if this information about the type is available then the block is trained, otherwise it is not and it is discarded by the tracking algorithm. Another keyword is "defined", which means that the block has already been casted to the correct type.
In this implementation an original *trainedBlock** is casted to one of its derived class (*Note** or *Function**) after it is trained. This casting is essential for the overloading of the methods used by both **Note** and **Function**, that must produce different behaviors based on the type of the calling class.
Inside the body of the *BlockDetector::update(...)* function these properties are checked to understand the status of the considered block.
If the considered block is trained (meaning that its type is not "X") but not defined, then we need to cast it to the according type. In the case of a *Note** block the algorithm also loads into it the audio file declared in the configuration file. Otherwise, if the considered block is both trained and defined, then we only need to update its position, and in the case of a *Function** block, update its logic.

4.3.6 Board Detection

Now that we have all our blocks correctly detected and saved inside *tBlocks*, we need to take care of the Lego board.
From the board itself we only need few information, namely the coordinates of its bounding box vertices and the portion of space that occupies inside the

window (its area).
These information are retrieved by the function *Board::identifyBoard(...)* that represents the core of the *Board* class.
The *identifyBoard* method calculates, given the board threshold matrix, its convexHull points, and then uses those points to extract the corners of the board by applying some geometric operations.
The four corners of the board are defined as: TL (top-left), TR (top-right), BL (bottom-left), and BR (bottom-right).
These points are then used to calculate the board's area, and most importantly, to determine the last line of the board (the red rectangle in Figure 4.13), which identifies which blocks will be played in parallel.

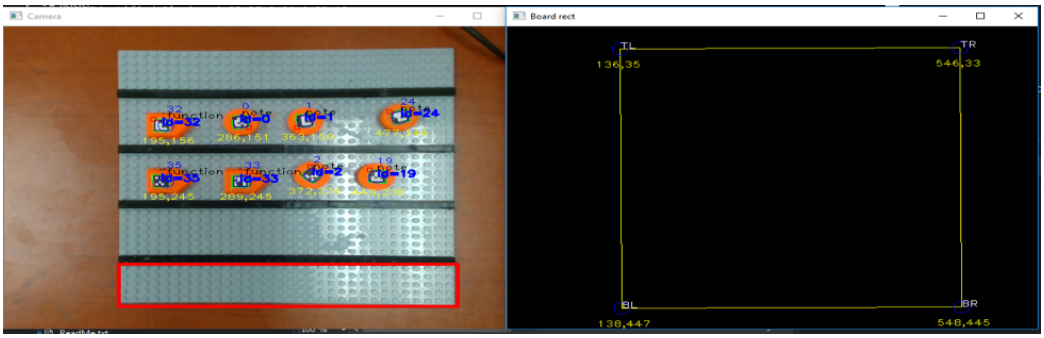

Figure 4.13: Board detection

4.3.7 Blocks logic

Pointing to a block or clicking on it with the cursor of the mouse will cause the firing of the method *CallBackBlocks*, which outputs the sounds.
CallBackBlocks takes advantage of polymorphism and methods overriding to produce the correct output for both **Note** and **Function** blocks.
The behavior of *CallBackBlocks* can be summarized in three steps:

1. Find which block has caused the callback to fire
2. If the block is a function definition, calculate which blocks are placed on its right-hand and set them as its content

3. Play the content of the block

The first step is trivially accomplished by searching, inside *tBlocks*, the block whose area contains the coordinates of the mouse's cursor and/or of your fingertips (depending on which input systems is in usage).
The second step is a little bit more complicated.
The hard task here is to decide whether the considered block is a function definition or a function call. In the first case it must look for its defining blocks, while in the latter case it must not, but it only should play the content defined in its definition, disregarding the fact that other blocks could be placed on its right-hand. To easily access this information, a special Boolean property, called "reference" is defined in *trainedBlock.h*, where a true value for this property assesses that the considered block is a reference to another one.
The value for "reference" is calculated inside the already mentioned *BlockDetector::update(...)* function, which internally calls the *BlockDetector::checkIfReference(...)* method.
The *BlockDetector::checkIfReference(...)* method basically checks if the considered **Function** block, between all the **Function** blocks saved in *tBlocks* that have an "equivalent" identifier, appears first on the board, meaning that a **Function** block to be considered the *firstOfItsKind*, must have the smallest x and y coordinates between all the other **Function** blocks that could be possibly reference, considering the Origin placed on the top-left corner of the window. If this is the case, then the **Function** block will be set as *firstOfItsKind*, and its content will be defined by the blocks on its right-hand side. On the opposite hand, if this is not the case, the algorithm will look for the *firstOfItsKind* block (which is the only one with the "reference" property set to false), and it will assign a pointer to that block as the content of the considered block.
Now that we can distinguish between a function definition and a function call, we can proceed on examining how the program plays the content of the blocks.

The body of the function *CallBackBlocks* calls internally the method *play()*, which is a virtual method declared in the *trainedBlock* class and overridden both in *Note* and *Function* classes.
In the case of a **Note** block calling the *play()* method, an external library is called to perform the reproduction of the audio file configured into it.
In the case of a **Function** block calling the *play()* method, its content is disassembled into the basic **Note** blocks defined into it, and then the *play()* method is iteratively called for all the notes.
For example, let's consider the following scenario:

```
+----------------+
  F = 33 + 2 + 19
+----------------+
  Q = F + 25 + 22
+----------------+
        Q
+----------------+
```

Then the output reproduced by Q will result into the sequential concatenation of the sounds contained in: Q = 33 + 2 + 19 + 25 + 22

4.3.8 Loops

An interesting feature that has been added to the language during the last phases of development is the introduction of loops. In this context, loops can only be infinite, since conditional logic is not supported.
In Figure 4.14 are reported the loop blocks as they look like in real life, while in Figure 4.15 is displayed a possible configuration with the addition of the loops.
The semantics of these blocks is really simple: if a **Loop** block is placed on the same line of a **Function** block, then the content of the **Function** will be played forever, and the only way to interrupt the reproduction will be to physically move the **Loop** from its position. The tracking for these kind of

blocks is really similar to what already discussed in 4.3.5, the only difference is that in this case the tracking algorithm is more simple, since we do not need an *unique identifier* for each block but only their position on the board. If a **Loop** is detected in a position contained in the range of a **Function**, then the considered **Function** will be set as *looping*, and the first time that the user decides to play its content the reproduction will keep looping.

Figure 4.14: Loop Blocks

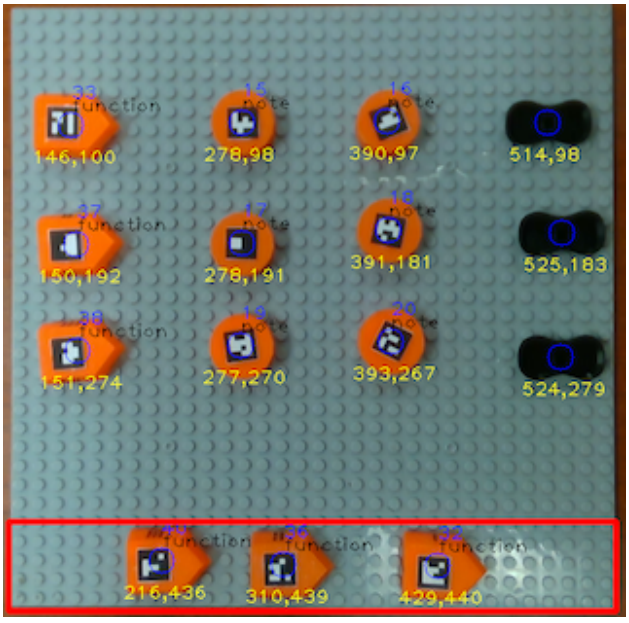

Figure 4.15: Configuration with loops

4.3.9 Configuration and Dependencies Files

Let's take a step back now to realize what we have accomplished so far.
The system is able to successfully detect the blocks and the board, through HSV filtering, create the relative object representation in the program, and understand the logic of their positioning.
Following the music parallelism mentioned in the **Introduction** chapter, we can think of it as having an orchestra: we know the name and the position of each player, and the players themselves know how to play music with each other.
What does this scenario miss? Musical instruments. And this is where configuration files come into play.
We can think of a configuration file as a list with names and instruments that need to be played by a specific person.
Returning to our implementation, the configuration file specifies which **Note** block contains which sound. The configuration file(s) are placed in the folder "conf" inside the project's directory.
The following screenshot contains a partial representation of the file "pianoConf.txt":

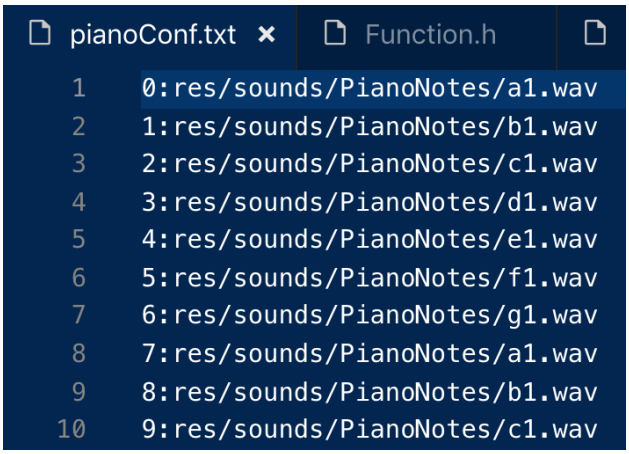

Figure 4.16: pianoConf.txt

The numbers on the left side of the colon ":" represent the marker id to which

the wave file specified on the right will be assigned.
For instance, in the scenario represented by Figure 4.16, we have that the block represented by the marker id 0 contains the file "a1.wav" specified in the folder "res/sounds/PianoNotes". There is no limit on the length of the wave files that can be reproduced by the **Note** blocks.
In this way, a user can simply switch from one sound to another by editing the configuration file used by the *ConfigurationManager* class, which is specified in its constructor.
Now that we have our sounds, we still need to define which Function blocks are "equivalent", meaning that they refer to the first one defined. These dependencies are declared in the file "deps.txt" contained in the "conf" folder as well. Here is how "deps.txt" looks like:

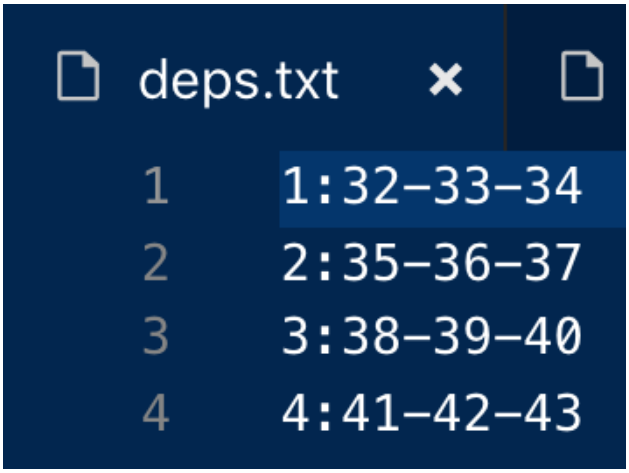

Figure 4.17: deps.txt

The numbers on the left side of the colon ":" specify the roman number that can be found on the side of the blocks identified by the ids on the right.
The ids on the right express which blocks are "equivalent". Notice that the identifiers must be separated by a dash symbol "-", to be properly parsed by the *ConfigurationManager* class.
For example, the first line of "deps.txt" asserts that the **Function** blocks with identifiers 32, 33, and 34 are equivalent, and they all have a roman

number "I" on their side.
Furthermore, the user that wants to create new **Function** blocks must be sure that the markers represented by the ids on the right are correctly glued on a **Function** block with the relative roman number on its side.
In fact, it is of critical importance that the identifiers contained in "deps.txt" do not overlap with the ones contained in the configuration file(s), resulting in an unexpected behavior. Some guards could be coded inside the *ConfigurationManager* class to warn the user if this situation takes place.
The type of the blocks is discriminated using those two files: each identifier that compares in "deps.txt" corresponds to a **Function** block, while all the remaining ones defined in the configuration file(s) correspond to **Note** blocks.

4.3.10 SDL_mixer

```
#include <SDL_mixer.h>
```

SDL_mixer is an external library that has been chosen in order to reproduce sounds inside the platform, and its usage is limited to the *Note* class.
Moreover, only the methods needed for music reproduction have been used, while the remaining set of functions included for mixing and editing have been set aside. The full documentation is available at: SDL_mixer
Returning to the *Note* class, by looking at the *Note.h* file, you can see that it contains a special kind of property, called "chunk" of type *Mix_Chunk**.

```
private:

    string fileAudio;
    ConfigurationManager* config;
    Mix_Chunk* chunk;

};
```

Figure 4.18: Note.h

This is the type of data that *SDL_mixer* uses to reproduce the sound.
Once a **Note** block is properly defined, it needs to load its audio file. This task is accomplished by the function *Note::loadSound(...)*, which stores the audio specified in the configuration file inside the "chunk" property by internally calling the library function *Mix_LoadWAV(...)*.
One of the best features of this library is that the different sounds can be played on different channels. Each channel is identified by an integer, starting from 0.
When the function *Mix_PlayChannel(chunk, channel, ...)* is called inside the body of *Note::play()*, it looks for a free channel (if none is specified), plays the sound, and returns the index of the channel where "chunk" is playing on.
The good thing about channels is that if we want to play a sound on a channel that is occupied, we need to wait for it to be free. This means that if we want to play sounds sequentially, we only need to play them on the same channel, waiting for each one to be finished, while if we want to reproduce sounds in parallel we just need to assign a different channel index for each audio that we need to play, making the reproduction extremely easy.

Chapter 5

Evaluation

In this chapter the output of the program will be analyzed trough different test cases, to demonstrate that the platform works as expected, as discussed in the previous sections.

5.1 Test

Let's then consider the following scenario:

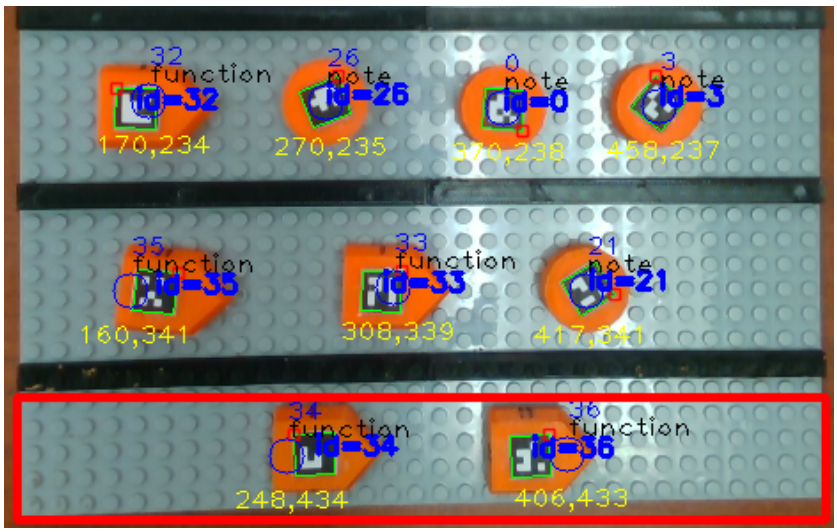

Figure 5.1: Test Case

Which can be translated on paper into the following model:

$$+\text{———————}+$$
$$32 = 26 + 0 + 3$$
$$+\text{———————}+$$
$$35 = 33 + 21$$
$$+\text{———————}+$$
$$34 \;\&\&\; 36$$
$$+\text{———————}+$$

For this configuration, the *dependencies* file states that:

- Function 32 corresponds to Function 33 and 34 $\implies$ 32 == 33 == 34
- Function 35 corresponds to Function 36 $\implies$ 35 == 36

Therefore, we can rewrite the proposed model as follows:

$$+\text{———————}+$$
$$32 = 26 + 0 + 3$$
$$+\text{———————}+$$
$$35 = 32 + 21$$
$$+\text{———————}+$$
$$32 \;\&\&\; 35$$
$$+\text{———————}+$$

Moreover, the *configuration* file "pianoConf.txt" used for this run states that:

- Note 26 contains: d1.wav
- Note 0 contains: a1.wav
- Note 3 contains: d1.wav
- Note 21 contains: f1.wav

Thus, we can re-rewrite the model as follows:

+————————————+

$$32 = d1 + a1 + d1$$

+————————————+

$$35 = d1 + a1 + d1 + f1$$

+————————————+

$$32 \;\&\&\; 35$$

+————————————+

5.1.1 Note test

In the following images is reported the output of the program for the firing of the method *play()* on **Notes** 26, 0, and 3.

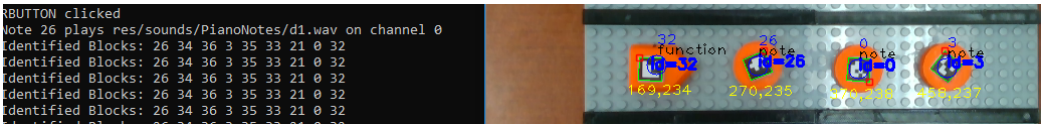

Figure 5.2: Note 26

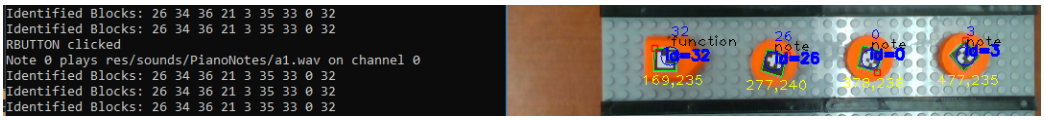

Figure 5.3: Note 0

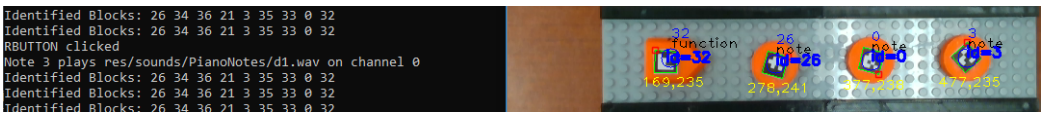

Figure 5.4: Note 3

From the output window on the left, it can be noticed that the different **Notes** output correctly the wave file configured into them.

5.1.2 Function test

Let's now examine the output reported for the **Function** blocks: what we are interested to see here is whether the **Functions** are able to correctly reproduce the sounds in the order given by the blocks disposition on the board.

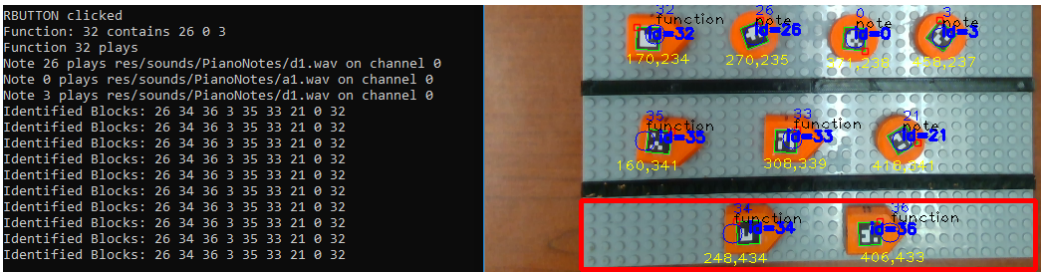

Figure 5.5: Function 32

Figure 5.6: Function 35

From the output window on the left, it can be noticed that the **Functions** output the sounds defined in the blocks in the correct order.
During the execution of the program the blocks can be moved to obtain different sounds combinations, in the following pictures is reported the output for the **Functions** given a new configuration, which has been created by switching the place of **Note** 26 and **Note 3** in the first line of the previous configuration, and by switching the place of **Note** 21 and **Function** 33 in the second line.

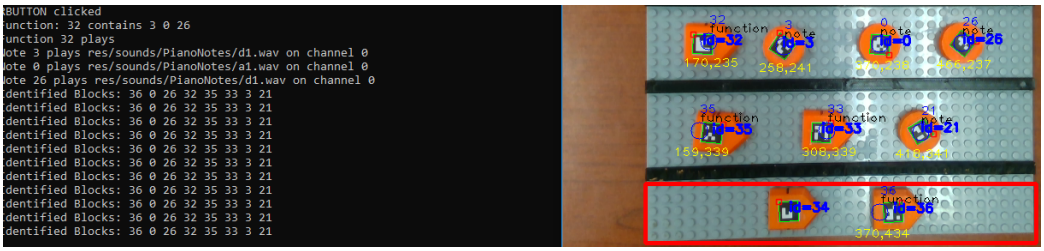

Figure 5.7: Function 32

Figure 5.8: Function 35

This test case shows that the **Function** blocks correctly mantain the order of the blocks defined, even when those are moved from their initial position at run-time.

5.1.3 Parallel test

In order to obtain the parallel execution of the sounds, the *#pragma omp parallel for* has been used over the blocks positioned in the last line of the board, which all call internally the method *play()*. In the following image is reported the output for the parallel execution of blocks 34 and 36. Unfortunately in this case the output is not really representative of the correct behavior, since the parallel execution of the method *play()* cause the prints to overlap each other, but for this very same reason, it should be clear to the user that the processes are really executed in parallel.

The parallel execution can be obtained by pointing with a finger to the last

line of the board, or by clicking the left button of the mouse (in any area of the window).

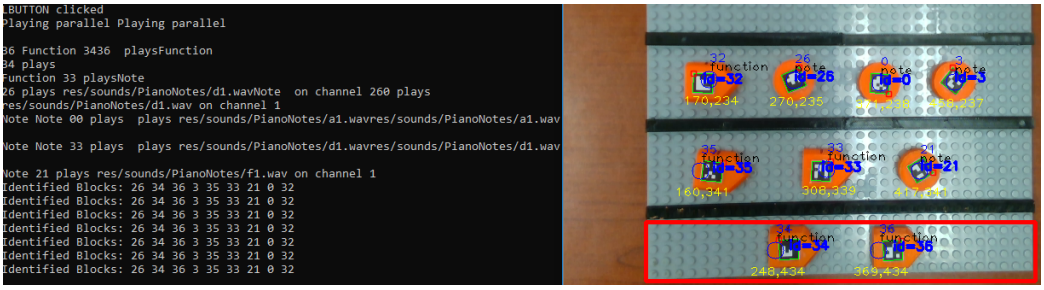

Figure 5.9: Parallel execution

Chapter 6

Conclusions and Future work

In conclusion, with the presented work have been accomplished the following tasks. First, the realization of custom 3D models for Lego Blocks, that can be used as a baseline for future designs. Secondly, the implementation of a tracking mechanism for real-life objects based on HSV Filtering and on the usage of Aruco Markers. The tracking is independent from the logic of the application, and therefore it should be relatively easy to extrapolate it from its context in order to use it for different applications. Lastly, the creation of a language based on Lego Blocks and their positioning, aimed to expose the user to basic concepts of programming, such as functions, variables, and loops, through the utilization of sounds and gestures.

6.1 Future work

Since the proposed platform is a new system that has been developed from scratch, there are many aspects that could be improved in the future to benefit the user experience. The future works should be oriented towards the following directions:

- Realization of a GUI to let the user change the sounds configured inside the **Note** blocks dynamically at run-time

- Implementation of a tool to automatically generate configuration files for **Aruco Markers**, given the camera in usage
- Implementation of a tool to automatically calibrate the colors of the blocks and the board
- Analysis of the whole set of functionalities that *SDL_mixer* offers, to enhance the audio reproduction
- Extension of the existing 3D models to present numbers carved on their side, or even better, the implementation of a tool to automatically insert numbers in a given position inside of a given model
- Addition of new blocks and logic to increment the expressiveness of the language
- Create new configurations file with new sounds to play

Acknowledgements

My first thank you goes to Dean Enrico Pontelli, who has been my role model for the entire duration of this experience. Furthermore, he gave me the chance to come to the United States to attend college, he gave me a job, and he opened me the doors of his office in Science Hall, which has basically been my home for 12 months. I hope that this work will serve you good.
The second thank you goes to Fabio, who has been my office-mate, my friend, and the person that has always helped me out when I was facing problems that I couldn't solve, your teachings have been the most valuables.
The third thank you goes to all the people that I got to know during this experience at NMSU, amongst them I want to express a special thank you to Francesco and Casey.
Lastly, I want to give my biggest thanks to my family, especially to my mother and father, for giving me the freedom to choose and the means to realize my aspirations.

Thank you all.

Bibliography

[1] A. V. Aho, "Computation and computational thinking," Comput. J., vol. 55, no. 7, pp. 833-835, 2012.

[2] J. Maloney, M. Resnick, and N. Rusk, "The Scratch programming language and environment," *ACM Trans. Comput. Educ.*, vol. 10, no. 4, pp. 1-15, 2010.

[3] D. Wang, Y. Zhang, and S. Chen, "E-block: A tangible programming tool with graphical blocks," Math. Probl. Eng., vol. 2013, 2013.

[4] S. Kakehashi, T. Motoyoshi, K. Koyanagi, T. Ohshima, and H. Kawakami, "P-CUBE: Block type programming tool for visual impairments," *Proc. - 2013 Conf. Technol. Appl. Artif. Intell. TAAI 2013*, pp. 294-299, 2013.

[5] L. R. Milne, R. E. Ladner, and P. G. Allen, "Blocks4All: Overcoming accessibility barriers in blocks programming environments for children with visual impairments," 2018.

[6] V. Koushik and S. K. Kane, "Tangibles + Programming + Audio Stories = Fun," in *Proceedings of the 19th International ACM SIGACCESS Conference on Computers and Accessibility - ASSETS '17*, 2017.

[7] E. Utreras, "Literature Review of Programming tools for children with Visual Impairments", New Mexico State University