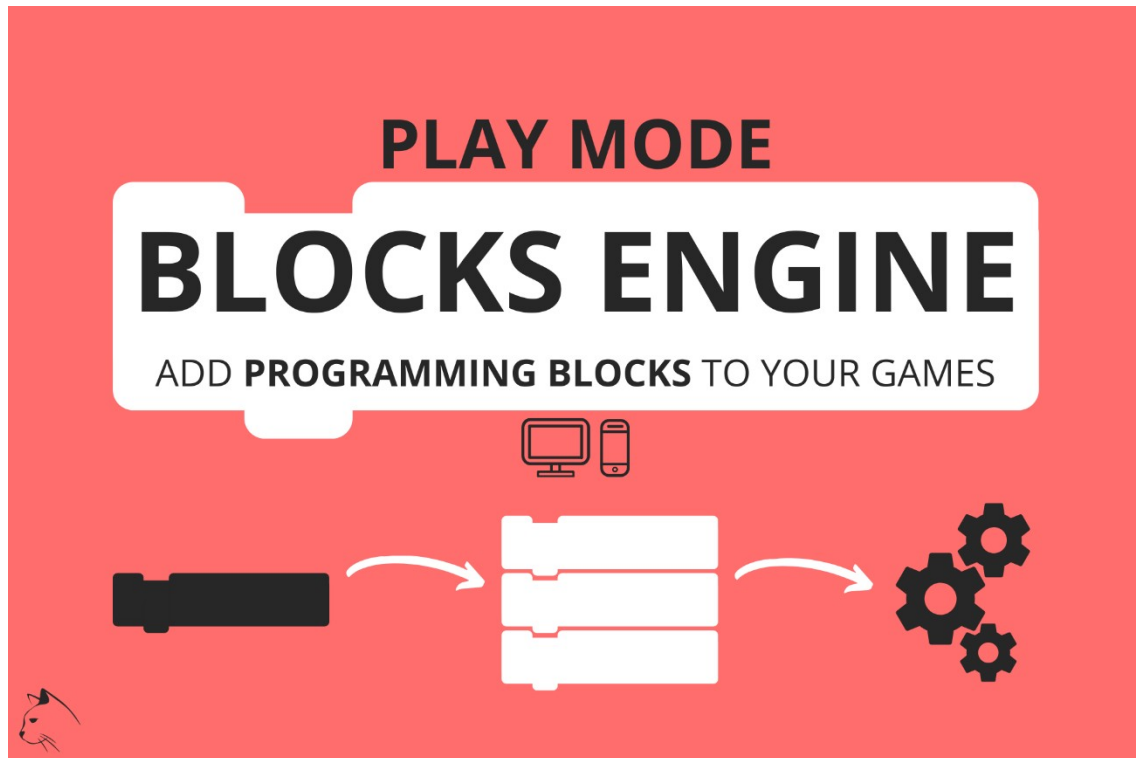


Play Mode Blocks Engine



Add **Programming Blocks** to your desktop or mobile Games easily with this engine based on the most popular block programming languages. For educational, logic puzzle or any other type of game.

This Play Mode Blocks Engine asset documentation contains the detailed explanations for the implemented features, classes, methods, variables, concepts and setting adjustments.

Contact



assetstore.unity.com/publishers/19535



github.com/danielcmcg



d.cavalcante.m@gmail.com

Daniel C Menezes
<http://danielcmcg.github.io>



1 CONTENTS

1 Contents.....	2
2 Overview.....	3
3 game Interface.....	4
3.1 Game View.....	4
3.2 Programming Interface.....	4
3.2.1 Block Selection View.....	5
3.2.2 Programming Environment.....	7
4 Model Classes.....	9
4.1 BE Target Object.....	9
4.2 BE Block.....	9
4.2.1 Block Types.....	9
4.2.2 Block Header.....	10
4.2.3 BE Inputs.....	11
5 BE Controller.....	11
5.1 BE Instruction.....	11
5.2 Instruction Stack.....	12
5.3 BE Variables.....	12
5.4 Accessible BE Lists.....	12
5.5 Play Next Methods.....	13
5.5.1 Play Next Outside.....	13
5.5.2 Play Next Inside.....	13
5.6 Scene (Inspector).....	13
5.7 Workshop (Inspector).....	14
5.7.1 Building a Block and Related Instruction.....	15



2 OVERVIEW

The Play Mode Blocks Engine is a complete project that implements a drag and drop blocks programming engine to be used in desktop or mobile games.

The blocks layout and rules of snapping and construct the code have similar patterns as the most popular blocks programming languages. This Engine, however, is built based on the Unity classes, making it possible to deploy for any target platform supported by the Unity engine.

It runs in desktop and mobile using simple touch events (drag, click, right click (pc) or hold) and can be adapted for different screen sizes with the proper scaling and adjusting of the layout.

The system is composed of a Game View, where the result of the code is reflected and viewed; Programming Interface (UI), where the blocks are dragged and dropped to construct the code; Model Classes, that represent the main components of the Engine; and a Controller, that manage the code cycles and states.

Despite the familiar component names, it is not but roughly similar as an MVC architecture, since the complexity of the Engine, the possibility of changing values during the running of the blocks code cycles and some particularities make the controller rely on some blocks (UI) internal states, yet, the blocks code can run while the UI is disabled.

One thing to have in mind is that the Engine may not perform well with heavy graphics and polygons, even though the scene can have multiple 3D Target Objects and multiple triggers at the same time, it is still an Engine that has a complex UI system tangled with the controller and a CPU like process implemented. The Blocks Engine is, though, a base for creating educational, code learning and puzzle focused games based on the functionalities already present on this project, drag and drop blocks to construct a sequence of instructions to manipulate one or some Target GameObjects in the scene, thus, to provide an experience of coding with simple actions.

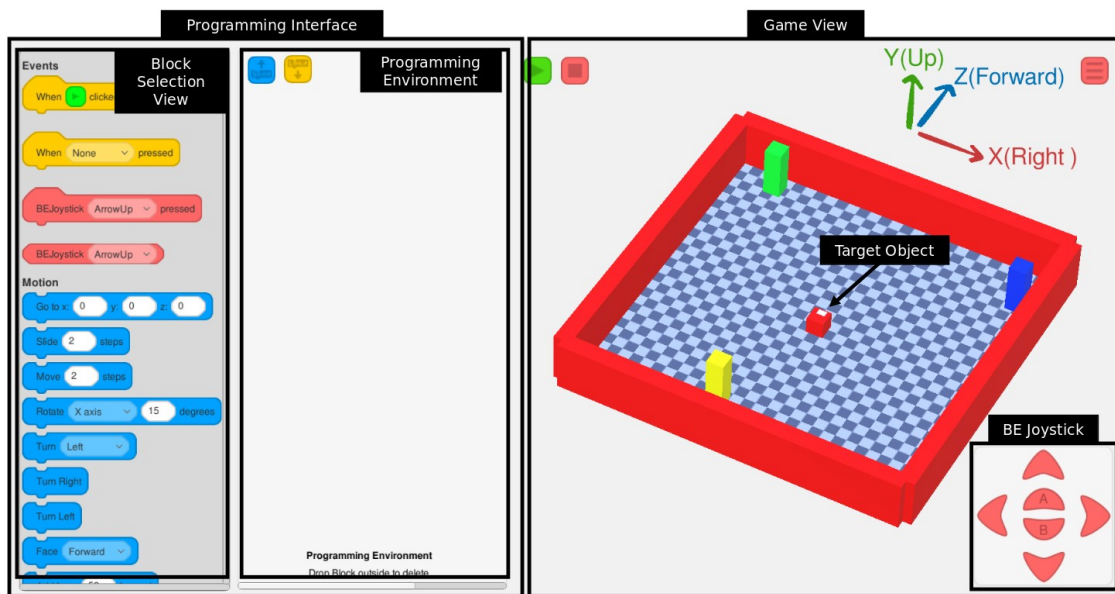
The Engine concepts are detailed and explained in the next sections.



3 GAME INTERFACE

The game interface has the needed objects set up and ready to be used by the players to try their own blocks code and see the results. The UI still can be adapted with additional objects to improve the usability, as adding a Target Object selection view or a functionality for changing the Target Object Sprite (Planned to next versions).

Below there is a print of the sample scene game interface with Game View and Programming Interface composed of Block Selection View and Programming Environment.



It is possible to easily modify the basic UI characteristics as size and position of the components, also the size and shape of the blocks with some changes on the code, although it is not recommended to highly modify the UI hierarchy, since the model is based on the UI blocks hierarchy organization.


3.1 PROGRAMMING INTERFACE

The programming interface is the whole UI provided for the player to build their blocks code. Along with sections for choosing blocks and for place the blocks in sequence to form codes, it includes functionalities as Play, Stop, Save and Load code.

The buttons for playing and stopping the running of the codes are placed in the top right corner of the Game View, however, they can be placed wherever fits better your project.



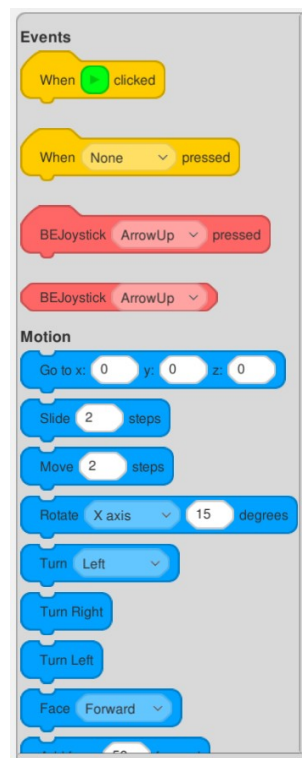


The Stop button will stop any block group that is active, but the Play button will start only the block groups that starts with a specific Trigger block set to respond to this input, as the “When  clicked” block.

3.1.1 BLOCK SELECTION VIEW

The Block Selection View is part of the programming interface and is the section where all the blocks the player can choose are exposed in categories.

The blocks at this section are fixed, once the player drags it, another block is created so it is possible to drop it onto the Programming Environment.



3.1.1.1 Blocks

The blocks represent functions, operations or variables the Player can use to build a complete code. The variable blocks are also considered as operation blocks, since both perform an instruction that return a value to be used as Input of the Function blocks.

The Engine was made to achieve a generic coding behavior, therefore the player does not need to worry about input or variable types.



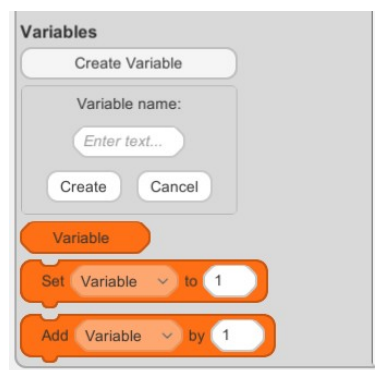
- **Function Blocks:** The Function blocks perform an instruction that is going affect or Target Object or the block group cycle;
- **Operation Blocks:** Operation blocks return a value and are used as input for Function blocks.

3.1.1.2 Categories

Each category of the Block Selection View is related to the general behavior that the instruction will cause to the Target Object or block group.

- **Events:** Trigger blocks that function as event listeners and are needed for the activation of a block sequence;
- **Motion:** Blocks related to movement, positioning and direction of the Target Object;
- **Control:** Blocks that act on the block group, enabling or not blocks to be executed or pausing the cycle of instructions;
- **Looks:** Blocks that alter the visual aspect of the Target Object, currently only the color;
- **Sound:** Blocks that execute sounds;
- **Operators:** Blocks that perform logic operations and are used as inputs for function blocks;
- **Variable:** Variable blocks and Blocks that perform variable related operations.

In the Variable category, there is a button for creating new variables that opens a panel to insert the new variable name and confirm the creation.



3.1.2 PROGRAMMING ENVIRONMENT

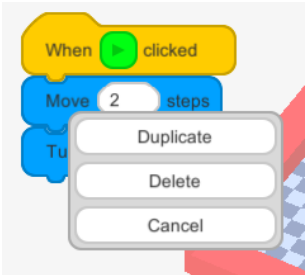
The Programming Environment is the actual coding section, where the blocks are placed to form the sequential block groups that are going to be run.

Each Target Object have its own Programming Environment, enabling the possibility of having multiple Target Objects on scene and each with a set of blocks. For abstraction purposes, each Target Object has its own Canvas

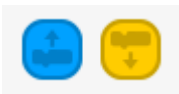


containing the Programming Environment (ScrollView), they are bound by the BEProgrammingEnv component attached to the canvas.

Once the blocks are dropped in the Programming Environment, there is the possibility of opening a panel with actions by hold/right click on the desired block. This menu contains the Duplicate and Delete actions that are going to be applied to this block.



In this environment are also two buttons, Save and Load code, for the correspondent feature.

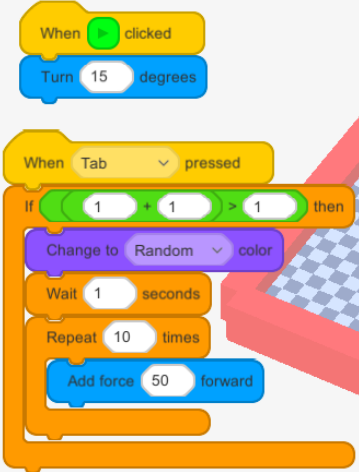


3.1.2.1 Save Code

The save feature performs a simple file save of the current canvas blocks and creates a .BE file that can be opened by any text editor to easy manipulation of the data and understanding of the code. There are custom functions to handle the translation of the blocks into text.

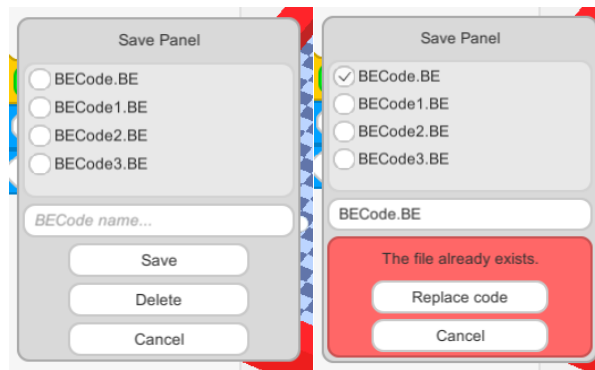
The codes are saved in a folder called “SavedCodes” in the persistent data directory provided by Unity from Application.persistentDataPath.

Below there is an example code translated from blocks code to the custom Blocks Engine text-based code.

Blocks code	BE text-based code
	<pre>ControllerMain():341.9,524.2501 FunctionTurn('15) ControllerWhenPressed('Tab):356.9,389.2501 FunctionIf(0OperationBiggerThan(0OperationSum('1','1'),'1)) FunctionChangeColor('Random) FunctionWait('1) FunctionRepeat('10) FunctionAddForceForward('50)</pre>



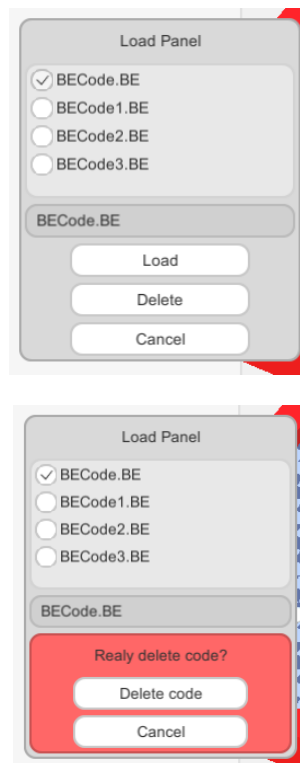
The Save button opens the Save pop-up menu with a list of the saved codes and the possibility of saving with a new name, rewrite a saved file or delete a file.



3.1.2.2 Load Code

The loading of a BE code is done by translating the BE text-based code back to blocks that are going to be placed in the current environment.

The Load code button opens a menu, similar as save menu, with the possibility of choosing a file to load and delete.



3.2 GAME VIEW

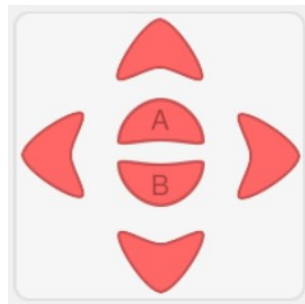
The Game View is composed by the Unity's camera view of the virtual world (2D or 3D objects) and the BE Joystick. The positioning of the objects in the game view area should be adjusted to better fit the screen size. The Target Objects



for the blocks code can be either 2D or 3D Transforms, but mind that the available block instructions need to address the correct Rigidbody and Collider type.

3.2.1 BE JOYSTICK

The BE Joystick is a virtual joystick with four directional buttons (ArrowUp, ArrowLeft, ArrowDown and ArrowRight) and two action buttons (ButtonA and ButtonB) that can individually be mapped as inputs for the Blocks Codebe using the proper event Blocks.

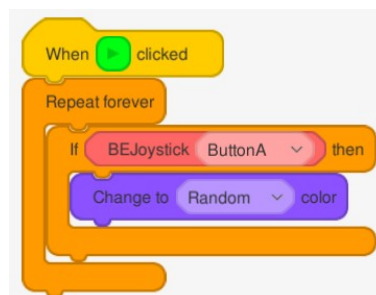


Example: Mapping the BE Joystick ArrowUp as input to move the TargetObject forward:



Is also possible to use the input event as an input for other Blocks, as the follows:

OBS.: Note the button event is only triggered after the Play button is pressed and the infinite loop (Repeat forever) is running.



4 MODEL CLASSES

The model classes define the main objects used by the Engine, the Target Objects and Blocks.

4.1 BE TARGET OBJECT

The Target Object is the object to be manipulated in the scene by the user via blocks code, groups of sequenced blocks. Once a sequence of blocks with a trigger block on top is created and the trigger is activated (via Play button or the correspondent trigger), the cycle of code is started and the instructions related to those blocks will affect the Target Object to make it behave as set in the sequence.

4.2 BE BLOCK

The BEBlock class define the characteristics and rules of the blocks, UI RectTransforms, each visually representing one instruction that can be placed in sequence by the player.

These blocks are related to the correspondent instruction by its GameObject name, that is the same as the class that correspond to the instruction behavior.

4.2.1 BLOCK TYPES

The blocks have an internal type that classifies it by the type of behavior it performs, since it is a visual programming tool, the behavior is also related to its visual layout and drag/drop rules.

4.2.1.1 *Trigger*

Function Blocks that behave as event listeners and are needed for the activation of a block group, being the first blocks of sequence;



4.2.1.2 *Simple*

Function Blocks that are executed a single time per cycle and doesn't wrap children blocks;



4.2.1.3 *Loop*

Function Blocks that execute loop instruction, they wrap children blocks that are going to be executed if the loop condition is still met





4.2.1.4 Conditional

Blocks that execute conditional instruction and wrap children blocks that will be executed if the condition is met.



4.2.1.5 Operation

Blocks that serve as inputs to be placed on the header of Function blocks, they execute operations and return a single string value as result.



4.2.2 BLOCK HEADER

All the blocks have a first child that corresponds to the block's header, containing a descriptive text and the needed inputs for the correspondent behavior. The header's reference is the BlockHeader variable.

As an example, the block that makes the Target Object turn in its axis for a given degrees value have an InputField so the player can set the degrees.



The player can also use other blocks called Operation Blocks on top of the InputField or even combine Operation Blocks.



Another example is the block that makes the Target Object change the color, it has a Dropdown so the player can pick a color.



4.2.3 BE INPUTS

Each block correspondent instruction needs to have the current input values, those values are stored in the BEBlock object under a type called BEInput.

The BEInput have a map for the inputs as string and as float values, respectively stringValues of type string[] and numberValues of type float[], along with a flag, isString, that is true if at least one input value cannot be converted into number.

As an example of a BEBlock object called beBlock that have two inputs, the first input value can be called as beBlock.BeInputs.stringValues[0] or beBlock.BeInputs.numberValues[0]. The second input value then is called beBlock.BeInputs.stringValues[1] or beBlock.BeInputs.numberValues[1].

The already implemented block instructions have examples of the implementations of the block inputs and consider both string and number inputs, so the player can use the blocks in a generic manner.

5 BE CONTROLLER

The BE Controller is the core of the engine, responsible for loop through all the blocks associated with Target Objects. It also contains the methods that start determined trigger blocks, stop all the block groups, activate the next block of the corresponding block group sequence, as well as general methods that can be called by BE Instructions.

5.1 BE INSTRUCTION

BE Instruction is a class part of the controller that defines the possible instructions to be called by the Controller.

An implemented instruction contains one or both BEFunction and BEOperation methods depending on the behavior that needs to be implemented. Those are overridable methods that the controller recognizes in a single manner as instruction.

For determined instruction to be associated with the correspondent blocks, it must be created as a new script of the same name as the block and inheriting from BE Instructions and placed in the Instruction Stack as a component. The BE Instruction uses one or both methods that are called by the controller, those are BEOperation and BEFunction.

See section 5.6.1 on how to create new blocks and associated instructions.



5.2 INSTRUCTION STACK

The BE Controller GameObject (containing the BEController component) have a child GameObject called Instruction Stack.

As the controller is conceptually close to a CPU, the Instruction Stack is close to the ALU, having all the possible instructions (functions and operations) as components, that can be called. Even though the actual calling occurs from a list, for efficiency purposes, this GameObject is always mandatorily active for the block codes to run.

5.3 BE VARIABLES

For the experience of constructing codes with blocks to be as close as possible to an actual coding practice, there is a possibility of using variables as function blocks or setting a variable as an input for a function block.

The BE Variables are accessed by a list of a custom type called BEVariable, that store the name and value of the variable. Also, the controller have already two methods to create/modify a variable, SetVariable(string, string), or retrieve its value, GetVariable(string).

Some blocks using BE Variable list and methods are already built and ready to be used.

5.4 ACCESSIBLE BE LISTS

Along with the BE Variable list, there are two other lists originally implemented as part of the BE Controller for being used in by instructions if needed. Those are, Colors and Sounds lists, they are populated by the controller based on its respective resources folder (Colors and Sounds) and can be used to perform specific instructions such as changing the color of the Target Object or playing a sound.

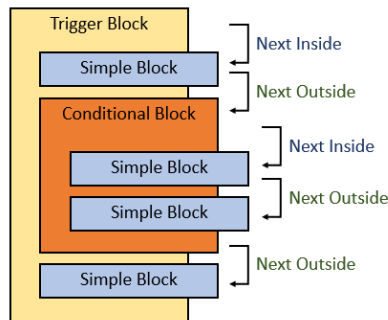
All three lists have a Get method implemented on the controller to be used, GetVariable(string), GetColor(string) and GetSound(string).

5.5 PLAY NEXT METHODS

All the implemented instruction must have a Play Next method, it informs the controller which instruction to play after the actual is finished.

There are two possibilities that can be called by instructions depending on the need, Play Next Outside or Play Next Inside as seen in the scheme below in a simple case of a sequence with a Conditional block with the condition met.





5.5.1 PLAY NEXT OUTSIDE

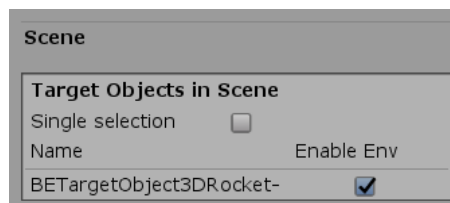
Method called in the end of Simple blocks, in the end of Loop or Conditional blocks or if their conditions are not met.

5.5.2 PLAY NEXT INSIDE

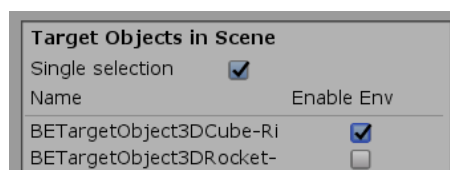
Method called in Trigger blocks and in Loop or Conditional blocks if their conditions are met.

5.6 SCENE (INSPECTOR)

The Scene section, in the BE Controller inspector, is a location to easily identify all the Target Objects currently active in the scene and to enable or disable its Programming Environment.



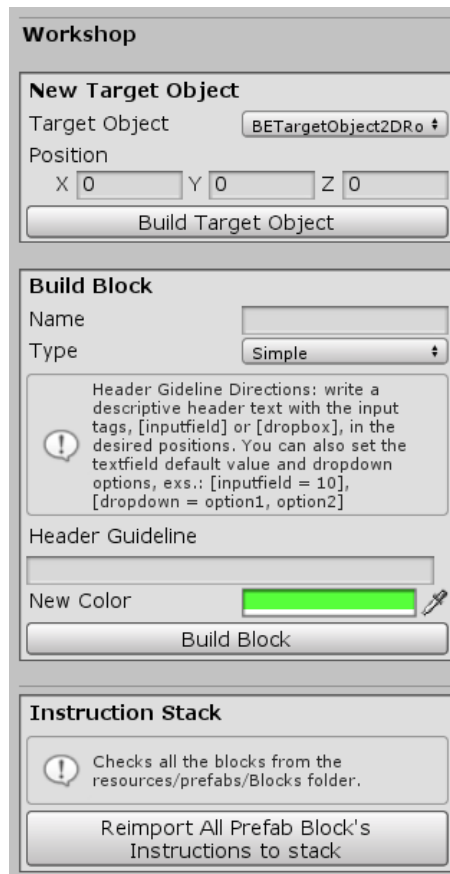
Once you have more objects in the scene (you can add more objects to the scene from the Workshop, explained in the next section), is a good choice to check the “Single selection” option. That way, only one Programming Environment is shown at a time by checking only the Target Object you want to drag blocks to.



5.7 WORKSHOP (INSPECTOR)

At the bottom of the BE Controller inspector there is a section called Workshop, with functionalities to easily create new Target Objects and Blocks just by setting some variables.

The workshop method for creating new Target Objects can also be used to implement a new UI feature for the gameplay, enabling the player to create their own Target Objects.



The Workshop panel is divided into three main sections:

- New Target Object:** Contains a 'Target Object' dropdown menu (currently showing 'BETargetObject2DRo'), 'Position' input fields for X (0), Y (0), and Z (0), and a 'Build Target Object' button.
- Build Block:** Contains a 'Name' text field, a 'Type' dropdown menu (currently showing 'Simple'), a 'Header Guideline' text area with a warning icon and instructions: 'Header Guideline Directions: write a descriptive header text with the input tags, [inputfield] or [dropdown], in the desired positions. You can also set the textfield default value and dropdown options, exs.: [inputfield = 10], [dropdown = option1, option2]', a 'Header Guideline' text field, a 'New Color' color picker (currently showing a green color), and a 'Build Block' button.
- Instruction Stack:** Contains a warning icon and text: 'Checks all the blocks from the resources/prefabs/Blocks folder.', and a button labeled 'Reimport All Prefab Block's Instructions to stack'.

OBS.: When creating more than one Target Object in the scene, mind that you will have to deal with one Programming Environments per target. You can add an UI feature for choosing the Target Object and exhibit only one Programming Environment at a time.

5.7.1 BUILDING A BLOCK AND RELATED INSTRUCTION

There are some directions to be followed so you can add new blocks in the Engine and raise the coding possibilities.

To understand how to add new blocks, we are going to follow the steps of creation and implementation of the instruction script of a new block for a chess knight movement.



Later you can also see the already implemented Instructions scripts for guidance on the creation of different type blocks.

5.7.1.1 Step 1: Plan the behavior and choose the block type

We want to use this new block to make the Target Object to perform a chess knight like movement, meaning it needs to move two steps up, down, right or left and one step perpendicular to the last move. For learning purposes, we will also include the possibility of repeating this movement N times.

For our explained behavior, we need the player to provide two inputs so the instruction knows where and how many times the Target Object will move. These input interfaces will be a Dropdown for the direction (options: up-right, up-left, down-right, down-left, right-up, right-left, left-up and left-down) and an InputField for setting the number of repetitions.

Even though the base movement is repeated for N times, the whole instruction (L movement for N times) is called only once per cycle, so our block will be a Simple Block.

In short, the behavior is: Move the Target Object in L movement to a given direction for a given number of times.

5.7.1.2 Step 2: Create the Block from BE Controller Workshop

Once we have in mind our block type and behavior rules, we can use the Build Block function on the BE Controller's Workshop to automatically create our needed objects into the Unity's scene.

- **Choose a descriptive name for the block**

We need to set up a name for our block GameObject, that will also be the same as the Instructions' script name. We will choose "ChessKnightMove", since it is a descriptive name easily associated with the behavior.

- **Choose the Block Type**

Since the behavior we want to create will execute a single time per cycle and is not a loop or conditional, we choose the "Simple" type.

- **Write the Header Guideline**

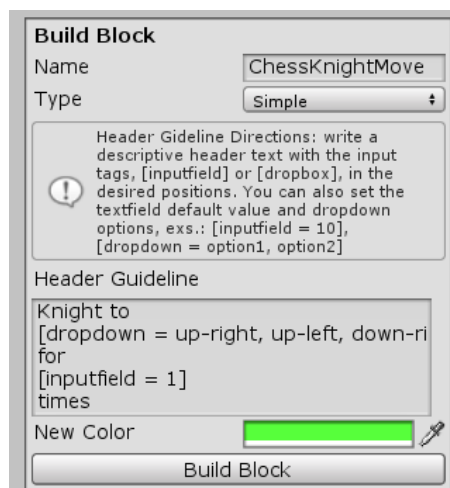
To correctly create the block header, we must set up the header guideline string with the text we want to exhibit on the block and the input types in the correct positions.



Following the field directions given in the info box, we write a comprehensive description of the behavior followed by the input tags + values as follow:

```
Knight to  
[dropdown = up-right, up-left, down-right, down-left, right-up, right-down, left-up, left-down]  
for  
[inputfield = 1]  
times
```

Make sure to let each part of the header in a single line. We passed the values for the Dropdown and a default value for the InputField to facilitate the process, but it is always possible to change the values in the object's inspector or not to pass any default value and set them manually later.



- **Set a Color for the block**

The block color is a visual helper to better organize the blocks in categories. Since our block have a movement category behavior, we could choose blue as the other movement blocks, however we will set it to a tone of Green, as a new chess movement category.

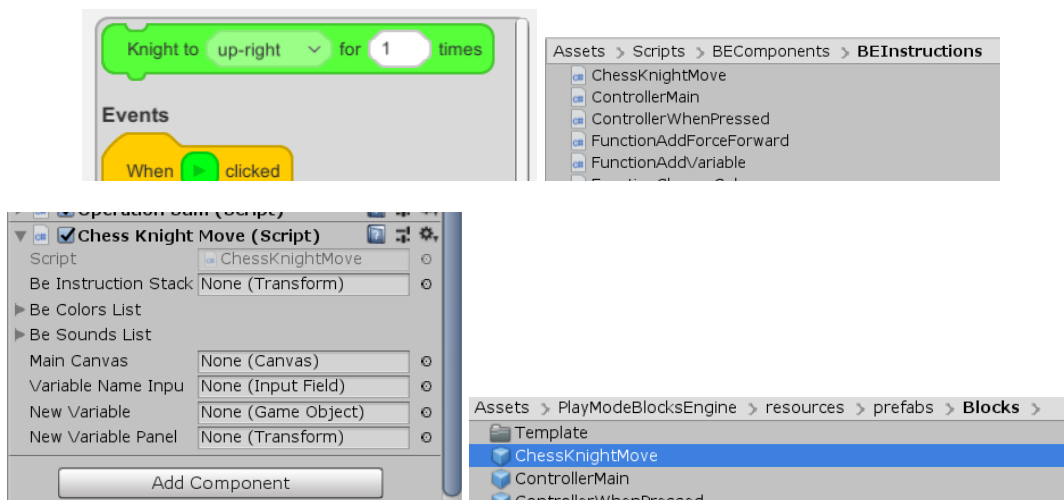
- **Create the block and wait the instruction to be inserted in the stack**

Once you hit the “Build Block” button, you will be shown the message “Block and Instruction created. Trying to import Instruction to stack...”, wait until the message disappears and we are done with this step.

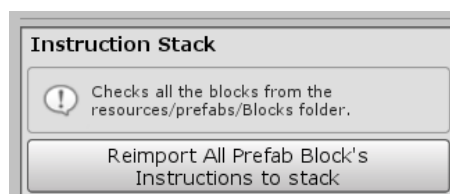




The result should be the new block appearing on the Blocks Scroll View, the new script in the BE Instructions scripts folder, the instruction added as component to the Instruction Stack and a new prefab in the resources blocks folder ("resources/prefabs/Blocks").



OBS.: If the process takes too long you can stop the importing and add the newly created script from "Scripts/BEComponents/BEInstructions", to the Instruction Stack GameObject that is a child of the BE Controller or reimport all the instructions pressing the button "Reimport All Prefab Block's Instructions to stack".



5.7.1.3 Step 3: Implement the Instruction Script

Right now, the block can already be viewed and dragged around the UI but has no effect on the Target Object yet. To finish our creation process, we are going to implement the instruction code.

The newly created script has already the needed template for start coding our instruction behavior.

```
using UnityEngine;
using System.Collections;

public class ChessKnightMove : BEInstruction
{
    // Use this for Operations
    public override string BEOperation(BETargetObject targetObject, BEBlock
beBlock)
    {
        string result = "0";

        // Use "beBlock.BeInputs" to get the input values

        return result;
    }

    // Use this for Functions
    public override void BEFunction(BETargetObject targetObject, BEBlock beBlock)
    {
        // Use "beBlock.BeInputs" to get the input values

        // Make sure to end the function with a "BeController.PlayNextOutside"
method and use "BeController.PlayNextInside" to play child blocks if needed
        BeController.PlayNextOutside(beBlock);
    }
}
```

First we need to know only few concepts, the targetObject variable is the scene object we are going to move as the knight, the beBlock is our just created block that have the player inputs and BEFunctions always ended with a PlayNextOutside method so the Controller can correctly call the instructions (See the Loop and Conditional Blocks instructions for examples on how to use the PlayNextInside method).

In our desired case we are going to delete or let the BEOperation aside and implement only the BEFunction method.

We have a structured data that comes from the Dropdown, having a first direction a separator and a second direction. We, then, implement a support method to get the direction based on a string.

```
// Support method to get a direction based on the passed string
Vector3 GetDirection(string option)
{
    // Switch statements based on the string value
    switch (option)
    {
        case "up":
            return Vector3.forward;
        case "down":
            return Vector3.back;
        case "right":
```



```

        return Vector3.right;
    case "left":
        return Vector3.left;
    default:
        return Vector3.zero;
    }
}

```

We then get the block first input using “beBlock.BeInputs.stringValues[0]” (review section 4.2.3 on how the inputs can be used), treat it using the separator and use the previously created method to get the directions we are going to use later.

```

// Get the string value from the first block input
string option = beBlock.BeInputs.stringValues[0];
// Separate the dropdown value by the '-' separator
string[] options = option.Split('-');

// Set the first and second movement directions
Vector3 firstMoveDirection = GetDirection(options[0]);
Vector3 secondMoveDirection = GetDirection(options[1]);

```

With our directions set, we then transmit them to the Target Object, wrap the whole movement in a loop that repeats the amount of times given by the Player in the second Block input, “beBlock.BeInputs.numberValues[1]” and finally end the function with the mandatory PlayNextOutside method.

```

// Repeat the whole movement based on the InputField value
for (int i = 0; i < beBlock.BeInputs.numberValues[1]; i++) {
    // Repeat first movement 2 times
    for (int j = 0; j < 2; j++)
    {
        targetObject.transform.position = targetObject.transform.position +
        firstMoveDirection;
    }
    // Do second movement 1 time
    targetObject.transform.position = targetObject.transform.position +
    secondMoveDirection;
}

// End the function
PlayNextOutside(beBlock);

```

The complete script will look like the code below.

```

using UnityEngine;
using System.Collections;

// Instant knight movement
public class ChessKnightMove : BEInstruction
{
    // Use this for Functions
    public override void BEFunction(BETargetObject targetObject, BEBlock beBlock)
    {
        // Get the string value from the first block input
        string option = beBlock.BeInputs.stringValues[0];
        // Separate the dropdown value by the '-' separator
        string[] options = option.Split('-');

        // Set the first and second movement directions
        Vector3 firstMoveDirection = GetDirection(options[0]);
        Vector3 secondMoveDirection = GetDirection(options[1]);

        // Repeat the whole movement based on the InputField value
    }
}

```



```

        for (int i = 0; i < beBlock.BeInputs.numberValues[1]; i++) {
            // Repeat first movement 2 times
            for (int j = 0; j < 2; j++)
            {
                targetObject.transform.position = targetObject.transform.position +
firstMoveDirection;
            }
            // Do second movement 1 time
            targetObject.transform.position = targetObject.transform.position +
secondMoveDirection;
        }

        // End the function
        BeController.PlayNextOutside(beBlock);
    }

    // Support method to get a direction based on the passed string
    Vector3 GetDirection(string option)
    {
        // Switch statements based on the string value
        switch (option)
        {
            case "up":
                return Vector3.forward;
            case "down":
                return Vector3.back;
            case "right":
                return Vector3.right;
            case "left":
                return Vector3.left;
            default:
                return Vector3.zero;
        }
    }
}

```

You can later add a timer, so the Target Object does not jump instantly to the last position, or even use the Rigidbody methods to make the movement smoother, as the following suggestion code.

```

using UnityEngine;
using System.Collections;

// Smooth Knight movement
public class ChessKnightMove : BEInstruction
{
    int counterForRepetitions;
    float counterForMovement = 0;
    float movementDuration = 0.3f; //seconds
    Vector3 startPos;
    int moveSelection = 0;

    public override void BEFunction(BETargetObject targetObject, BEBlock beBlock)
    {
        if (beBlock.beBlockFirstPlay)
        {
            counterForRepetitions = (int)beBlock.BeInputs.numberValues[1];
            beBlock.beBlockFirstPlay = false;
        }

        // [0] first input (dropdown)
        // options: up-right, up-left, down-right, down-left, right-up, right-down,
        left-up, left-down
        string option = beBlock.BeInputs.stringValues[0];
        string[] options = option.Split('-');

        //movements
        Vector3 firstMovement = GetDirection(options[0]);
        Vector3 secondMovement = GetDirection(options[1]);
        Vector3[] movements = new[] { firstMovement, firstMovement, secondMovement };
    }
}

```



```
        if (counterForMovement == 0)
        {
            startPos = targetObject.transform.position;
        }

        if (counterForMovement <= movementDuration)
        {
            counterForMovement += Time.deltaTime;
            targetObject.transform.position = Vector3.Lerp(startPos, startPos +
movements[moveSelection], counterForMovement / movementDuration);
        }
        else
        {
            moveSelection++;
            counterForMovement = 0;
        }

        if(moveSelection == 3)
        {
            moveSelection = 0;
            counterForMovement = 0;
            counterForRepetitions--;

            if (counterForRepetitions <= 0)
            {
                beBlock.beBlockFirstPlay = true;
                BeController.PlayNextOutside(beBlock);
            }
        }

    }

    private Vector3 GetDirection(string option)
    {
        switch (option)
        {
            case "up":
                return Vector3.forward;
            case "down":
                return Vector3.back;
            case "right":
                return Vector3.right;
            case "left":
                return Vector3.left;
            default:
                return Vector3.zero;
        }
    }
}
```

Feel free to improve the code as much as you want and ask questions about your new block and instruction ideas as well as new functionalities for the Blocks Engine.

Now we can test the block in Play Mode!



Play Mode Blocks Engine – Version 1.2

