

Add **Programming Blocks** to your Games with **Blocks Engine 2**. Friendly and familiar, based on the most popular block programming languages. Perfect for Educational, puzzle or any type of Game.

This Blocks Engine 2 asset documentation contains the detailed explanations for the implemented features, classes, methods, variables, concepts and setting adjustments.

Contact



[Asset Store Page](#)



[Github](#)



support@meadowgames.com



MeadowGames
MeadowGames.com

CONTENTS

Overview	4
User Interface	5
Blocks Selection Panel	5
Categories	6
Programming Environment	6
Saving and Loading Code	7
Blocks	7
Game Scene	9
Virtual Joystick	9
BE2 Front Components	11
BE2 Programming Environment	11
BE2 Target Object	11
BE2 Block	11
Sections	11
Inputs	11
BE2 Core Components	13
Instruction	13
Execution Methods	13
Properties and Get Methods	13
Stack Pointer Call Methods	14
Blocks Stack	15
Execution Manager	15
Variables Manager	15
Block Serializer	15
Events Manager	16
BE2 Inspector	17
Block Builder	17



Building a New Block (LookAtAndMove)	17
Updating the LookAtAndMove Block Using Lerp	25
Template Block Parts	27
Paths Setting	27
Extras	29
Adding Icons/Images on the Block Header	29
Adding a New Category in the Blocks Selection Panel	30
Adding a Custom Target Object	31



1. OVERVIEW

The Blocks Engine 2 is a complete project that adds an engine for visual block programming and interface to your games. It comes with 30+ blocks, interface for selecting blocks by category, interface for programming, variable viewer and creator, blocks code serializer (save/load), events manager and a 3D environment for playing with the blocks code.

The engine expands the game mechanics possibilities and gives the players the possibility to code their own player movements and learn coding and valuable programming concepts by playing with a friendly tool.

The project can be used to create educational, code learning games, puzzles or even combat games where players need to build their own code to control objects in the scene using visual blocks programming.

The Blocks Engine 2, is an upgrade from the former Play Mode Blocks Engine. It is more powerful, faster, easily customizable and built to allow even more block possibilities, still keeping the layout and rules similar to the most popular block programming languages.

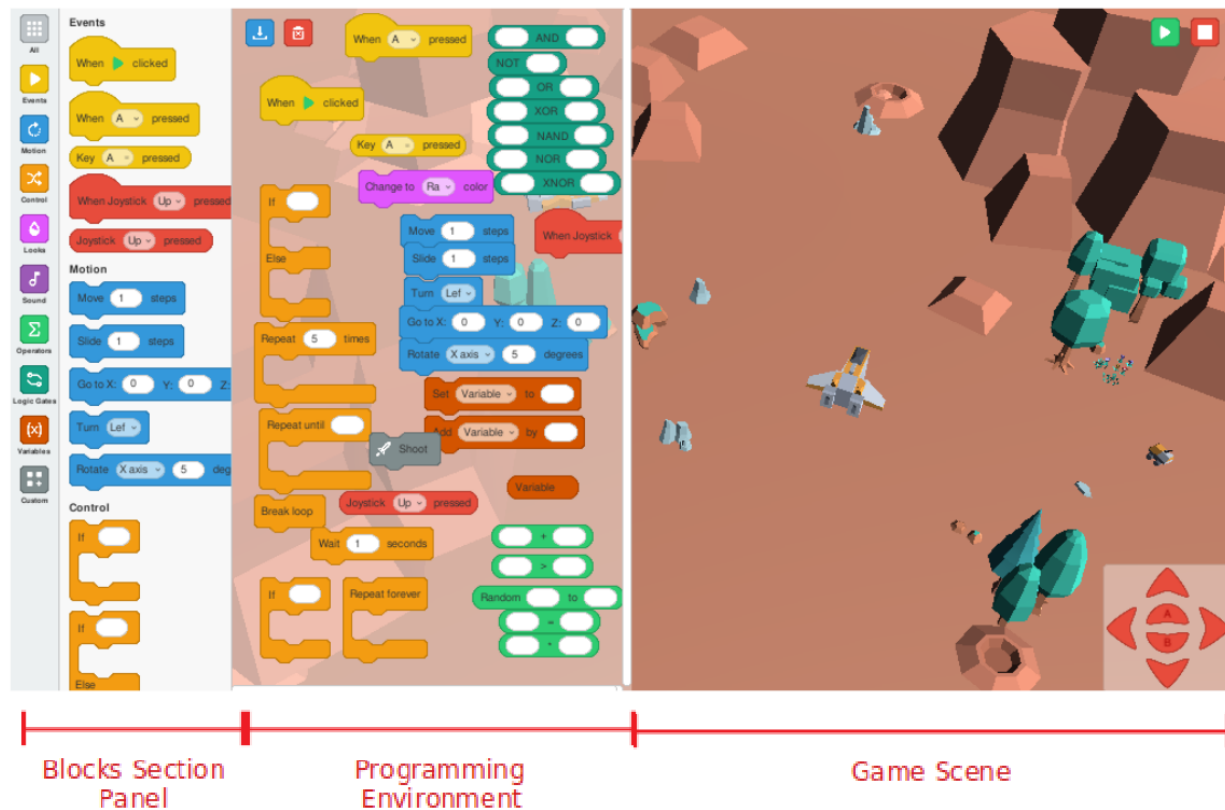
The engine makes the integration between visual block programming and the power of the Unity engine, including the possibility to deploy for multiple platforms.

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



2. USER INTERFACE

From the start you can play with the blocks and feel the environment, the project is ready out of the box. The user interface contains the Blocks Selection Panel, Programming Environment and the Game Scene. Below is a print of the sample scene.



The user interface can be customized to fit your unique project by changing the layout, size, position and other visual characteristics, as well as expanding the functionalities and adding game mechanics.

It is also possible to create more blocks and instructions and customize the blocks characteristics.

2.1. BLOCKS SELECTION PANEL

The Block Selection Panel is where the available blocks are provided and organized in categories to the user.

These blocks have the same appearance to the instruction blocks used in the Programming Environment, but these GameObjects contain different components that allow the user to drag new blocks from them to be placed in the Programming Environment.

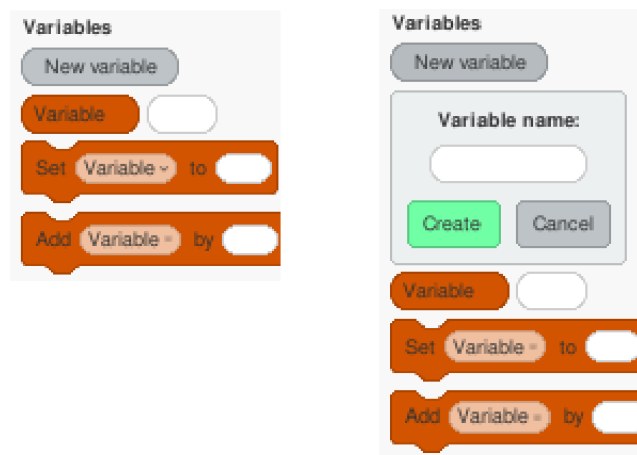


2.1.1. CATEGORIES

Each category of the Block Selection View is related to the general behavior of the instruction.

- **Events:** Trigger blocks that start a sequence of blocks and operation blocks that function as event listeners. Colors #F1C40F and #E74C3C;
- **Motion:** Blocks related to movement, positioning and direction of the Target Object. Color #3498DB;
- **Control:** Blocks that can alter the run cycle, repeating the cycle of a blocks group, delaying or conditioning the execution of specific blocks group. Color #F39C12;
- **Looks:** Blocks that alter the visual aspect of the Target Object. Color #E056FD;
- **Sound:** Blocks that execute sounds. Color #9B59B6;
- **Operators:** Operation blocks that perform math and comparative operations used as inputs. Color #2ECC71;
- **Logic Gates:** Operation blocks that perform logic operations used as inputs. Color #16A085;
- **Variable:** Blocks that perform variable related operations, act on variables, variable manager and the variables (operations) used as inputs. Color #D35400;
- **Custom:** Additional blocks later created and blocks that act on specific Target Objects. Color #7F8C8D.

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.



2.2. PROGRAMMING ENVIRONMENT

The Programming Environment is the area used to build the Blocks Code. Its component has a reference to the Target Object which the code will take effect.

The environment contains two buttons, the arrow down opens up a panel for saving and loading the Blocks Code of the environment; and the bin clears the environment, destroying all the blocks in it.



It is possible to have multiple Target Objects in the scene, and for that it is recommended that each should be related to its own Programming Environment.

2.2.1. SAVING AND LOADING CODE

The saving of a Blocks Code is done by translating the Blocks into XML strings and storing them as a single .BE2 file. This process is done using the BE2_BlocksSerializer class.

The XML was chosen to serialize due to a depth limitation on Unity's JsonUtility, however, it is possible to use another Json alternative.

The Load process is the inverse, also using the serializer, the reading of the .BE2 file and translation back into Block GameObjects, that is instantiated to the Programming Environment.

The folder where the saved Codes are located is originally the "BlocksEngine2/Saves", but it is possible to change it to be the persistent data directory (Application.persistentDataPath).

From the Programming Environment, the saving and loading is done by the menu shown below.



2.2.2. BLOCKS

The blocks are the visual representation of the code functions, operations or variables. The user can place the function blocks in a logic sequence to build a code and insert operation blocks (and variables) as inputs in other blocks.

There are five block types (Trigger, Simple, Loop, Condition and Operation), those types can be identified by the shape of the block and for its behavior. All types except Operation are Function blocks.



Trigger

Blocks that begin each program, sequence of blocks and can behave as event listeners. They indicate how and when the program will start running.



Simple

These blocks are usually responsible for the main actions of the program and execute its function once per cycle, they don't wrap children blocks.



Loop

Blocks that execute loop instruction, they wrap children blocks that are going to be executed if the loop condition is still met. These blocks can have multiple sections, as an example of the If/Else block.

This type of block is by default executed once per frame due to their repetitive behavior, meaning that its ExecuteInUpdate setting of its Instruction is always true.



Conditional



Blocks that execute conditional instruction and wrap children blocks that will be executed if the condition is met. They also can have multiple sections.



Operation

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



2.3. GAME SCENE

The Game Scene is where the code results are exposed to the user. In a 2D or 3D scenario, the Target Object performs the code instructions and can interact with other environment elements.

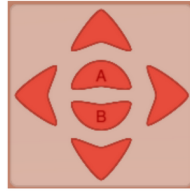
A virtual joystick is also present in that view, it can be used in composition with specific blocks for listening to the buttons.

There are a great number of possibilities for composing the Game Scene and letting your creativity flow, some examples are coding tutorials, puzzles, top down, combat, racing, coin catch games.

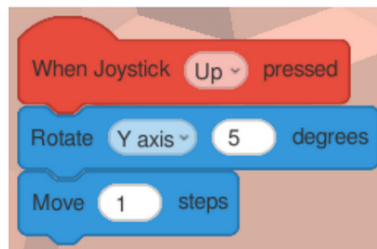
2.3.1. VIRTUAL JOYSTICK

The virtual joystick contains directional buttons (ArrowUp, ArrowLeft, ArrowDown and ArrowRight) and two action buttons (ButtonA and ButtonB) that can be mapped as inputs for the Blocks Code by using the proper Blocks.

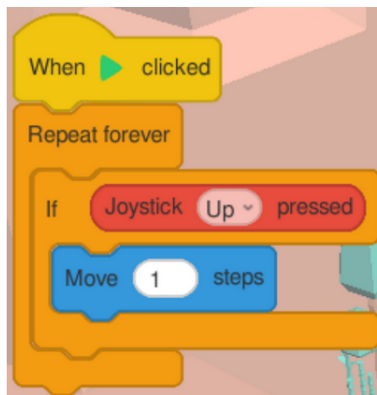




Example: Mapping the BE Joystick ArrowUp as input to move the Target Object forward:



It is also possible to listen to the buttons using the correspondent Operation Block as input:



3. BE2 FRONT COMPONENTS

The front components have a direct relation to the user actions and define the visual layout, behavior and relationships between themselves and the core components.

3.1. BE2 PROGRAMMING ENVIRONMENT

The Programming Environment inherits from the **I_BE2_ProgrammingEnv** interface, it is the link between the Blocks and the Target Object, it indicates that the code given by the blocks on a specific Programming Environment should act in the corresponding Target Object.

3.2. BE2 TARGET OBJECT

The Target Object inherits from the **I_BE2_TargetObject** interface, it is the object that will reflect the blocks code results. This object is viewed in the Game Scene along with the game environment, where it can be commanded, based on the blocks code, to move along, change or perform tasks.

3.3. BE2 BLOCK

The Blocks classes inherit from the **I_BE2_Block** interface, they define the logic and visual structure of the blocks, having a reference to its layout and instruction (function or operation).

These blocks are related to the corresponding instruction by its Instruction component and its GameObject name.

3.3.1. SECTIONS

The block sections are composed of a header and an optional body, each of the section's header has also inputs that can be used in the Instructions scripts.

In case of Trigger, Simple and Operation blocks, there is only one section and its header. The Loop and Condition blocks can have more than one section (as the If/Else block) and the section body is present so other blocks can be dropped into it.

3.3.2. INPUTS

The block inputs inherit from the **I_BE2_BlockSectionHeaderInput** interface, they can be accessed from its Instruction using Section0Inputs or GetSectionInputs(int).



The inputs used by the block instructions (by default, Inputfields, Dropdowns and Operation blocks) are returned as **BE2_InputValues** type, which contains the actual value as float and string.



4. BE2 CORE COMPONENTS

The core components of the engine are responsible for managing the execution of the blocks code, update the needed core classes and return the results to the front components and objects (as return the code results to make the Target Object perform the instructions).

4.1. INSTRUCTION

The Instruction component makes use of the **I_BE2_Instruction** interface. It is where the block logic is implemented.

Each block has its own Instruction containing the execution method (Function or Operation) that will be called during the run cycle.

Depending on the block type its logic should be implemented either in the Function method (for non-operation blocks) or the Operation method (for operation blocks). The implementation is usually done by accessing some components of the base class, its properties and get method (TargetObject, Section0Inputs, GetSectionInputs and ExecuteInUpdate), stack pointer call methods (ExecuteSection and ExecuteNextInstruction) and additional event methods.

For the instruction to be associated with a Block, it is added as a component to the Block's GameObject.

4.1.1. EXECUTION METHODS

The implementation of one of the execution methods is mandatory for the instruction to work properly. For non-operation blocks (trigger, simple, condition and loop), the "Function" method should be implemented, while the operation blocks have the "Operation" method implemented.

void Function() – Used to implement the logic of trigger, simple, condition and loop blocks.

string Operation() – Used to implement the logic of operation blocks.

4.1.2. PROPERTIES AND GET METHODS

I_BE2_TargetObject TargetObject - Reference to the Target Object.

I_BE2_BlockSectionHeaderInput[] Section0Inputs – Return an array with all the inputs of the block's first section.



`I_BE2_BlockSectionHeaderInput[] GetSectionInputs(int sectionIndex)` – Return an array with all the inputs on the indicated section.

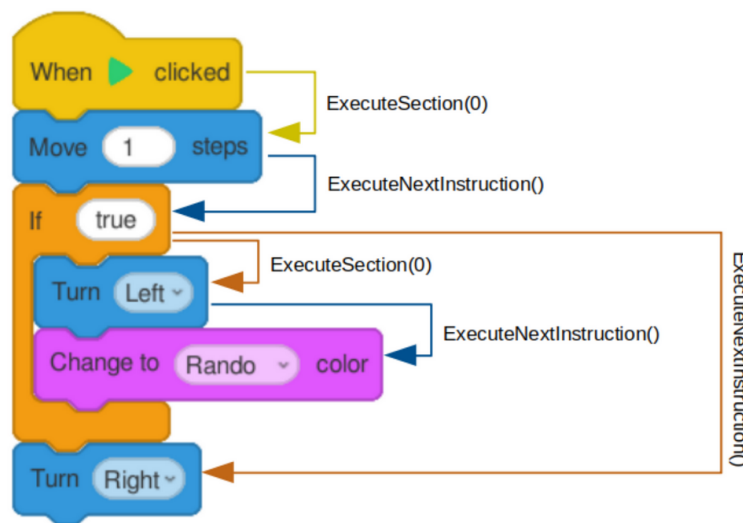
`bool ExecuteInUpdate` – Default value is false. Set true to force the Instruction to be called in the update method of the Execution Manager. This setting should be set true for blocks that contain a condition to finish, causing it to be called repeatedly, ex.: loop blocks, wait, lerp (block slide forward).

4.1.3. STACK POINTER CALL METHODS

These methods indicate the next instruction that will be called based on the blocks sequence, those are, `ExecuteSection(int)` and `ExecuteNextInstruction()`.

The non-operation blocks should use one or both pointer call methods to direct the execution of the instructions.

The scheme below has a simple scenario to illustrate when these methods should be called and which instruction is going to be executed next:



`void ExecuteSection(int sectionIndex)` - Method that makes the stack point to the first child block's instruction of the indicated section. Commonly called in the instruction of Trigger, Loop and Conditional blocks.

`void ExecuteNextInstruction()` - Method that makes the stack point to the next block's instruction in the same level as the current instruction. Commonly called in instruction of Simple, Loop and Conditional blocks.



4.2. BLOCKS STACK

The Block Stack makes use of the **I_BE2_BlockStack** interface. It is a component present only in the Trigger blocks and serves as a stack for the subsequent instructions.

The stack is used by the Execution Manager to fetch the current instruction and call the execution method.

4.3. EXECUTION MANAGER

The Execution Manager, **BE2_ExecutionManager** class, contains the reference for all the available Blocks Stacks and is responsible for calling their current instructions during the run cycle.

This component lives in a single GameObject in the scene.

4.4. VARIABLES MANAGER

The Variables Manager, **BE2_VariablesManager** class, is an essential part for improving the user experience on using the Blocks Engine 2 and make it a complete coding practice by having the possibility of using Variable operation blocks as inputs and having Variable manipulation blocks.

void AddOrUpdateVariable(string variable, string value) – Adds a new variable and sets its value, or, if the variable already exists, updates its value.

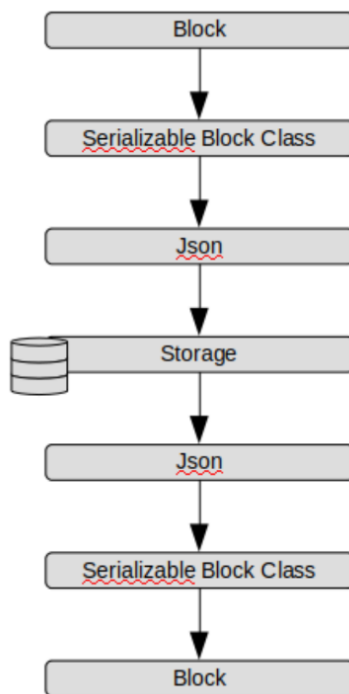
BE2_InputValues GetVariableValues(string variable) – Get the variable values as BE2_InputValues type, which contains the actual result in both string or float types.

4.5. BLOCK SERIALIZER

The Block Serializer is used by the system to save and load blocks code as well as to duplicate blocks in the Programming Environment.

The serialization/deserialization processes are given in the scheme below:





4.6. EVENTS MANAGER

The Events Manager, **BE2_EventsManager** class, aggregates all the needed BE2 events to help on the components decoupling.

It contains two types of events:

- **BE2EventTypes**, that are subscribed to UnityEvent without parameters. Currently enumerated as the following IDs: OnPlay, OnStop, OnDrag, OnPointerUpEnd, OnAnyVariableValueChanged, OnAnyVariableAddedOrRemoved, OnBlocksStackArrayUpdate.

and

- **BE2EventTypesBlock**, that are subscribed to UnityEvent<I_BE2_Block>, used for events that happen per block. Currently enumerated as the following IDs: OnStackExecutionStart, OnStackExecutionEnd.

It is possible to subscribe to these events or add new events if needed.



5. BE2 INSPECTOR

The BE2 Inspector editor provides a direct interface to simplify some customization tasks, currently it provides a way to facilitate the creation of new Blocks and the setting of needed Paths. It is accessed from the BE2 Inspector GameObject's inspector panel

5.1. BLOCK BUILDER

The Blocks Engine 2 Block Builder is cleaner and more direct than the later versions, therefore, the builder serves only the main purpose of facilitating the creation of custom blocks.

The creation of blocks is done by the setting of variables and the use of the custom header markup to indicate the block header items.

Mind that after the creation of the block, the correspondent instruction should be implemented using C# and making use of the recommended instruction API explained in section 4.1.

There are 3 steps to be followed so you can add new blocks in the Engine. These are simple steps and for a better understanding, the next sections will explain in detail these steps and guide you through the creation process of a custom block that performs a composed movement (look at direction and move), from using the Block Builder to the Instruction implementation.

5.1.1. BUILDING A NEW BLOCK (LOOKATANDMOVE)

The composed movement the Target Object will perform depends on at least two inputs from the user, we are going to use both types of inputs (text and dropdown) as well as use these input results in the instruction script.

The complete instruction script is also commented to serve as a guide for the next block instructions you implement.

Step 1: Understanding the Block Behavior

The new block will make the Target Object perform a composite movement, first facing a direction then moving forward, for that, there is a need for at least two inputs, and we are going to use first a dropdown with predefined directions (Up, Down, Left, Right) and a text field for entering a number value.

Step 2: Creating the Block With the Inspector



Once we have in mind our block type and behavior rules, we can go to the Block Builder Inspector.

Choose a descriptive name for the block

The chosen name will also be used to organize the instructions scripts, then, "LookAtAndMove" is descriptive enough.

Choose the Block Type

The whole instruction is called only once per cycle, therefore, the block type Simple will be our choice.

Set a Color for the block

The block color is a visual helper to better organize the blocks in categories.

Write the Block Header Markup

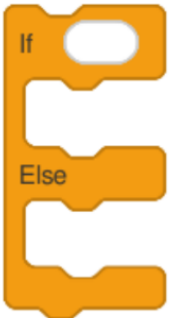
Before writing the Header Markup for our new block, we need to understand how it works.

The Block Header Markup is an essential field to indicate which items the header, or headers, will contain. It is written using a simplified markup to indicate when a header item is a text input, using the string "\$text", or a dropdown, using the string "\$dropdown".

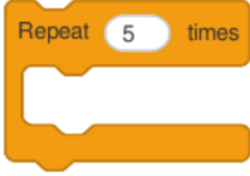


Each line of this field will correspond to a header, meaning that if the built block is of type simple, trigger or operation, it should have only one header, while condition and loop might have more than one headers, each related to a block section.

If the header markup contains any input, below the text area will be shown a text field for each input to be entered the default value or the dropdown items.

Below there are some examples of blocks with the related header markup:

Block	Header Markup	\$Inputs
	<p>If \$text</p> <hr/> <p>Else</p> <hr/>	<p>\$input0:</p> <hr/>



	Repeat \$text times	\$input0 : 5
	Rotate \$dropdown \$text degrees	\$input0 : X axis, Y axis, Z axis \$input1 : 5
	Joystick \$dropdown pressed	\$input0 : Up, Down, Left, Right

Continuing the creation of our new block, the Header Markup and Inputs will be:

Header Markup	\$Inputs
Look at \$dropdown and move \$text steps	\$input0 : Up, Down, Left, Right \$input1 : 1

BE2_Inspector (Script)

Block Builder

Instruction Name

LookAtAndMove

Block Type

Simple

Block Color

Block Header Markup

Write the header text and inputs in a single line. Additional headers in new lines.
possible input types are \$text or \$dropdown

Look at \$dropdown and move \$text steps

Below you can define the text \$input default value or \$dropdown option values separated by comma.

\$input 0

Up, Down, Left, Right

\$input 1

1

Build Block

Press “Build Block”

Once you hit the “Build Block” button the system do a sequence of automatic procedures to:



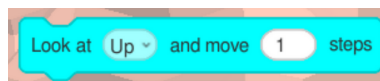
1. Create a new block, add the header, add the needed components and instantiate the new block;
2. Create the new instruction from the script template, refresh the asset database and recompile;
3. Add the new instruction to the block;
4. Add the new block to the selection menu and create its prefab.

The result after the process should be the following log messages on the console:

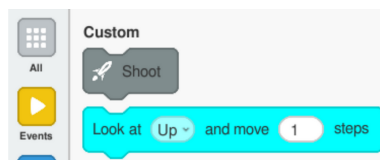
```
+ Block created
+ Start creating instruction
+ End creating instruction
+ Instruction added
+ Block added to selection menu
+ Block prefab created
```

You can verify if all was correctly created:

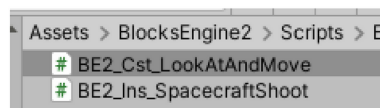
1. The block should be in the Programming Environment



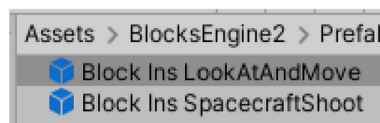
2. The block selection panel (Custom section) should contain the block



3. The Instruction should be in the project folder "BlocksEngine2/Scripts/EngineCore/Instruction/BlockInstructions/Custom"

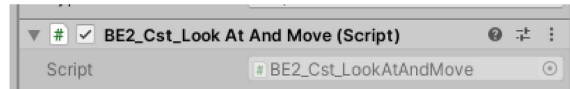


4. There should be a prefab for the block in the project folder "BlocksEngine2/Prefabs/Resources/Blocks/Custom"



5. The new block should have the Instruction added as component





Step 3: Implementing the Instruction Script

The block is almost ready to be used in the codes, its instruction has no effect on the Target Object yet. To finish our creation process, we are going to implement the instruction code.

The new instruction is created using a template that contains all the needed items for you to code the instruction behavior as well as comments explaining it all.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class BE2_Cst_LookAtAndMove : BE2_InstructionBase, I_BE2_Instruction
{
    // ► Refer to the documentation for more on the variables and methods

    // ### Execution Methods ###

    // --- Method used to implement Function Blocks (will only be called by types: simple,
    // condition, loop, trigger)
    public void Function()
    {
        // --- use Section0Inputs[inputIndex] to get the Block inputs from the first section
        // (index 0).
        // --- Optionally, use GetSectionInputs(sectionIndex)[inputIndex] to get inputs from a
        // different section
        // --- the input values can be retrieved as .StringValue, .FloatValue or .InputValues //
        Section0Inputs[inputIndex];

        // ### Stack Pointer Calls ###

        // --- execute first block inside the indicated section, used to execute blocks inside this
        // block (ex. if, if/else, repeat)
        //ExecuteSection(sectionIndex);

        // --- execute next block after this, used to finish the execution of this function
        ExecuteNextInstruction();
    }

    // --- Method used to implement Operation Blocks (will only be called by type: operation)
    public string Operation()
    {
        string result = "";

        // --- use Section0Inputs[inputIndex] to get the Block inputs from the first section
        // (index 0).
        // --- Optionally, use GetSectionInputs(sectionIndex)[inputIndex] to get inputs from a
        // different section
        // --- the input values can be retrieved as .StringValue, .FloatValue or .InputValues //
```



```

        Section0Inputs[inputIndex];

        // --- operation results are always of type string.
        // --- bool return strings are usually "1", "true", "0", "false".
        // --- numbers are returned as strings and converted on the input get.
        return result;
    }

    // ### Execution Setting ###

    // --- Use ExecuteInUpdate for functions that plays repeatedly in update, holding the
    blocks stack execution flow until completed (ex.: wait, lerp).
    // --- Default value is false. Loop Blocks are always executed in update (true).
    //public bool ExecuteInUpdate => true;

    // ### Additional Methods ###

    // --- executed after base Awake
    //protected override void OnAwake()
    //{
    //
    //
    //}

    // --- executed after base Start
    //protected override void OnStart()
    //{
    //
    //
    //}

    // --- Update can be overridden
    //void Update()
    //{
    //
    //
    //}

    // --- executed on button play pressed
    //protected override void OnButtonPlay()
    //{
    //
    //
    //}

    // --- executed on button stop pressed
    //protected override void OnButtonStop()
    //{
    //
    //
    //}

    // --- executed after blocks stack is populated
    //public override void OnPrepareToPlay()
    //{
    //
    //
    //}

    // --- executed on the stack transition from deactive to active
    //public override void OnStackActive()
    //{
    //
    //
    //}
}

```



Since we are implementing a relatively simple instruction and it is a non-operation block, we will only use the Function() method, therefore, to better view our code, we can erase the Operation() method and all the additional methods. The Execution setting (ExecuteInUpdate) is by default set false and we can also erase it from the script, since the block will only run once per cycle.

The result should be:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class BE2_Cst_LookAtAndMove : BE2_InstructionBase, I_BE2_Instruction {
    public void Function()
    {
        // --- use Section0Inputs[inputIndex] to get the Block inputs from the first section (index
        // 0).
        // --- Optionally, use GetSectionInputs(sectionIndex)[inputIndex] to get inputs from a
        // different section
        // --- the input values can be retrieved as .StringValue, .FloatValue or .InputValues //
        // Section0Inputs[inputIndex];

        // ### Stack Pointer Calls ###

        // --- execute first block inside the indicated section, used to execute blocks inside this
        // block (ex. if, if/else, repeat)
        //ExecuteSection(sectionIndex);

        // --- execute next block after this, used to finish the execution of this function
        ExecuteNextInstruction();
    }
}
```

We have a string value that comes from the dropdown, so we implement a support method to get the direction based on a string.

```
Vector3 GetDirection(string option)
{
    // returns the look direction 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 will use this method to get the direction which the Target Object should face, then use the second input as float to translate it forward.

```
// Section0Inputs[0].StringValue is used to get the dropdown value as string
Vector3 direction = GetDirection(Section0Inputs[0].StringValue);
TargetObject.Transform.rotation = Quaternion.LookRotation(direction);

// Section0Inputs[1].FloatValue is used to get the inputfield value as float float
steps = Section0Inputs[1].FloatValue;
TargetObject.Transform.position += TargetObject.Transform.forward * steps;
```

The **complete instruction** script is given below.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class BE2_Cst_LookAtAndMove : BE2_InstructionBase, I_BE2_Instruction {
    public void Function()
    {
        // Section0Inputs[0].StringValue is used to get the dropdown value as string
        Vector3 direction = GetDirection(Section0Inputs[0].StringValue);
        TargetObject.Transform.rotation = Quaternion.LookRotation(direction);

        // Section0Inputs[1].FloatValue is used to get the inputfield value as float float
        steps = Section0Inputs[1].FloatValue;
        TargetObject.Transform.position += TargetObject.Transform.forward * steps;

        // --- execute next block after this, used to finish the execution of this function
        ExecuteNextInstruction();
    }

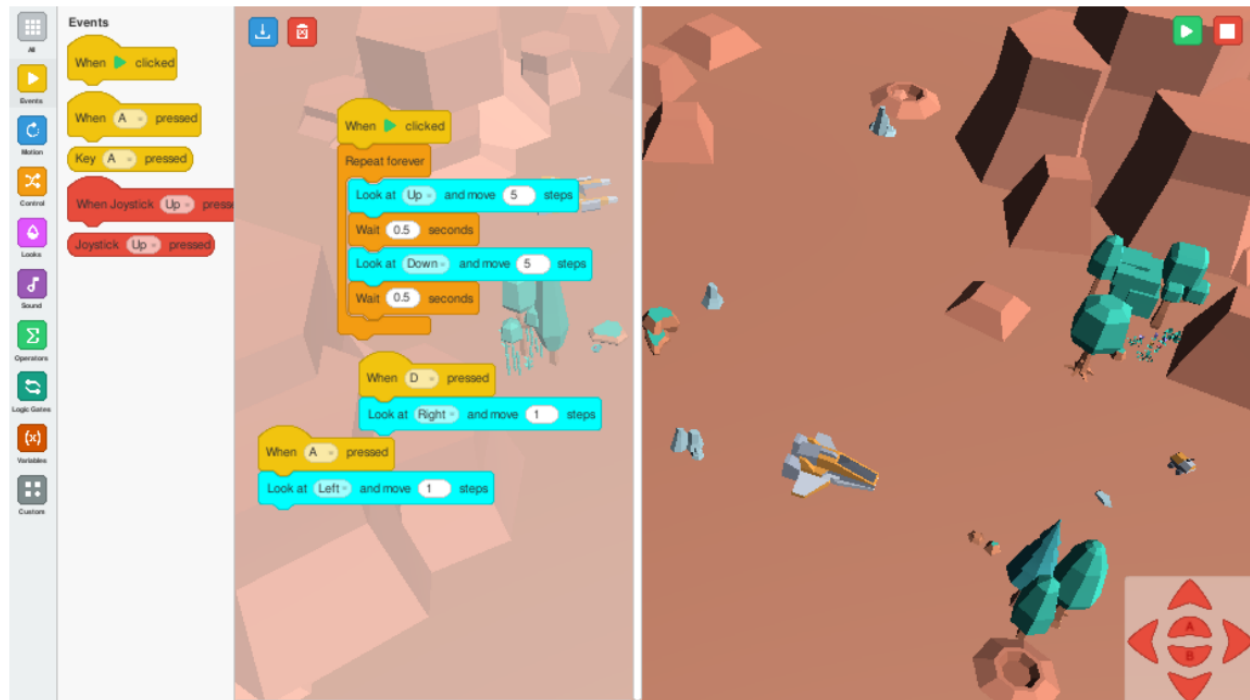
    Vector3 GetDirection(string option)
    {
        // returns the look direction based on the string value
        switch (option)
        {
            case "Up":
                return Vector3.forward;
            case "Down":
                return Vector3.back;
            case "Right":
                return Vector3.right;
            case "Left":
                return Vector3.left;
            default:
                return Vector3.zero;
        }
    }
}
```




```
}
}
```

Feel free to improve the code and ask questions about your new blocks and instructions as well as new functionalities for the Blocks Engine.

Now we can test the block in play mode!



5.1.2. UPDATING THE LOOKATANDMOVE BLOCK USING LERP

When the the block should make the Target Object move smoothly, or the block instruction takes more than one frame to end the behavior, it is needed to use the ExecuteInUpdate parameter equals true, this is done so the system keeps updating the Target Object and the environment each frame.

For us to upgrade the LookAtAndMove block to perform a smooth rotation and translation, we are going to use lerps to translate and rotate at the same time, as the following code.

```
using UnityEngine;

public class BE2_Cst_LookAtAndMove : BE2_InstructionBase, I_BE2_Instruction
{
```



```

public bool ExecuteInUpdate => true;

bool _firstPlay = true;
float _timer = 0;
int _counter = 0;
Vector3 _initialPosition;
Quaternion _initialRotation;
Vector3 _direction;

public override void OnStackActive()
{
    _firstPlay = true;
    _timer = 0;
    _counter = 0;
}

public void Function()
{
    if (_firstPlay)
    {
        _initialPosition = TargetObject.Transform.position;
        _initialRotation = TargetObject.Transform.rotation;
        _direction = GetDirection(Section0Inputs[0].StringValue);
        _firstPlay = false;
    }

    if (_counter < Mathf.Abs(Section0Inputs[1].FloatValue))
    {
        if (_timer <= 1)
        {
            _timer += Time.deltaTime / 0.2f;
            TargetObject.Transform.position = Vector3.Lerp(_initialPosition,
            _initialPosition +
            (TargetObject.Transform.forward * (Section0Inputs[1].FloatValue /
            Mathf.Abs(Section0Inputs[1].FloatValue))), _timer);

            TargetObject.Transform.rotation = Quaternion.Lerp(_initialRotation,
            Quaternion.LookRotation(_direction), _timer);
        }
        else
        {
            _timer = 0;
            _counter++;
            _firstPlay = true;
        }
    }
    else
    {
        ExecuteNextInstruction();
        _counter = 0;
        _timer = 0;
        _firstPlay = true;
    }
}

Vector3 GetDirection(string option)
{
    // returns the look direction 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;
}
}
}

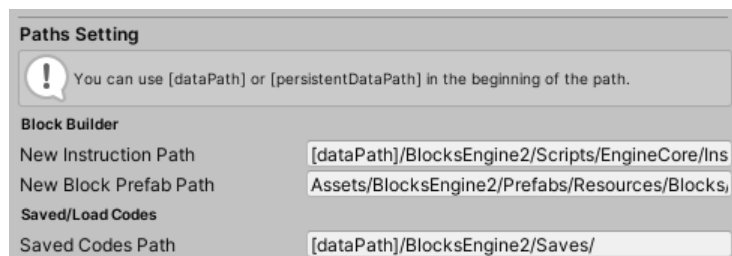
```

5.2. TEMPLATE BLOCK PARTS

This section exposes the block part templates used to create new blocks using the “Block Builder”, it enables the customization of blocks by replacing the templates for new ones.

5.3. PATHS SETTING

The BE2 system has defined locations used to store new custom blocks (instruction and prefab in editor) and the user saved codes (in play mode).



The paths can be adjusted manually from the BE2 Inspector or by script setting the corresponding static property from BE2_Paths:

Block Builder

- New Instruction Path: Where the new instruction script will be saved

```
string BE2_Paths.NewInstructionPath
```

- New Block Prefab Path: Where the new block prefab will be saved. The prefab must be saved inside a resources folder.



```
string BE2_Paths.NewBlockPrefabPath
```

Save/Load Codes

- Saved Codes Path: Where the user codes are saved and loaded from.
OBS.: Make sure to have access permission to the folder. For Mobile and Release builds, it is recommended to replace the [dataPath] with [persistentDataPath] at the beginning of the path string.

```
string BE2_Paths.SavedCodesPath
```



6. EXTRAS

This section explains how to do some customization.

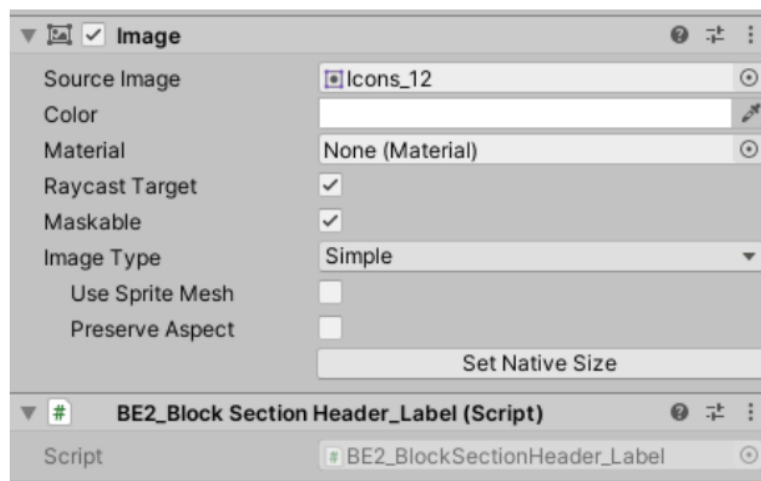
6.1. ADDING ICONS/IMAGES ON THE BLOCK HEADER

By default, the block header can be composed of Text, Dropdown and InputField components, however, it is possible to also have images, as icons that symbolize the block functionality.

(1) Add an image as a child of the Block Header. Adjust the image size and color.



(2) Add a BE2_BlockSectionHeader_Label component on the Image.

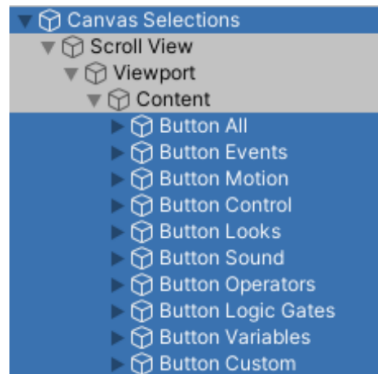


The use of images (icons) on the header is a more common scenario, however, the customization of the block headers is not limited to this, it is also possible, with the proper implementations, to have other types of inputs, such as toggles, buttons and more.



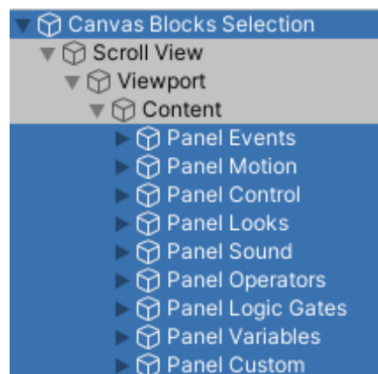
6.2. ADDING A NEW CATEGORY IN THE BLOCKS SELECTION PANEL

Go to the Hierarchy of the project and find the Canvas Selections GameObject. As a child of this GO there is a Scroll View and the content has the section buttons as children.



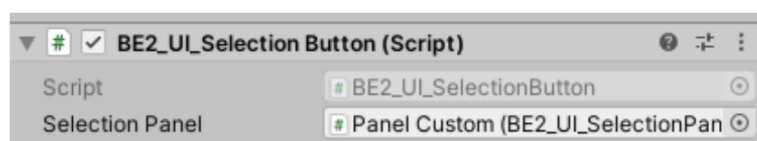
(1) **Duplicate one of the buttons** of the Canvas Selections and adjust its layout as desired. There is a BE2_UI_SelectionButton component on the button that will be adjusted later.

Now find the **Canvas Blocks Selection** and its Scroll View that has the block sections as children.



(2) **Duplicate one of the panels** of the Canvas Blocks Selection and delete its child blocks.

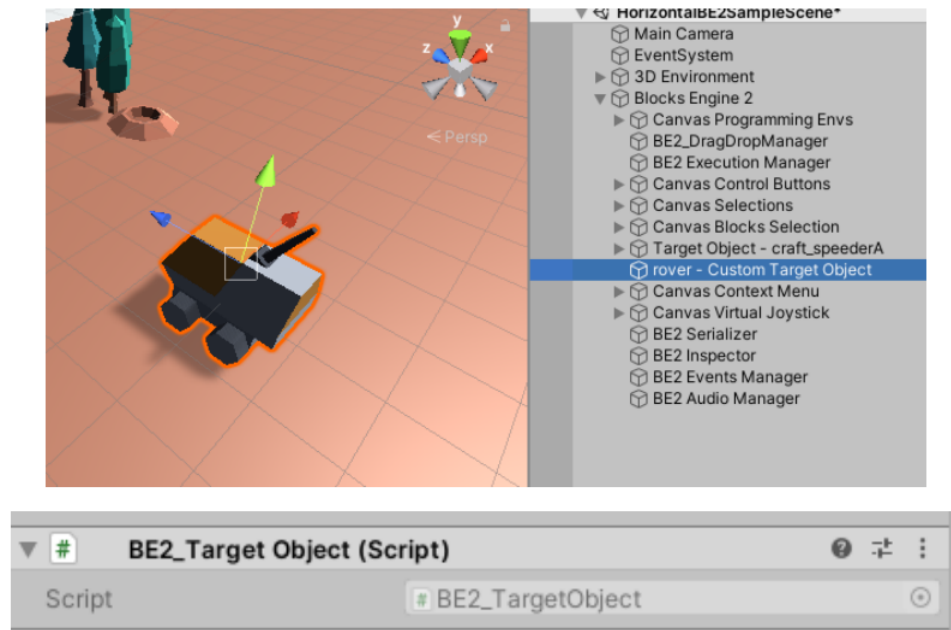
(3) **Drag the new panel** to the **BE2_UI_SelectionButton** component of the step (1) to link the panel to the button.



6.3. ADDING A CUSTOM TARGET OBJECT

You may want to use another GameObject instead of the default spacecraft (Target Object - craft_speederA), to achieve it it is needed to make sure the new Target Object is linked to a proper Programming Environment.

(1) Add a BE2_TargetObject component to the new Target Object. You can also add a component that inherits from that class, as an example of the BE2_TargetObjectSpacecraft3D.



(2) Go to the Programming Environment's inspector and drag the new Target Object to the "Target Object" field. You can duplicate the ProgrammingEnv if you need more than one simultaneous Target Object in the scene.

