Roboproc

User Guide

(version 1.0)

developed by rayleigh

(rayleigh@protonmail.com)

April 18, 2017

Contents

1 Introduction

Roboproc offers a new concept of the game mechanics: players control the character not directly but through a special set of the commands – procedure. The project is easily customized due to the editor of the levels, characters and other game elements.

Key features:

- Unique game mechanics.
- Ready to use user friendly interface.
- A few ready to use components of the interface.
- Profile system.
- Level and other game elements editor.
- 4 demo-levels and 4 characters.

If you have any questions or suggestions, email me.

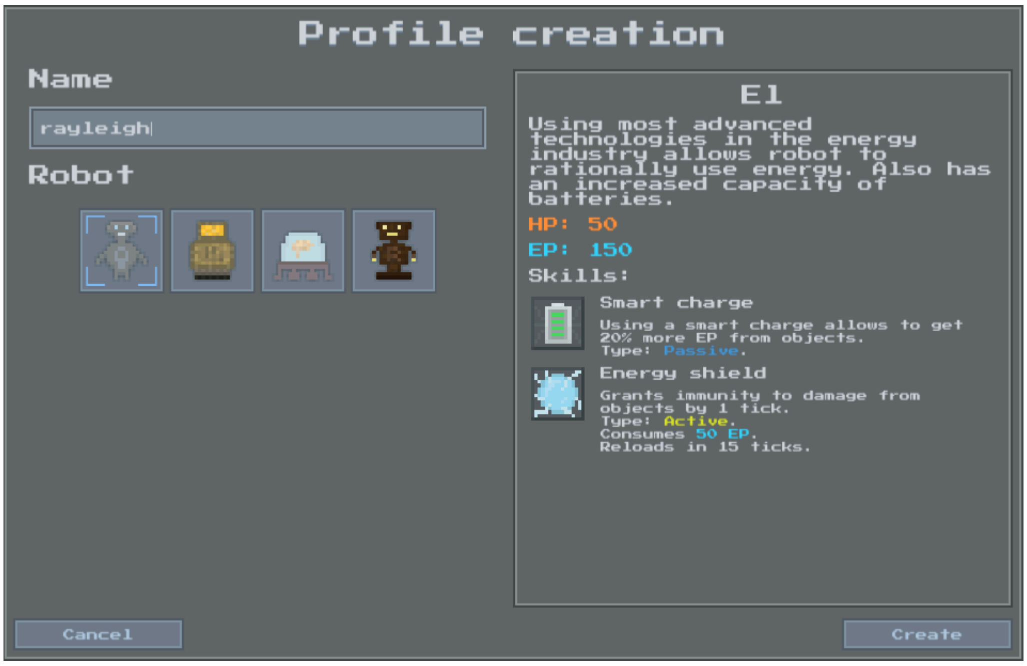
Please note that this project uses an open font Press Start 2P. The license of this font you can see here.

2 Gameplay

2.1 Beginning

First you need to create the profile and choose a character (Roboproc characters are *robots*). When you choose a robot, its description is shown in the right part of the profile creation window (fig. 1). Each robot has different health (HP) and energy (EP) points and also a own set of skills.

After the profile is created, the level selection window appears. Initially, only the first level is available. In order to open next ones, player must complete the previous.



E1

Using most advanced technologies in the energy industry allows robot to rationally use energy. Also has an increased capacity of batteries.

HP: 50

EP: 150

Skills:

Smart charge

Using a smart charge allows to get 20% more EP from objects.
Type: Passive.

Energy shield

Grants immunity to damage from objects by 1 tick.
Type: Active.
Consumes 50 EP.
Reloads in 15 ticks.

Figure 1: Profile creation window

2.2 Game

The game starts as soon as the available level button is clicked. The level and the interface on top of it are shown on the screen. Interface (fig. 2) consists of the following elements:

1. Procedure input field.
2. Camera scale buttons.
3. Button that open on-screen menu.
4. Procedure execution control buttons.
5. Game log.
6. Current robot status.
7. Active robot effects.
8. Active robot skills.

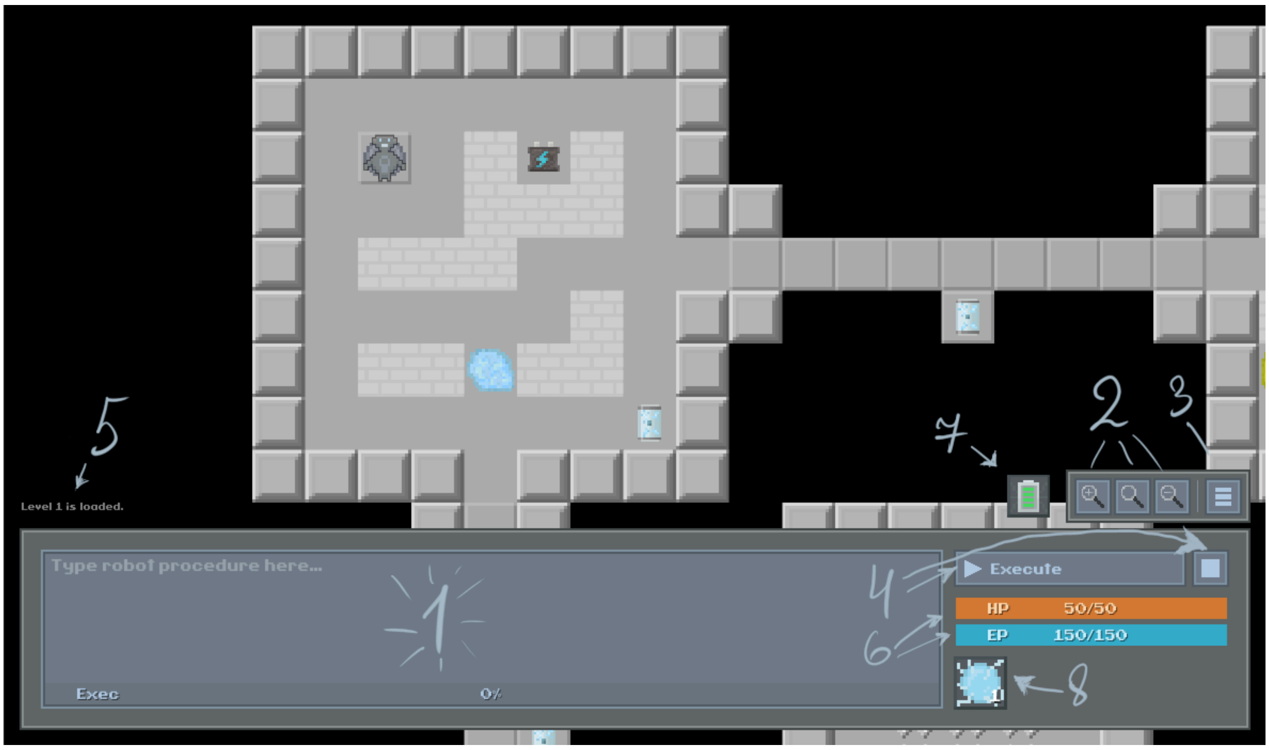


Figure 2: Game user interface

The goal of each level is to get from starting to ending point. Players must do this by typing a special procedure (program) in a special formal language. The language itself has pretty simple formal grammar, that is described in detail in subsection 2.5.

Level must be finished in one procedure. If during execution robot doesn't reach the ending point, it is sent back to the starting point and its status is reset. Procedure execution control buttons allow you to launch the execution of the procedure, pause and stop it. In case of pushing the stop button the level is reset as if robot didn't reach the ending point.

Robot moves on the *level's blocks* in four directions: up, down, left and right. It can move on the floor, but can't move through the walls and also it can't fall down into the "void" (places with no blocks).

Robots can interact with different *objects*, scattered throughout the level, automatically or using special command. These objects can be placed only on the level's blocks. There are also two special object types – starting and ending points. On the level must be one starting point and at least one ending point. When robot reach the ending point, current level completes and next one unlocks.

2.3 Robot skills

Each robot has up to 4 skills. Skills can create effects or interact with robot's HP or EP. They also can be *active* or *passive*.

Passive skills create robot's permanent effects and cannot interact with its HP or EP. Such skills aren't displayed in the game interface and can only be seen when creating profile in the robot information panel. Instead there is info of a permanent effect, created by the skill.

Active skills create temporary effects or interact with robot's HP or EP. They can have a cooldown, and also wasting some amount of robot's energy. Half-transparent cooldown progress bar appear over the used skill icon when it is reloading.

2.4 Robot effects

Effects can be *temporary* and *permanent*. Permanent effects are automatically applied by the passive skills at the start of the level. Temporary effects are applied on using some active skills. They have duration that are displayed over the icon of an active effect. When duration of the effect ends it removed. Duration of the active effect refreshes when using it again.

Effects may positively or negatively affect on robot's properties (HP or EP) or they can change objects influence on the robots. For example, effect can reduce damage from level's object, increase HP amount that restored from repair kits or regen HP/EP for each game tick.

2.5 Formal language of the robot's procedure

The robot's procedure language consists of 4 commands, 3 of them having parameters. Commands are divided by space character. Procedure's commands are executed in order they were input. Every command takes fixed time to execute. It can be changed in the on-screen menu (opens by pressing `Esc` key in game).

Command is defined by one character, followed by parameter (if command required it). Parameter of the command and command itself aren't divided by space.

List of commands:

- **Move** — moves robot by one cell in chosen direction.

  Syntax: `m<d>`, где `<d>` – move direction: `u` – up, `r` – right, `d` – down, `l` – left.

  Example: `mu` – move robot up by one cell.

- **Use** — uses level's object in current robot coordinates.

  Example: `u`.

- **Repeat** — repeats the command following this one, specified amount of times.

  Syntax: `r<n>`, где `<n>` – number of iterations.

  Example: `r5 mr` – move robot five cells right.

- **Skill** — use specified active skill.

  Syntax: `s<n>`, где `<n>` – active skill number[1].

  Example: `s1` – use skill under the number 1.

Examples of procedures, which are walkthrough of the demo-levels, you may find in appendix A.

[1]Skill number is displayed in right bottom corner of the active skill's icon

3 Getting started

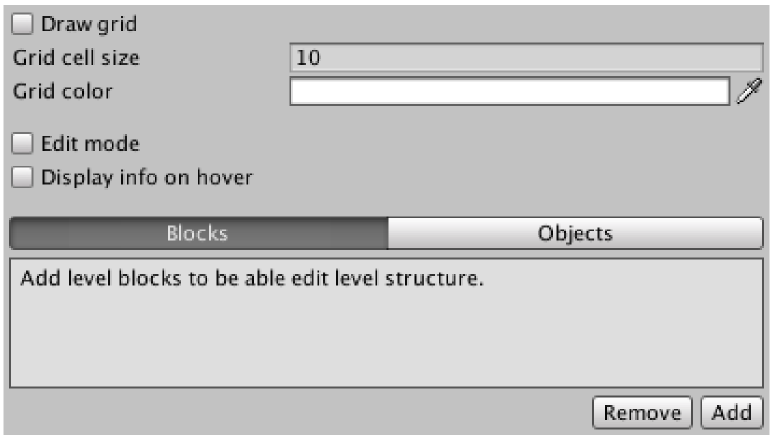
Project works out-of-box, just open `Main` scene from `Scenes` directory. It already has all necessary game objects. In `Controllers` object you can adjust basic settings. Here you can select game levels and adjust their order, edit characters and help text.

You can change the name of the project (displayed only in main menu) in `UI/Main Menu Canvas/Logo` object.

4 Level editor

To begin with let's look at creation of a *new level*. First, you have to create a new gameobject (GameObject –> Create Empty), then add `Level` component through menu (Component –> Scripts –> Level), or through inspector, using Add Component button, or drag a script `Scripts/Data/Level/Level` onto a created object.

After completing previous steps, inspector of the `Level` component is opened (fig. 3). It is used for level creation and editing.



☐ Draw grid
Grid cell size: 10
Grid color:
☐ Edit mode
☐ Display info on hover

Add level blocks to be able edit level structure.

Figure 3: Inspector of the `Level` component

Level consists of square blocks. In order to see they the borders, set checkmark Draw grid, then in Scene window a *grid* will appear (fig. 5), along which blocks are placed. The scale of the grid is adjusted by the Grid cell size field and color – with Grid color field.

Before moving on to creating level structure (subsection 4.1) you need to add resources: blocks and objects. To do that, press Add button in bottom right corner of the inspector window. If blocks tab is selected, window for adding level's block (fig. 4) will appear. If objects tab is selected, window for adding level's object will appear. By pressing Add button, chosen element will be added to level resources. After that it may be used for building the level structure.

In the adding block/object window tag search is also available. To find the necessary element, input the tag into the field that is placed in the upper part of a window. After that elements list will be filtered.

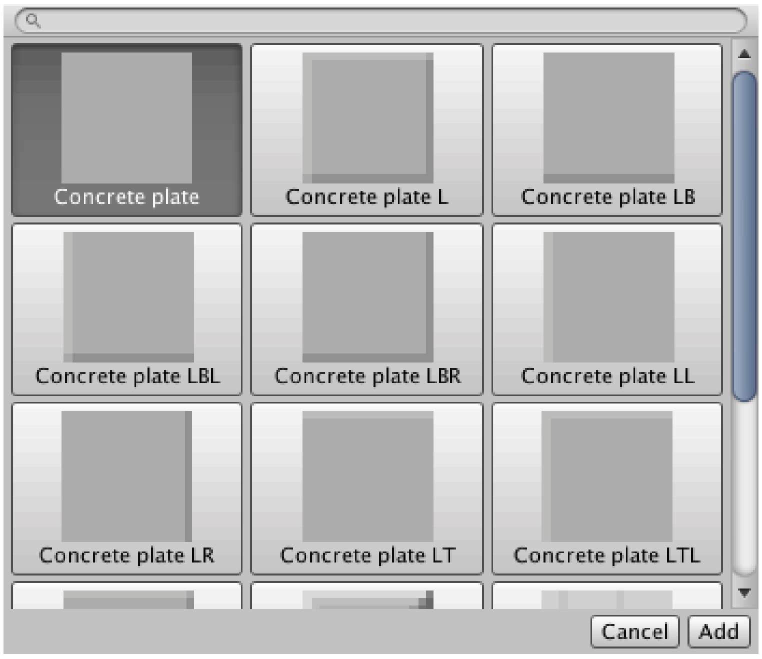


Figure 4: Window for adding level's block

4.1 Editing of the level structure

To begin editing a level structure set Edit mode checkmark in `Level` component inspector. After that the level edit mode will be toggled on and additional interface on the bottom of Scene window (fig. 5) will appear. It shows cursor coordinates and that the level edit mode is enabled. It also allows to select action, that is performed in Scene window on left mouse button (LMB) click.

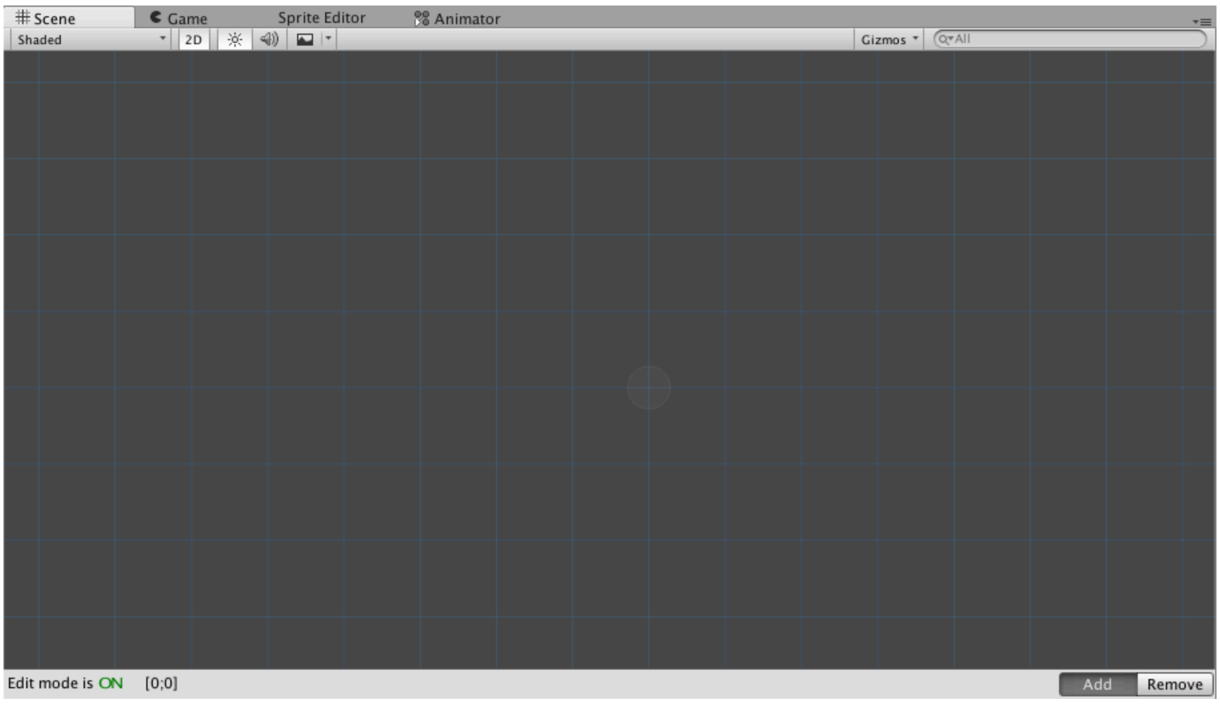


Figure 5: `Scene` window with the enabled level editor mode

Click of the left mouse button in Scene window places a block, that is selected in the inspector, in current coordinates of a cursor. If Objects tab is selected, an object will be placed instead of a block. Clicking on empty space doesn't add any objects because they can only be placed on the level blocks.

To delete a block or an object, select Remove mode in bottom right corner of the Scene window. After this LMB click will delete elements.

Adding an element on occupied place, replaces placed element with the newest one. Also, if the block contains an object, it will be deleted too.

For more convenient level editing, you may turn on the information tooltip of the hovered block or object. To do this, set Display info on hover checkmark in level inspector (fig. 3).

Each level *must* contain only one starting and at least one ending point. You can use `Start Portal` object as level starting point and `End Portal` as level ending point. This objects is used in demo-levels. If level has no ending point, player will not be able to pass the level. So don't forget to add at least one.

Example of a minimal structure of the level, created from demo levels elements is introduced on the next picture:

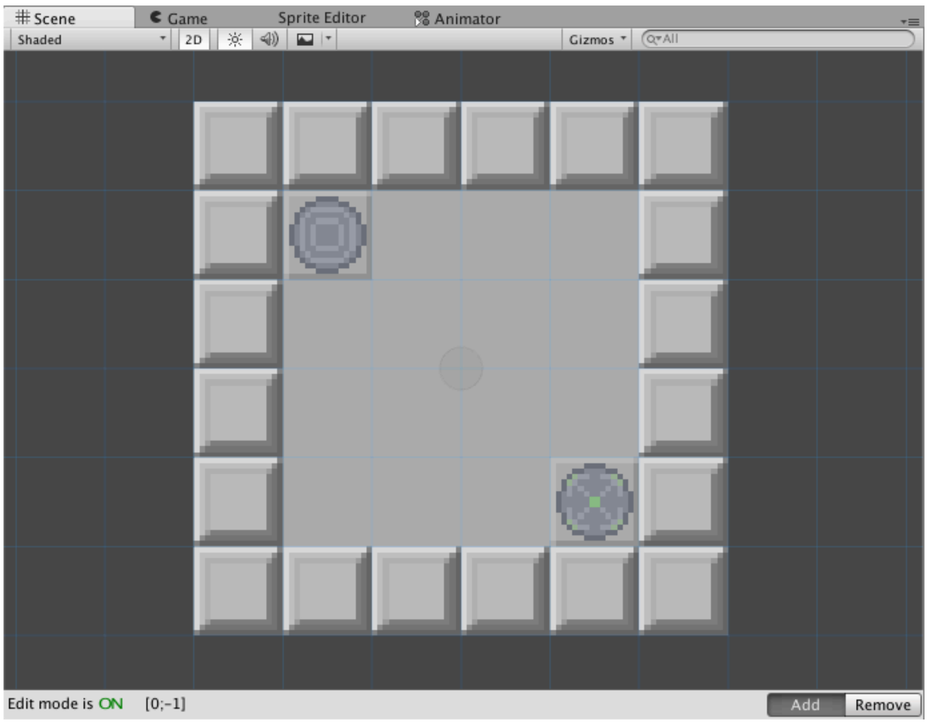


Figure 6: Example of a minimal structure

After creating the structure, level must be placed into the prefab (Assets –> Create –> Prefab). Levels are stored in `Prefabs/Levels/` directory. In order to add the created level into the game select `Controllers` gameobject and add created prefab into the levels list in the inpector of the `GameController` component. Order that set in this list will correspond to the level order in the game.

4.2 Editing of existing levels

Levels are stored in `Prefabs/Levels/` directory. Add a level to the scene by dragging the level prefab into the Scene or Hierarchy window to be able to edit it.

After editing the level it is necessary to apply changes into prefab. To do this, click Apply button in the Inspector window. If you're not going to edit the level anymore, delete it from Hierarchy window.

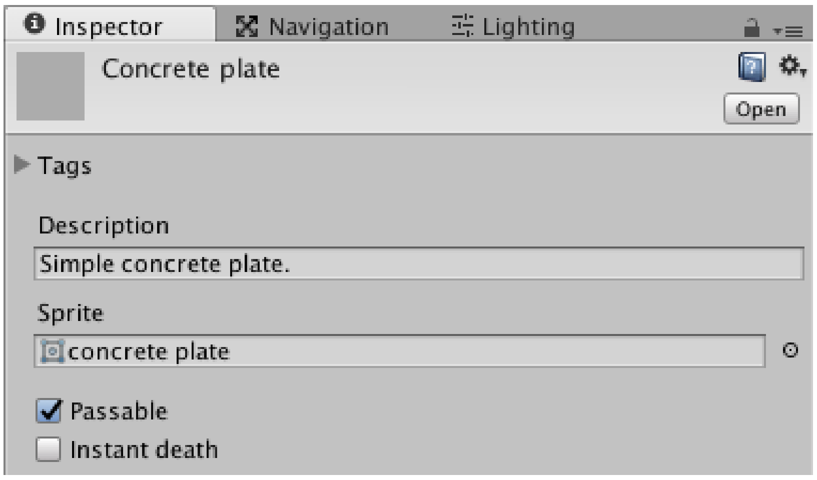
5 Adding game elements

Roboproc includes next game elements: robots, robot's skills, robot's effects, level's blocks, level's objects. All listed elements can be created by using context menu Create –> Roboproc.

Game elements are stored in `Data` directory.

5.1 Adding level blocks

Level block is added by opening context menu and pressing Create –> Roboproc –> Level Block. Block name, set in Project window, will be used in game.


Inspector | Navigation | Lighting

Concrete plate

Open

Tags

Description

Simple concrete plate.

Sprite

concrete plate

☑ Passable

☐ Instant death

Figure 7: Level's block inspector

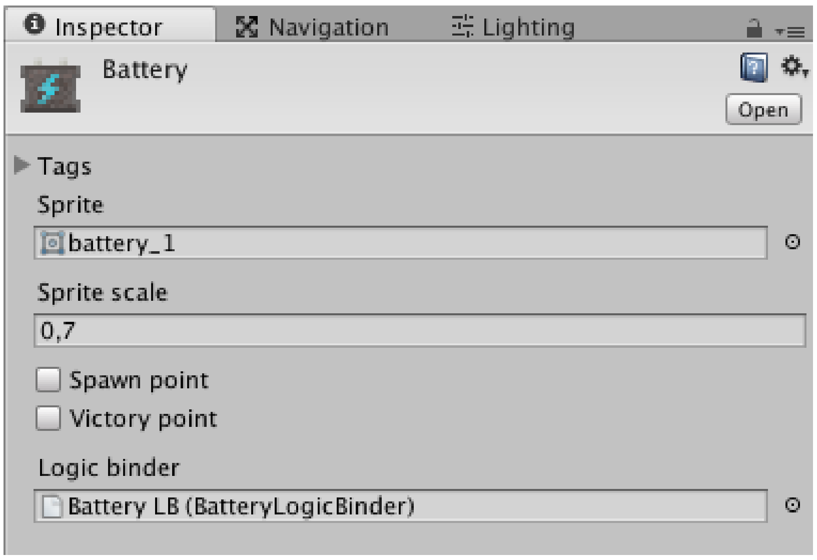
Set parameters after creating a new block (fig. 7). Setting Passable checkmark allows to make an object passable for the robot. Setting Instant death checkmark causes robot to destroy on collision with level's block.

5.2 Adding level objects

Level object is added by opening context menu and pressing Create –> Roboproc –> Level Object. Object name, set in the Project window, will be used in game.

Object parameters are shown on figure 8. Sprite scale field is responsible for sprite scale in game. Set the scale lower than 1, in order for the object not to overlap the block, it is placed on. Setting Spawn point and Victory point checkmarks make selected object as level starting or ending point. Only one checkmark may be set simultaneously.

Logic Binder field is responsible for object logic. If spawn or victory point checkmark is set, this field is unavailable. See subsection 5.3 for details.



Sprite
battery_1

Sprite scale
0,7

Spawn point
Victory point

Logic binder
Battery LB (BatteryLogicBinder)

Figure 8: Level's object inspector

5.3 Level objects logic

To understand the principle of operation of object logic, you need basic knowledge of `C#` language.

Level object logic defines behavior of an object in different situations: using object by command execution or robot entering or leaving object coordinates. Object binds with logic through `LogicBinder`, that allows using logic repeatedly. Therefore, objects with different sprites and parameters that function the same, can exist.

For example, let's look at `Battery` object, which properties are shown on figure 8. As `LogicBinder`, `Battery LB` is set for that object. It is an instance of `BattertyLogicBinder` class. In properties of `Battety LB` you can set a number of EP restored by the `Battery` object.

During the level initialization, LogicBinder creates object of the logic class and sets it parameters that assigned in the inspector. Let's look at BatteryLogic class code:

```
using UnityEngine;

namespace Roboproc.Data.Level.ObjectsLogic
{
  public class BatteryLogic : ObjectLogic
  {
    private readonly float amount;

    public BatteryLogic(float energyPoints)
    {
      this.amount = energyPoints;
    }

    public override void OnRobotUse()
    {
      var resultAmount = this.RechargeRobot(this.amount);
      this.Destroy = true;
      GameLog.Instance.Log("Robot used \"" + this.ObjectName + "\"
         and recharged by " + resultAmount + " EP.", new Color32(52,
         212, 253, 255));
    }
  }
}
```

As a parameter, class constructor accepts EP amount which will be restored on using an object with given logic. OnRobotUse method is overrided and called when command Use is executed. In the body of this method, RechargeRobot method is called. It restores given amount of robot's EP. The result of method execution is total amount of restored EP. It may *vary* from the passed amount, because active robot *effects* can influence on result value.

Then Destroy class property is set on true, what causes an object to be destroyed after using. Call of the Log method prints a message of using an object in the game log.

5.3.1 ObjectLogic class capabilities

Next methods are available for overriding in child classes:

- **OnRobotUse** – is called by executing object command Use.
- **OnRobotEnter** – is called when robot enters the area of the object.
- **OnRobotLeave** – is called when robot leaves the area of the object.

On calling methods that change robot's properties passed value, on which robot's property is changed, may differ from *returned* value because its calculation is based on active effects that influence robot's properties change.

To change current robot properties, next methods are available:

- **RepairRobot** – repairs robot. This method accepts HP amount to be restored, as an argument. Returns total HP amount, that the robot has recovered.
- **RechargeRobot** – recharges robot. This method accepts EP amount to be restored as an argument. Returns total EP amount that the robot has recharged.
- **DamageRobot** – damages robot. This method accepts HP amount to be taken from a robot, as an argument. Returns total HP amount that taken from a robot.
- **DestroyRobot**[2] – destroys robot. Does not accept any arguments.

To destroy object, set `Destroy` property to `true`.

5.3.2 LogicBinder

Let's look at `BatteryLogicBinder` class code:

```
using System;

#if UNITY_EDITOR
using UnityEditor;
#endif

namespace Roboproc.Data.Level.ObjectsLogic
{
  [Serializable]
  public class BatteryLogicBinder : ObjectLogicBinder
  {
    public float energyPoints = 20f;

    public override ObjectLogic CreateLogic()
    {
      return new BatteryLogic(this.energyPoints);
    }

    #if UNITY_EDITOR
    [MenuItem("Assets/Create/Roboproc/Logic Binders/Battery Logic
       Binder")]
    public static BatteryLogicBinder Create() { ... }
    #endif
  }
}
```

`Serialized` attribute is necessary to save the values of the fields that are set in the inspector. `BatteryLogicBinder` class has `energyPoints` field, through what a set amount of EP will be restored by an object that uses this `LogicBinder`.

Then `CreateLogic` method is overrided. This method creates and returns object of the `BatteryLogic` class. Here is defined what object logic binds to the level object.

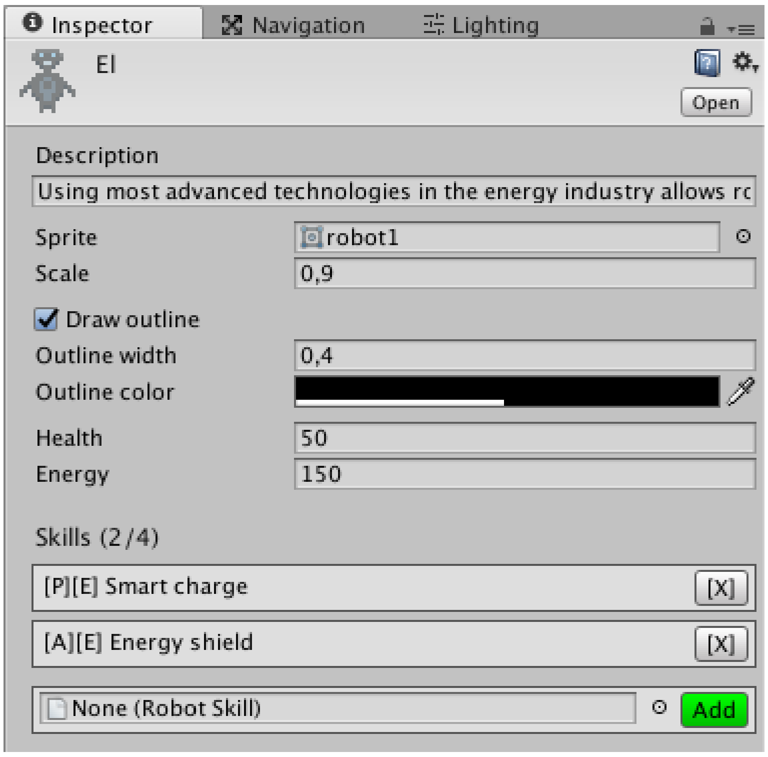
[2]Do not call after calling `Destroy` method any other methods that affect the number of robot's HP.

Static method `Create` is necessary for creating an instance of `BatteryLogicBinder` through Unity interface. This method is placed in a preprocessor directive `#if`, because it is used only in Unity editor and it requires `UnityEditor` namespace. `MenuItem` attribute creates an item in the context menu to call this method.

5.4 Robots

Robot is added by opening context menu and pressing Create –> Roboproc –> Robot. Robot name, set in the Project window will be used in game.

After creating a robot, set its properties (fig. 9). Scale field is responsible for the sprite scale of the robot in the game. Set the scale lower than 1 in order the robot to be correctly displayed on the level blocks.



Description
Using most advanced technologies in the energy industry allows rc
Sprite: robot1
Scale: 0,9
☑ Draw outline
Outline width: 0,4
Outline color
Health: 50
Energy: 150
Skills (2/4)
[P][E] Smart charge [X]
[A][E] Energy shield [X]
None (Robot Skill) Add

Figure 9: Robot inspector

Draw outline checkmark turns on robot's outline in the game. Outline width and Outline color fields allow you to set width and color of the outline respectively. Value of 1 in Outline width field equals one pixel wide outline.

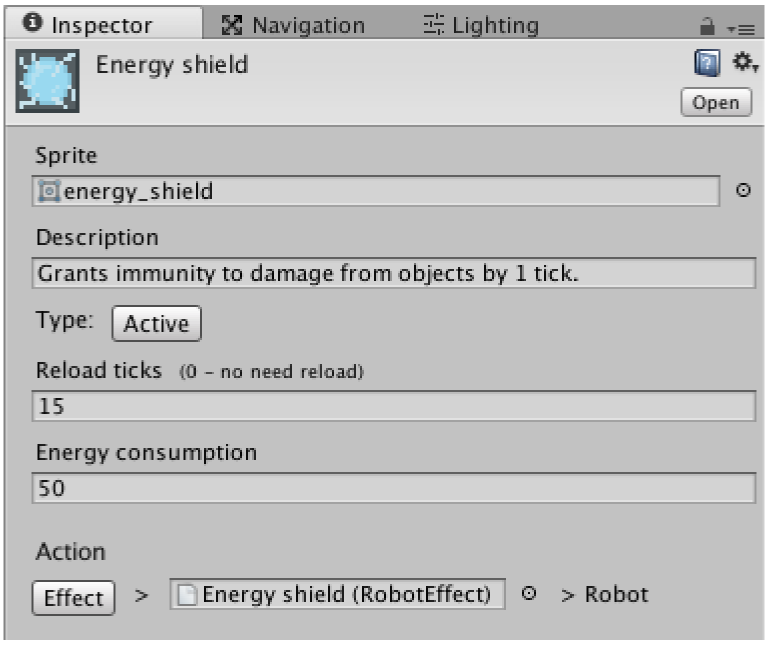
Health and Energy fields allow to set maximum amount of health (HP) and energy (EP) points.

The list of *robot skills* is on the bottom part of the inspector window. Robot can have 4 skills at max. In the skill row (in square brackets) are shown the symbols that offer short information about a skill. The first symbol defines the type of the skill as passive (P) or active (A); the second symbol defines the effect of a skill: whether it creates an effect (E) or interacts with robot properties (R). [X] button deletes the skill from the list. To add the skill, assign it to the field below the list and press Add button.

5.5 Robot skills

Robot skill is added by opening context menu and pressing Create –> Roboproc –> Robot Skill. Robot skill name, set in the Project window will be used in game.

After creating robot skill, you must set its properties (fig. 10).



Energy shield

Sprite
energy_shield

Description
Grants immunity to damage from objects by 1 tick.

Type: Active

Reload ticks (0 – no need reload)
15

Energy consumption
50

Action
Effect > Energy shield (RobotEffect) > Robot

Figure 10: Robot skill inspector

Button Active to the right of the Type label shows the type of the skill. Pressing the button switches skill type between passive and active.

Reload ticks field allows to set skill cooldown time. It is set in *game ticks*. Set its value to 0 to remove cooldown.

Amount of EP that required for usage of the skill can be set in Energy consumption field. When robot doesn't have required amount of energy points and activates the skill, the procedure will be stopped because the robot will be discharged.

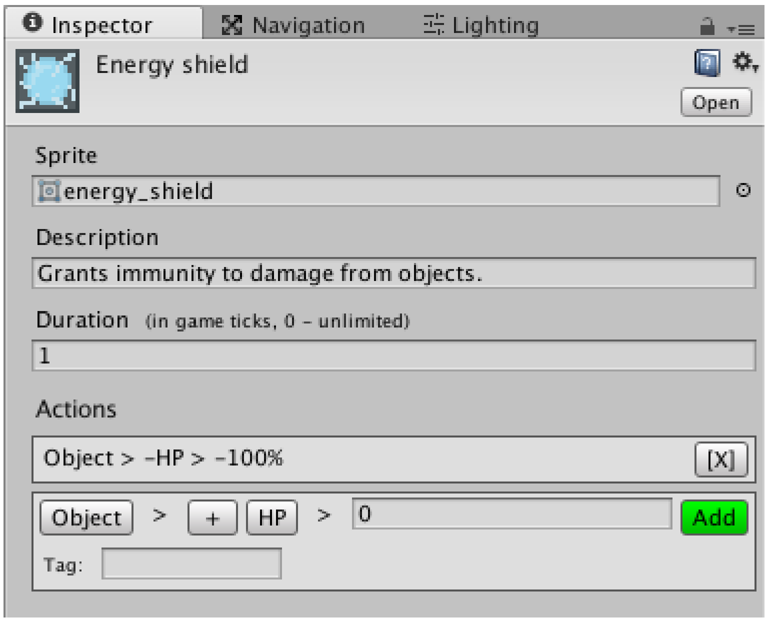
Control elements after Action label allow to set an action that performs the skill. Effect button switches skill action from creating an effect to interaction with robot properties. You need to assign effect in the field after the button, if you selected effect creation action. Upon selecting interaction with robot properties, you need to set amount of HP or EP points that will be added to the robot's corresponding property. You can also set negative values (although it is useless).

Use passive skills only for creating permanent effects as they are activated upon level initialization.

5.6 Robot effects

Robot effect is added by opening context menu and pressing Create –> Roboproc –> Robot Effect. Effect name, set in the Project window will be used in game.

After creating robot effect, set its properties (fig. 11).



Energy shield

Sprite

energy_shield

Description

Grants immunity to damage from objects.

Duration (in game ticks, 0 – unlimited)

1

Actions

Object > -HP > -100% [X]

Object > + HP > 0 Add

Tag:

Figure 11: Robot effect inspector

Duration field allows to set an effect duration in *game ticks*. Set its value to zero to make it *permanent*.

What effect can do defines by a set of Actions. Effect can have either an arbitrary amount of actions or no actions at all. Field of action addition is below the list of actions. You can delete an action from the list by pressing [X] button.

Each action has a *target*. It may be an Object or a Robot. Target of the action can be switched by pressing the first button in action addition area. Action fields set is changed upon switching the target. If target is an object, action can change the influence of the object that applies on the robot. If target is a robot, action can interact with selected robot property (HP or EP) each tick. For example, robot can gain 5 HP each game tick.

If target is an *object*, you need to witch choose robot property and how (negative or positive) it interacts with them. It is performed by pressing switch buttons that placed on the right of target switch button. At first, direction of the robot's property change is chosen: positive (+) or negative (–). Then, choose robot property (HP or EP). After this, specify percentage of object influence change in range `[-100%; 100%]`, omit the percent symbol. Also you can set object tag. If the tag isn't set, action will be applied to *every* object the robot interacts with. If the tag is set, action will be applied only on the objects that have the specified tag.

In string representation, actions which target is object have following structure:

`Object[ (<tag>)] > <sign><property> > <percent>`, where

`tag` – object tag, actions are applied to;

`sign` – direction of the robot's property change by the object;

`property` – robot property;

`percent` – percent on which you need to change the influence of an object on a robot.

Examples:

- Reduce acid damage by 75%.

  `Object (acid) > -HP > -75%`

- Increase HP regeneration from all objects by 15%.

  `Object > +HP > 15%`

- Remove damage from any object.

  `Object > -HP > -100%`

- Add EP regeneration from objects with tag "repair" by 50%.

  `Object (repair) > +EP > 50%`

If action target is a *robot*, it is necessary to choose robot's property (HP or EP) that will be changed each game tick during the duration of the effect. In string representation it has following structure:

`Robot > <property> > <amount>`, where `property` – is robot's property; `amount` – amount on which need to charge chosen robot's property.

Examples:

- Add 5 HP for each game tick: `Robot > HP > 5`
- Remove 3 EP for each game tick: `Robot > EP > -3`

6 Extension of the procedure language

Procedure language is extended by adding *commands*. Each command consists of two parts: *parser* and *class*.

Command class must be inherited from `ICommand` interface. Let's look at its code:

```
public interface ICommand
{
  bool Execute(RobotExecData robotData, BlockContainersList blocks);
  bool EatNext { get; }
  void Eat(ICommand command);
  uint CountTicks { get; }
}
```

`Execute` method – is the command main method. It is called by executing appropriate command through procedure executor. Current robot stats and the list of level blocks are passed to it. It returns bool value. In case if method return `true`, it tells procedure executor that command must be executed again. This opportunity is used in `Repeat` command.

`EatNext` property tells the executor to "eat" the next command. Eaten command won't be executed but it will be sent to `Eat` method of the current command. Example of using this mechanism you can see in code of the `Repeat` command.

`CountTicks` property must counts amount of the game ticks is needed for executing a command. Usually it is 1. This property is used to display the progress of procedure execution.

After creating command class you have to *register* it in the parser. That can be done in `CreateProcedureParcer` method of `Level` class. Command is registered by help of the delegate that accept command text as argument. Parser *must* identify, is the text of the command corresponds to the command that it must return. If yes, it returns the command, if no, returns `null` value. Procedure parser pass the command text to next command parser, until one of them returns command's object. Command text is ignored if all the parsers returned `null` value.

For example, let's look at parser code of `Use` command:

```
parser.RegisterCommand(commandText => commandText.Length == 1 &&
    commandText[0] == 'u' ? new UseCommand() : null);
```

Parser checks the text of the command is length of 1 and it character is `'u'`. In case of successful check, parser creates and returns command object. In case of failing the check, parser returns `null` and the text is given to the next parser in the chain.

7 Interface components

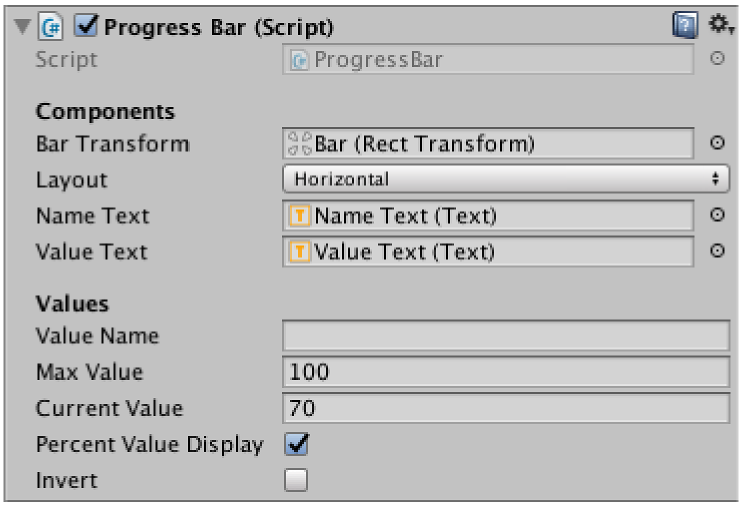
Roboproc provides components for use that greatly simplify creation of progress bars, popup dialog windows and game log.

Most of the components and templates in prefabs that are in `Prefabs/UI/` directory.

7.1 Progress bar

Horizontal and vertical prefabs of the progress bars are in `Prefabs/UI/Progress Bar/` directory. Inspector of the progress bar is shown on figure 12.

Type (horizontal or vertical) is defined by Layout property. Name of the horizontal bar, that is shown in its left part is defined by Value Name property. Max Value and Current Value properties define maximal and current value of the progress.



Progress Bar (Script)

Script	ProgressBar
Components	
Bar Transform	Bar (Rect Transform)
Layout	Horizontal
Name Text	Name Text (Text)
Value Text	Value Text (Text)
Values	
Value Name	
Max Value	100
Current Value	70
Percent Value Display	✓
Invert	☐

Figure 12: Progress bar inspector

If Percent Value Display checkmark is setted, than displayed value from "`current/max`" changed to calculated by following formula:

$$x = \frac{current}{max} \cdot 100\%$$

Invert checkmark inverts movement of progress bar stripe.

7.2 Popup dialog

Popup dialog window can be only called from the script. To set dialog window appearance use `DialogPrefabsSet` component. It is a set of prefabs, used for creating elements of the window. This allows to set a dialog style for different situations, such as: display the error, message, warning, waiting for user action choice, etc. Only one set of prefabs is used in Roboproc project. It is placed by the following path: `Data/UI/Dialog Prefabs`.

To understand the principle of creation of the dialog window, let's look at following part of code that responsible for display delete profile confirmation dialog window.

```
var deleteDialog = new PopupDialog(canvas, prefabs);

deleteDialog.SetHeader("Deleting profile");
deleteDialog.SetContent("Are you sure that you want to delete profile
    \"" + profileName + "\"?");

deleteDialog.AddButton("Cancel");
deleteDialog.AddButton("Delete", true, delegate
{
  this.profilesList.Remove(this.profile);
  this.profilesList.Save();
  Destroy(this.gameObject);
});

deleteDialog.Show();
```

At the beginning object of the `PopupDialog` class is created. Its constructor accepts next arguments: `Canvas` on which dialog is shown and `DialogPrefabsSet` that sets the style of window's elements.

Window header text is set by `SetHeader` method. Window content is set by SetContent method. Text string or GameObject both can be contents of the dialog window. `GameObject` must be an interface component.

`AddButton` method adds button to the dialog. Buttons order is determined by `AddButton` methods call order. First added button will be placed on the left, the last one on the right. Method has next arguments:

- String with text that is shown on the button.
- Boolean value that defines if the window will be closed on pressing the button.
- `Action` delegate that is called on pressing the button.

`Show` method is used for display the window. At the beginning it creates window's elements and then it shows the window itself. This can cause slight delay before showing the window. Create elements beforehand by using `Build` method in order to avoid that.

To hide dialog window, you must call `Hide` method, and to destroy elements you must call `DestroyObjects` method. Also you can set the handlers of the show/hide dialog window events:

```
dialog.Shown += delegate
{
  //Window shown
};

dialog.Hidden =+ delegate
{
  //Window hidden
};
```

7.3 Game log

Game log is implemented by `GameLog` component that must be placed on an active object[3] with `Text` component. Sending messages to the game log is performed from the scripts. For more convenient access `GameLog` component implements "*Singleton*" pattern. Example of message sending:

```
GameLog.Instance.Log("Execution started.");
```

To send messages, `Log` method is used. It is overloaded and has next signatures:

```
public void Log(string message, bool bold = false, bool italic =
    false);
public void Log(string message, Color color, bool bold = false, bool
    italic = false);
public void Log(string message, Color color, int size, bool bold =
    false, bool italic = false);
```

Arguments:

- `message` — message text.
- `color` — message color.
- `size` — message font size.
- `bold` — bold font.
- `italic` — oblique font.

[3]There must be only one active `GameLog` component in the scene.

Appendices

Appendix A Demo-levels walkthrough

Next table contains robot procedures that allow to complete all demo-levels.

	El	Hev-R	Brio-1	R-Bot 1000
Level 1	mu r4 mr r2 md u r2 mr s1 r4 md ml	ml r4 md r2 mr u r2 md r4 mr mu	mu r4 mr r2 md u r2 mr s1 r4 md ml	mu r6 mr r6 md ml
Level 2	r6 mr u r6 md ml r2 md r4 ml u r2 ml mu r3 ml r2 mu u r6 ml r3 md r3 mr	r6 mr u mr s1 r8 md u md r8 ml s1 mu r6 ml	r6 mr u r6 md ml r2 md s1 r4 ml u r8 ml	r6 mr u mr r8 md u md r8 ml mu s1 r6 ml
Level 3	r6 md r12 mr r2 md u r2 ml r2 md r2 ml u r5 ml u r2 ml r2 md u r2 ml s1 r2 md u r2 mr r2 md u r9 mr r3 md u r5 mr r3 md u s1 r2 md u r2 md r5 ml u r5 md	r6 md ml s1 r2 md r3 mr u r4 mr u r4 mr s1 r2 md r2 ml u r3 ml s1 r2 ml u r2 ml r2 md u r2 ml r2 md u r2 mr r2 md u r5 md s1 md mr r4 md u s1 r2 ml u r2 mr r3 mu r3 mr u r3 md u s1 r5 mr u r5 md	r6 md r6 mr r2 md u r4 mr r2 md r2 ml u s1 r5 ml u r2 ml r2 md u r4 mr r2 md u r4 ml r2 md u r5 mr u r4 mr r3 md u r4 md u r3 md u s2 r5 md	r6 md r5 mr r2 md u mr u r4 mr r2 md r2 ml u r5 ml u r2 ml r2 md u r2 ml r2 md u r2 mr r2 md u s1 r5 mr u r4 mr r3 md u r4 md u r4 md s1 r4 md
Level 4	mu r3 mr md u mu r2 mr r3 md r6 mr md u mu r7 mr r3 md ml u mr mu s1 r2 mr r3 md u r7 md u r2 md r3 ml u mr r2 md r5 mr r2 mu ml u mr u r2 md ml r2 md r3 ml u r2 md r2 ml r3 mu u r4 ml md u mu r4 ml md u mu r2 mr r7 mu u r2 mu	ml r3 md r3 mr r2 md r3 mr u r3 ml r4 md ml u mr r2 md mr u ml r5 md r3 ml r2 md s1 r5 mr u ml r2 md r4 ml s1 r2 md r5 mr s1 mr mu r2 mu u r4 mr md u mu r2 mr s1 r7 mu u r2 mu	ml r3 md r3 mr r2 md s1 r3 mr u r3 ml r4 md s1 ml u mr r2 md mr u ml r5 md r3 ml r2 md r5 mr u ml s1 r2 md r4 ml r2 md r4 mr s1 r2 mr r3 mu u r4 mr md u s1 mu r2 mr r7 mu u r2 mu	ml r3 md r3 mr r2 md s1 r3 mr u r3 ml r4 md ml u mr r2 md mr u ml r5 md r3 ml r2 md r5 mr u ml s1 r2 md r4 ml r2 md r6 mr r3 mu u r4 mr md u mu r2 mr r7 mu u r2 mu