AMBIENT ASSISTED LIVING JOINT PROGRAME

AAL-2013-6-039

SeniorLudens

Serious Games development platform for older workforce training and intergenerational knowledge transference

D2.5
Serious Games development engine (M19)

Workpackage	WP2 – Serious games development engine design and implementation
Lead beneficiary	INDRA
Editor(s)	Dani Tost- CREB-UPC Ariel von Barnekow – CREB-UPC Núria Bonet Codina – CREB-UPC Salvador Aguilar – INDRA Gary Honegger - YR
Contributor(s)	Stefano Puricelli - CBIM
Reviewer(s)	Marije Blok – KBO Salvador Aguilar - INDRA
Release Date	10/2015
Version	V1.0
Circulation	Project Partners, AAL Control Management Unit, and National Funding Agencies.

Table of Contents

Abstract

This document aims for detailing the installation of SeniorLudens Game Engine. It can be also considered as a user manual attached to the software pilot developed.

The Game Engine is the system in charge of automating the Serious Games creation process. Based on the different element which comprises the SL Game Engine, this deliverable is composed of four parts: in the first part it describes the user manual of the Game Kit (SLGK), in the second part the Scenario Editor, the third part the Task Editor and the last one is reserved for the Training Program Editor.

All the information described in the document is available in an html wiki to support the organizations and game designers during the game creation process.

1- SeniorLudens Game Kit Documentation

SeniorLudens is a Serious Game Development Platform for older workforce training and inter-generational knowledge transfer, funded by the EU program AAL-2013-6-039. The platform is formed by multiple components and one of them is the GameKit, this documentation is about it.

To know more about the platform and the role of the Gamekit in the platform, please read SL_Gamekit.

This part introduces the SeniorLudens GameKit and then focuses on step- by-step instructions for the creation of games.

1.1- Installation

SeniorLudens GameKit currently only supports one game engine: Unity 3D. This section ex plains how to install the game kit and its dependencies in a Unity project.

Note: Please take note that, in the current stage of SeniorLudens development, the SeniorLu-dens GameKit and warehouses are installed in intermediary servers. This will be modified in the future since everything will be installed in the final SL server. Therefore, these instructions will need to be updated when the SeniorLudens GameKit will be uploaded to the SeniorLudens platform.

1.1.1- Install the Gamekit for Unity

1.1.1.1- Requirements

To use the game kit, you must install the following programs:

- Unity 3d 5.2.2f1
- Git
- Python >= 3.3
- Blender

Why Unity 5.2.2f1?

The reference version of Unity 3D for developing games with the SeniorLudens Gamekit is 2.2f1, however any minor version from the 5.2 branch should work and we recommend to use the latest version from this branch before opening to the public a new game.

- **Download links** *Windows Mac*
- **Release notes** *5.2.2*
- **API** *http://docs.unity3d.com/520/Documentation/ScriptReference/index.html*
- **API History** http://docs.unity3d.com/ScriptReference/40_history.html#5-2-1

The development of the game kit started with version 4.5.0 (27 May 2014), since the announcement of the new UI on version 4.6 we were expecting it. We tried it just after the release, on April, and we found some issues. We decided to wait few minor versions, finally we migrated from 4.5.5 to 4.6.3, few weeks later unity announced the public release of a new major version 5, at this point we decided to not upgrade to 5 and stay with 4.6 until autumn, we did a

minor upgrade to 4.6.4 and due to the fast release cycle of the following minor versions we decided to stick with this version until we move to Unity 5.

On late September 2015 we moved to Unity 5.1 and on October to the last release, 5.2.2f1, and we will try to freeze this version until we release our second prototype.

1.1.1.2- Obtain the Game kit

Warning: Once the game kit becomes more mature, we will provide it as an asset importable to Unity. By the moment the unique way is to obtain it directly from its repository.

You can obtain the game kit for Unity cloning the repository SeniorLudens UnityGameKit:git@movibio.lsi.upc.edu:seniorludens/unity.git inside your Assets folder, with the following commands:

```
# Clone the gamekit
git clone git@movibio.lsi.upc.edu:seniorludens/unity.git SeniorLudensGameKit
# Init gamekit submodules
cd SeniorLudensGameKit
git submodule update –init
```

Note: If your project uses git as source management control, you will prefer to add the gamekit as a submodule 3 instead of cloning the repository:

```
git submodule add git@movibio.lsi.upc.edu:seniorludens/unity.git Assets/SeniorLuden
# Init gamekit submodules
cd SeniorLudens
git submodule update --init
```

1.1.1.3- Obtain the Game kit SDK

The SeniorLudens Gamekit SDK extends the Unity Editor to improve the development experience with the SeniorLudens GameKit, its features are explained in the section gamekitunity3d.

Note: The following code expects to have Python installed, we recommend using Python 3.4 (amd64).

To install it you have to clone the SeniorLudens UnityGamekitSDK: git@movibio.lsi.upc.edu:seniorludens/unitysdk.git to your Assets/Editor folder, if you are inside your assets folder you can do it with:

```
# Clone the gamekit
git clone git@movibio.lsi.upc.edu:seniorludens/unitysdk.git Editor/SeniorLudensGame
pip install -r Editor/SeniorLudensGameKit/requirements.txt
```

Or if you prefer to use a submodule:

```
# Clone the gamekit
git submodule add git@movibio.lsi.upc.edu:seniorludens/unitysdk.git Assets/Editor/S pip
install -r Assets/Editor/SeniorLudensGameKit/requirements.txt
```

Finally you have to install manually lxml 4, if you are using MS Windows, we recommend you to use an unofficial build of lxml:

```
pip install "http://www.lfd.uci.edu/~gohlke/pythonlibs/3i673h27/lxml-3.4.4-cp34-non
```

1.2- Create a new world

1.2.1- Before you start

Every time a new virtual environment is needed, a new SeniorLudens world (see SL_worlds) must be built. These instructions explain how to do it. If you already have a world and you want to add it some objects or actions, go to “Modify a world” section. If you have a Unity3D scenario that you want to use in SeniorLudens, follow to “Migrate a Unity Scenario” section. In any case, before you start, read Section “Who is able to modify a world and why?” to make sure that you are aware of the roles and permissions in the SeniorLudens platform.

Creating a world means not only creating a 3D scenario, but also organizing it in the very specific way that SeniorLudens GameKit needs in order provide the required functionalities. In particular, you will need to create a new Unity3D project, do some settings, create a warehouse (see warehouses), scenes 1 and add the basic structural objects. After that, the world core will be created and you will be able to improve it adding objects and behaviours following the instructions of Modify a world.

Please, follow the instructions sequentially; otherwise you may end in a deadlock.

1.2.2- Instructions

1.2.2.1- Create a new SeniorLudens Unity Project at CREB’s VCS

In this document, you will learn how create a new repository at CREB’s VCS, to pull the needed files from the git repository and how to create and configure an empty Unity 3D project for your game.

1. Open http://movibio.lsi.upc.edu/gitlab/groups/seniorludens and authenticate yourself.
2. In the right menu, select New Project (green button), if you don’t see the button probably you don’t have permission to create new projects, please contact with someone from CREB.
3. Follow the step-to-step instructions: specify the path of the project and a short description. The namespace is seniorludens (default) and the scope is private. Confirm.
4. Open a new git window or console and follow the instructions step-by-step: general setting the first time and create a repository.
5. Create a directory called src the project directory, this will be the folder the Unity Project.
6. Open Unity –> File –> New Project and indicate the src directory.
7. Automatically, various directory will be created inside src, one of them called Assets will contain all the re- sources of the project.

8. Install the SeniorLudens Gamekit for Unity3D as it's explained at Installation.
9. In order to avoid uploading unneeded files, add the following contents to the .gitignore:

```
############################
# SeniorLudens .gitignore
# v1.0
############################

### Unity ### [Ll]ibrary/ [Tt]emp/ [Oo]bj/

# Autogenerated VS/MD solution and project files
*.csproj*
*.unityproj*
*.sln*
*.suo*
*.user*
*.userprefs*
*.pidb*
*.booproj*

#Unity3D Generated File On Crash Reports sysinfo.txt

# Other generated files bin/
obj/


### Blender####
*.blend?
*.blend?.meta

### Distribution files ####
dist/
Docs/

### Testing ###
coverage/
TestResult.xml
```

If you prefer you can download it from: SeniorLudens .gitignore

10. Modify the editor project settings Unity: Edit -> Project Settings -> Editor to use text serialization and visible metafiles, as show in the next image:

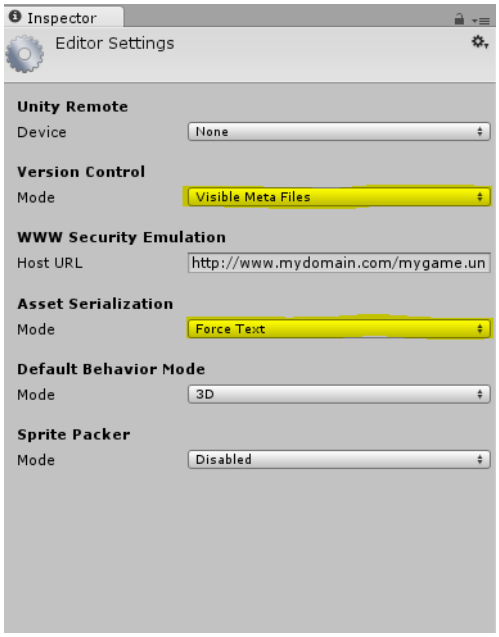

Figure 1 Enable Visible Metafiles

11. Do not close your Unity3D project, just proceed to the next step to create the basic SeniorLudens Project Files . Remember that from now on, you should periodically, add your project files, make a commit and push it to your repository:
    - git add (whatever needed, check with git status)
    - git commit
    - git push

1.2.2.2- Configure Unity

Unity 3D lets you name and configure the events of the mouse, keyboard or joystick as Input Axes, to use SeniorLudens Gamekit for Unity3D you have to rename and configure two of the default axes.

To configure the axes you have to go to Edit –> Project Settings –> Input and then Inspector –> Axes, and then:

1. Rename axis Fire1 to PrimaryAction.
2. Exchange the values of parameter Positive Button and Alt Positive Button (optional).
3. Rename axis Fire2 to SecondaryAction.
4. Exchange the values of parameter Positive Button and Alt Positive Button (optional).

Your configuration should look like:

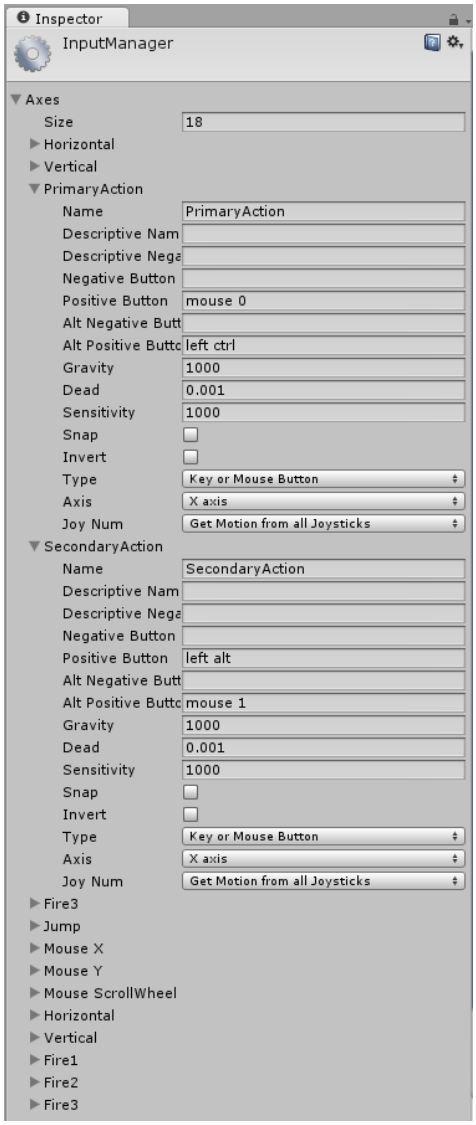

Figure 2 Set default actions

1.2.2.3- SeniorLudens project files

A SeniorLudens project needs the following files:

- **myprojectConfiguration.cs** that contains the definition of the actions that are specific actions of your world.
- **myprojectWorld.xml** that defines all the objects of your world
- **scenarioConfiguration.xml** that describes the initial content of all the scenes of your world. [1]
- For each task:

  mytask.xml that describes the contents of a task. [1]

1. Create the C# file *MyProjectConfiguration.cs*. By now, just copy the file

enclosed below. Do not forget to substitute *MyProject* by the name of your project.

```
using SeniorLudens.Unity;

public class MyProjectConfiguration :
SeniorLudensUnityAppConfiguration {

        public override void Configure
        (SeniorLudens.Core.Application app)
        {
                // Configure your application here adding
                your custom behaviou

        }

}
```

2. Create the xml file *MyProjectWorld.xml* as shown below. As you can see it is almost empty. It only includes a default object called *room*. With this you have enough right now, so just copy it as it is. Later, you will add the definition of all the objects of your world after this definition.

```
<?xml version="1.0" encoding="UTF-8" ?>
<world>

<objectdef id="room">
   <states default="decoration">
   <state name="decoration"></state>
   </states>
</objectdef>
```

3. Create the xml file *scenarioConfiguration1.xml*. By now, this file is also almost empty. Within the tags <positions> </positions> you'll put the initial positions of the instances of the objects existing at the beginning of the game in each scene of the world.

```
<?xml version="1.0" encoding="UTF-8" ?>
<scenarioconfig>


</scenarioconfig>
```

4. Create a first task FirstDemoTask.xml that simply closes the game after a lapse of time. You can copy the one below. Try to understand it. It has only the introduction step and one track composed of a sequence of three blocks. The first block is a conditional block that requires waiting 60 seconds. The second block is a system action consisting of showing the message 'Bye!'. Finally, the third block is again a condition of a waiting time of 20 seconds.

```
<?xml version="1.0" encoding="UTF-8" ?>
<task>
    <meta>
        <name>Test task</name>
    </meta>
    <introduction>
       <track>
            <sequence>

                <condition>
                   <expression>
                        <wait>60</wait>
                   </expression>
                   </condition>

                   <action>
                        <subject>system</subject><verb>setTextPro
                   perty</verb><directobject>message</directobject
                   >
                        <param name="text">Bye!</param>
                   </action>

                   <condition>
                   <expression>
                    <wait>2</wait>
                </expressi
                on>
                 </condition>

            </sequence>
        </track>
    </introduction>
</task>
```

5. In the next step you will create a warehouse for your world.

1.2.2.4- Create a new warehouse

In order to create a new warehouse, you just need to create the directory *warehouseMyWorld* in *Assets*. This warehouse will contain the objects that are specific to your world, have been designed for it and are not shared by any other world. In your world, you will use the objects of this warehouse and those of the SeniorLudens warehouse that are common to all SeniorLudens projects. See warehouses to know more about the structure of the warehouses in SeniorLudens.

1.2.2.5- Define the world

You've just done the *Basic setting for the creation of a new world* of your world. Unity 3D provides a default scene. Use it to create a new component that represents your world.

6. In the *Assets* directory create a new directory called *Scenes.* This is where you will store your scenes.
   - Save the default scene: *File -> Save Scene* (choose a convenient name and indicate the newly created direc- tory *Scenes*)
7. Create an empty object. Call it after your world's name (herein

   MyWorld). *GameObject -> Create Empty* (name it as *MyWorld*)
8. With the empty object selected, add the component *MyWorld Configuration* to it. It is the script that you've created in Step *SeniorLudens project files* :

   In the *Inspector* panel -> *Add Component -> MyWorld*
9. Configure *MyWorld Configuration* in the corresponding fields of the components panel. The Game level is the name of the scene. The world file is the xml file of definition of your world (*NameWorld.xml*), the scenariofile *ScenarioConfiguration1.xml* and the task *FirstDemoTask.xml*. See the image below for the world called Yalm.

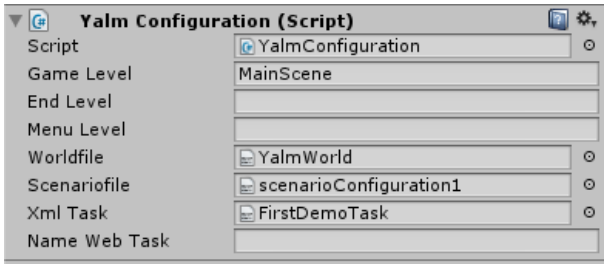

Figure 3 Configure MyWorld Configuration

1. Create a *prefab* with the empty object by dragging it from the *hierarchy panel* to the assets directory in the Project panel at the lower part of the screen. A prefab is a Unity3D template of the objects and its components.
2. You are now ready to work on this scene add to it structural elements and objects. We are now ready to create a scene.

1.2.2.6- Create a scene

Connect the scene with *SeniorLudens GameKit*

1. If you've just defined a new world (*Define the world* section) open it. You have already created a new scene. Otherwise, create a new scene in your old world and save.
2. In *Unity 3D* create an empty object that will represent the scene. Name the empty after the scene's name:

   GameObject → Create Empty → (name it as MySceneApp)
3. Assign the script *SeniorLudens Scene* to recently created object *MySceneApp*:

Window → Inspector → Add Component → SeniorLudens →SeniorLudens Scene

4. Fill the corresponding fields: the configuration script is MyWorlConfiguration.cs, the **Game Server URI** is http://movibio.lsi.upc.edu/seniorludens/dev and the **Development Server** http://localhost:5000. See the image below where the name of the project is *Yalm*.

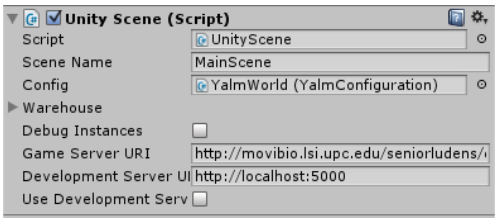

Figure 4 Complete Scene configuration

5. Ensure that the *game level* parameter of your world prefab is the scene's name.

1.2.2.6.1- Create a Skybox

A skybox is a panoramic texture that represents the background of the scene: a sky or something similar. To create it, just follow the instructions of Unity3D Skyboxes[1]. Summarizing:

1. Open the render settings: *Edit → Render Settings*
2. In the *Inspector* window, on the default layout at the right side, press the *Skybox Material* button and search the material (e.g.: *sunny1 Skybox*).

Note: You can design skyboxes of your own or use those provided in the corresponding Unity3D package *Assets → Import Package → Skyboxes*)

1.2.2.6.2- Add lights

To start, add just a basic Unity3D light[2], the available lights can be found on the menu under *GameObject → Light* or with the keyboard Alt-g Alt-l

For example to add a directional light you will select on the menu *GameObject → Light → Directional Light* ' or with the keyboard *Alt-g Alt-l Alt-d*.

1.2.2.6.3- Create your basic structure (ground and walls)

You can create them directly in Unity3D, or alternatively create them with a digital content creation editor (e.g. Blender) and insert them in your project. In both cases, you must be

[1] http://docs.unity3d.com/Manual/HOWTO-UseSkybox.html

[2] http://docs.unity3d.com/Manual/class-Light.html

aware of the structure of the files and directory in SeniorLudens projects. Please read first the warehouses Section. You will also need to create the definition files of the objects.

The ground and walls are not likely to be shared by any other project and thus, in general, they will not be stored in the SeniorLudens warehouse. Thus, you will create them and store them in your project's warehouse in the corresponding directories as follows:

- In your project's warehouse, create a directory called *objects*, and inside it create a directory called *structure*. If your ground is an open-air landscape, create a directory called *landscapes*. In these directories, create a directory for each of your objects (e.g. in the *structure* directory create *walls*, *roof* and *ground*).
- Create the structure objects with *Unity 3D* ...

  In the menu *GameObject -> 3D Object –>* select a plane, a terrain or whatever needed. In the *Inspector* panel, modify dimensions, position and other attributes.

- ... or import them from Blender

  – Read *first* the guidelines on how to create a Blender object to include it in a Senior-Ludens project in Section blender_objects. This will spare you a lot of time!

  – Save the *.blend* in the newly created directory.

  – In *Unity 3D*, drag the object from the *Project* to *Scene* and modify its property in *Inspector* panel.

After that, in both cases perform the following steps following the steps described in Section Unity_objects:

- Add the *VisualObject* component
- Add the *Collider* component
- Mark *Is Trigger*
- In the *Inspector* panel mark the *static* flag
- Create an *.xml* file with the definition of the object.
- Create the *prefab*

1.2.2.6.4- Add the definitions of the structural objects in the file ProjectWorld.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<world>

<objectdef id="ground">
        <states default="decoration">
                <state name="decoration">
                                <action     name="navigate"
                                function="place"/>
                </state>
        </states>
</objectdef>

<objectdef id="wall">
        <states default="decoration">
                <state name="decoration">
                                <action     name="navigate"
                                function="place"/>
                </state>
        </states>
</objectdef>

</world>
```

In the file YourProjectNameWorld.xml that you have created in the previous steps, add the definitions of the structural objects of the scene. For example:

1.2.2.6.5- Register the scene

Just skip this step by now until *SeniorLudens* supports multiscenes.

1.2.2.6.6- Add objects

To add objects in your scene, follow Section *Add an object to a world* section.

1.2.2.7- Create the user's avatar

SeniorLudens games are *first-person perspective*. This means that the environment is rendered from the point of view of the player's avatar. This avatar is a special SeniorLudens object composed of a camera and a very simple bounding box body used to control collisions. The avatar's camera is used at each frame to render the scene.

6. Add the user's avatar to the scene:

   Look for the object *avatar* in the SeniorLudens warehouse. Pick it in the *Project* panel and drag the *avatar* prefab to the *Scene*. In the *Inspector* panel, adjust position and rotation.

7. Remove the object *Main Camera*. You don't need it anymore, because you will use the

avatar's camera.

8. Check that you can observe the scene from the avatar's perspective:

   Press *Play* and move the mouse. You should be able to rotate the view direction.

   You'll probably get various error messages in the *Console* panel. Don't care about them, you'll fix them in the next steps.

9. Add the definitions of the avatar to the project definition file **myprojectWorld.xml** (see *SeniorLudens project files*). Since the avatar is a compound of objects, you must add the definition of each of the parts that you want to be able to refer to in the game logics. In general, it is enough to define the *GUIcamera* that composes the avatar's main camera and the avatar itself. Thus, add the following peace of definitions to the project definition file.

```
<objectdef id="GUICamera">
        <states default="decoration">
                <state name="decoration" model="GUICamera"/>
        </states>
</objectdef>

<objectdef id="user">
        <states default="motionless">
                  <action name="quit" function="subject"/>
                </state>
                <state name="motion">
                  <action name="stopNavigation"
                  function="subject"/>
                </state>
                <state name="staticCamera">
                  <action name="lockCamera" function="subject"/>
                </state>
        </states>
</objectdef>
```

10. Setup the navigation:

    (a) Check that all the objects that won't move during the game (e.g the ground and the walls) are *static*.

        With the object selected, check *static* in the *Inspector* panel (at top right).

    (b) For all *static* objects, define if the *avatar* will be allowed to navigate automatically towards it or not.

        With the static object selected, in the *Window* submenu of the main menu, select the *Navigation* option. In the *Navigation* panel, select the *Navigation layer*. Choose *Default* if you want to allow navigation towards the object and *Not walkable* otherwise. For instance, with the object *ground* selected, choose: *Navigation Layer -> Default*.

    (c) At the bottom right of the *Navigation* tab, press the button *Bake*.

You'll be able to see in blue the navigation mesh[3].

(d) Adjust the mesh by tuning the parameters:

- *Radius*: the proximity margin to the obstacles. Be careful with this value: if it is too large, you won't be allowed to pass through narrow doors. A good value according to SeniorLudens scenarios scale is 0.2
- *Height*: a good value is 1.8. If it is too high, the avatar won't be able to pass through doors.

11. Verify that the avatar navigates and recognizes the different scene's objects.

Press *Play* and click on the object you want to go to. On the bottom of the screen a message will indicate where you are navigating to.

Warning: If it doesn't work, check that the object you want to go to has a collider. In general a *Box Collider* is enough. However, if the object is a closed space (a room, for instance) and the avatar is inside, you need to put a box collider per wall or a *Mesh collider*. Check also that the flag *IsTrigger* is on.

1.2.2.7.1- Configure the avatar

Select the avatar and in the *Inspector* panel, in the *Nav Mesh Agent* component, you can modify interesting features:

1. **Speed:** navigation speed [p.ex: 1]
2. **Stopping Distance:** Distance to the clicked object at which the avatar stops

---

Note: Apparently, the minimum *Stopping Distance* is 0.8. Below, the behaviour is as if it was 0.

Warning: Choose carefully the distances, because if they are too large, the objects may all fall out of the scope of the avatar.

1.2.2.7.2- Configure the avatar's camera parameters

Select *Main Camera* in the *avatar* compound,. In the panel *Inspector* modify the component *Camera*, for instance, the clipping planes and the type of projection.

[3] http://docs.unity3d.com/Manual/nav-InnerWorkings.html

1.3- Modify a world

1.3.1- Before you start

1.3.1.1- Who is able to modify a world and why?

Your organization is provided with a world composed of a set of objects and a set of scenes (scenario). Using the Scenario Editor, you are able to create graphically configurations of the *scenes* based on existing objects of the world. Using the *Task Editor*, you are able to define tasks in a specific configuration of your world.

If you are missing objects or behaviours in your world, you will need to ask for them to the SeniorLudens management board. They will create the objects, program the new behaviours and include them in your world.

In this document, we explain how to modify a world adding new objects and new functionalities. You will only be able to this task if you have access to the SeniorLudens Game Kit. You must be familiar with Unity3D to do it. Skip this document if you are simply willing to modify a configuration of the scenario and use the *Scenario Editor* instead.

If you need to create a new world from scratch continue reading this section and then jump to Section *Create a new world*.

1.3.1.2- Learn about the existing resources

The objects available in your world come from two repositories:

- the SeniorLudens repository, available in all the SL projects.
- your own project repository.
-

These repositories are in the Assets directory: warehouseSL and warehouseProject. Take a look inside and look the available resources. They are classified according to their category: food, accessories, construction etc.

Note: May be you already have what you need!! See SeniorLudens Warehouses section to know more about the structure of these repositories.

Whether you create new objects or modify them you need to know the concepts of SeniorLudens object, Visual Object, Unity Objects and Prefab.

1.3.2- Add an object to a world

In this section you will learn how to add an existing object stored in a warehouse to your world. If you want to create a new object from scratch, go to *Create a new object* section.

Adding an object to a world will allow users of the Scenario Editor to create instances of that object for different scene configurations.

There are two ways of adding an object to a world: either you add directly an instance of the object in a scene of the world, or you add the object to the world without creating any instance of it. In the latter case the object will not belong to the initial configuration of the scene, but it will be available in the Scenario Editor to create instances of it in other configurations. Keep in mind that in the Scenario Editor, you will only be able to manage objects that are in the world.

The core of SeniorLudens Game Kit implements a lot of actions such as to pick, to drop and to change state. However, some objects have very specific actions or autonomous behaviour that are not included in the core. These objects have they own scripts. When adding them in a world, you will need to add also their actions. Follow the instructions to do it.

To add an instance of an object to the scene:

1. Drag the prefab from the warehouse.

To add a new object to a world without adding an instance of it in a scene, follow the steps:

1. Open the scene of your world in which you are working
2. Select the empty Unity object that represents a scene of your world (*MainScene*, e.g.)
3. In the left-side panel, add a new entry to the *Warehouse* drop-down submenu: *increase the number of objects* drag the prefab of the new object to the new entry.

In both cases:

1. In the world definition xml file add the description of the new object (see *SeniorLudens project files* subsection). The description is available in the warehouse in the file *definition.xml* just open it and cut and paste in *MyWorld.xml*.

   Check if the object has update scripts to implement autonomous behaviours not included in the core. If so, they should be in a folder called *scripts*, and within it in a subfolder *cs*. They should be named after the object's name. This is the case, for instance of the objects *extractor*. Its update script reproduces the sound of an extractor when it is on. The script is called *extractor.cs*. Take a look at the scripts and check if they are defined on the Visual Object or on the object. In the former case, the prefab already includes the behaviour programmed in the script, so you don't have to do anything. This is the case of the thermometer has a *thermometer.cs* script, but defined on the *ThermometreVisual*. In the latter case, you should notify the existence of the behaviour to the world. For that, open the script *Configure* of MyWorld class definition, and add the sentence *app.AddType(MyObjectTypeID, typeof(MyObjectType)*. This is the case of the extractor. For instance:

```
public void Configure (SeniorLudens.Core.Application app)   {

    app.AddType("Extractor", typeof(Extractor));

}
```

2. Check if the object has action scripts to implement specific reactive actions. If so, they should also be in the folder *cs* of the folder *scripts*. They should be named after the action's name. For instance the object *egg* has the action *break* implemented in the script *BreakAction*. You must notify the world of the existence of this action. Open the script *Configure* of MyWorld class definition, and add the new Action (*app.AddAction(new MyActionName)*). For instance:

```
public void Configure (SeniorLudens.Core.Application app)   {

    app.AddAction(new SeniorLudens.Warehouse.Egg.BreakAction());

}
```

1.3.3- Modify existing objects

You can modify existing objects of your world by adding to them new visual styles, new states and new actions.

1.3.3.1- Create a new visual style for an object

Your world has a ball, looking as a football ball with black and white patches. You need a different style, with red and blue patches. The first thing that you need to think about is if the second ball is only a variation of the first one (a different visual style) or if it is actually a different object. For instance rugby balls and football balls may be different objects, because they may play a different role in a training game. If you need to create a different object proceed to Section *Create a new object*. Otherwise, continue in this section.

Objects have different visual styles: colours textures and graphical design. In order to add a new graphical style you need to know how to use and create materials and how to use and create textures.

1. First, check if you are creating a new style that didn't exist previously in your world or if you are simply creating an existing style for your object. In the former case, thing carefully the name of your style. Choose it in a way that it will be clearly identifiable. In the second case, use the existing style name.
2. Create the new 3D model that will represent the object's new style. A new style can be a change of material and texture or even a completely new mesh.
3. Save the new model in the folder *styles* following the schema warehouses.
4. If you are using Blender, do not forget to unpack the textures and store them in the folder *textures* in the object's folder. If you are using another modeler, separate the textures in a similar way.
5. Add the new style in the object's definition file *definition.xml*.

```
<states>
</states>

<style name="kumato"> <!-- kumato will be the id of the
style -->

        <name>Kumato</name> <!-- Style name -->
        <description>Tomato variant named
        Kumato</description>

        <!-- Definition of the style's tag:
                - fruit: because a tomato is a fruit
                - red: the color of the tomato.
        -->
        <tags>fruit, red</tags>

<!-- For texture changes: -->
        <visual state="STATE_NAME">
                <texture>TEXTURE_IMAGE</texture>

        </visual>
    <!-- For mesh changes: -->
        <visual state="NOM_DE_L'ESTAT">
                <model>NOM_DEL_.BLEND</model>
        </visual>

</style>
```

6. Import the object in the *Unity3D* scene and create the *prefab.*

1.3.3.2- Create a new state of an object

Suppose that the tomatoes of your world have only one state: *full_raw*. You need to implement the action *cut* and add the state *half_cut_raw* to the tomato if you want to design tasks requiring to cut tomatoes. Let's do it.

7. Create a new state. It can include:
   - Texture modification: either in your object's modeler or directly in Unity3D.
   - Mesh modification: create a new 3D model with you 3D modeller.
   - New animation: create the animation with your 3D modeler.
8. Create a prefab for every state
9. Modify the object's definition_file and the *Add the definitions of the structural objects in the file ProjectWorld.xml* section. For instance, the

following definition file includes two states for a door object:

```
<objectdef id="door">
<states default="closed">
        <state name="closed">
                <action name="open">
                        <param name="animation">open</param>
                        <param name="state">opened</param>
                </action>

                <action name="navigate" function="place"/>
        </state>
        <state name="opened">
                <action name="close">
                        <param name="animation">open</param>
                        <param name="state">closed</param>
                        <param name="reversed"
                        type="bool">true</param>
                </action>

                <action name="navigate" function="place"/>
        </state>
</states>
</objectdef>
```

1.3.3.3- Add an existing behaviour to an object

Object's behaviour depends on the actions that are enabled on them. In this section, you will learn how to assign an existing action to a new object. For instance, add the action of disintegration to another existing object.

1. Open the file *definition.xml* of an object that already has the desired behaviour and copy the definition of the desired action.
2. Open the file *definition.xml* of the object to which you want to add the action. Paste the code to every state that you want to have these actions. For instance:

```
        <state name="full" model="apple">

            <!-- for the state 'full' we already had the actions
            navigate, pick an

            <action name="navigate" function="place"/>
            <action name="pick" function="directobject"/>
            <action name="drop" function="directobject"/>

            <!-- and we now add the action destroy -->

            <action name="destroy" function="directobject"/>

        </state>
```

3. Check the definition of the parameters.

1.3.4- Create a new action for an object

If *SeniorLudens Gamekit for Unity3D* does not include any action that matches your needs, you'll need to implement the new action and assign it to your object. For this you will have to do:

Decide the identifier of your action, see *How to create an identifier for an action*, as example from now on we use **myAction** as identifier.

Create the scripts for the action in C#, `Scripts/cs/idCore.cs` and `Scripts/cs/id.cs` inside the warehouse folder of the object, where *{id}* is the capitalized version of the identifier in this case **MyAction**.

`Scripts/cs/MyActionCore.cs` will define the action definition and the non-visual behavior of your action.

`Scripts/cs/MyAction.cs` will contain the implementation of the action that will need access to the visual layer logic, in this case Unity3D.

Take a look at the *SeniorLudens API* or the source code of the default actions to know more about how to implement an action.

Add the new action to the definition of the object in the object definition file `definition.xml` as well as in the project definition file `MyProject.xml` [15].

Register *MyAction* and *MyActionCore* in your configuration.

1.3.4.1- How to create an identifier for an action

We recommend to use a verb if it is possible, then the id will be the verb in infinite and in lowercase, for example if your verb is "to break", your identifier will be: **break**.

If your identifier is composed by more than one word you will join them capitalizing any word except the first one, for example if your action will be 'put At', the id will be: **putAt**.

Identifiers are case sensitive **putat** is not the same as **putAt**.

1.3.4.2- Examples

Along this section we demonstrate how to create actions with some examples. The actions provided by the *SeniorLu- dens Gamekit for Unity3D* complement this section offering.

Example 1 In this example we will create a simple action which will display a message when its called, the id for the action will be **myAction**.

First create the core for your action:

```
using System.Collections.Generic;

using SeniorLudens.Core.Action;
using
SeniorLudens.Core.Acti
on.ActionCalls;   using
SeniorLudens.Core.Data
.Definitions;



public class MyActionCore : BaseAction {

        protected       readonly       ActionDef       definition       =
        SeniorLudens.Core.Data.Xml.WorldXmlReader.Read
                @"<action name='destroy'>
                        <function name='directobject' control='true'/>
                        <function name='subject'/>
                </action>");

        public override ActionDef GetDefinition () {
                // Define the definition, you can do it programatically
                calling (new ActionDef()
                // or using the xml definition of the action.
                return definition;

        }

        public override bool CanDo (ActionSentence s) {
                return true; // Is there any restriction with the objects
                that can be checked by
        }

        public   override   IActionCall   Call   (ActionContext   context,
        ActionSentence sentence) {
                // Create a new visual action call.
                return       new       VisualActionCall(sentence.direct,
                definition.id, context.actionParams
        }

}
```

Now create the visual-action:

```
// Here we can use anything from UnityEngine
using UnityEngine;
```

```
using SeniorLudens.Core.Action;
using SeniorLudens.Unity.ActionComponents;


public class MyAction : ActionComponent {

        // When extending an ActionComponent you have to provide the id
        //  of the action to the constructor.
        public MyAction() : base("myAction") {
        }

        public override void Call(VisualObject object3d, ActionParams
        param) {
                // As we have access to the UnityEngine API, we can use
                Debug.Log
                //  to display a message on the console.
                Debug.Log ("MyAction called on object: " + object3d);
        }

        public override bool IsDone(){
                // After displaying a message the action is done
                return true;
        }
}
```

Finally add those scripts to your configuration file (highlighted lines):

```
public class MyProjectConfiguration : SeniorLudensUnityAppConfiguration {

        public override void Configure
                (SeniorLudens.Core.Application app) {
                app.AddAction(new MyActionCore());
        }

        protected override IEnumerable<Type> GetCustomActionComponents() {
                return new Type[] {
                        typeof(MyAction)
                };
        }
}
```

Example 2: Using GUIAction Here we will skip the creation of `MyActionCore.cs` and use the GUIAction as the core implementation for our action, this is a fast way to create a simple action with all the logic on the visual layer, however it only can be used on objects as a direct object action.

First create the visual-action:

```
// Here we can use anything from UnityEngine
using UnityEngine;
```

```
using SeniorLudens.Core.Action;
using SeniorLudens.Unity.ActionComponents;


public class MyAction : ActionComponent {

        // When extending an ActionComponent you have to provide the id
        //  of the action to the constructor.
        public MyAction() : base("myAction") {
        }

        public override void Call(VisualObject object3d, ActionParams
        param) {
                // As we have access to the UnityEngine API, we can use
                Debug.Log
                //  to display a message on the console.
                Debug.Log ("MyAction called on object: " + object3d);
        }

        public override bool IsDone(){
                // After displaying a message the action is done
                return true;
        }
}
```

Note: Yes, the action implementation is the same as in example 1.

Now we will add the action to the configuration of your project:

```
public class MyProjectConfiguration : SeniorLudensUnityAppConfiguration {

        public override void Configure
                (SeniorLudens.Core.Application app) {
                app.AddAction(new GUIAction("myAction"));
        }

        protected override IEnumerable<Type> GetCustomActionComponents() {
                return new Type[] {
                        typeof(MyAction)
                };
        }
}
```

1.3.5- Remove objects from your world

In general, objects should not be removed from a world unless they are erroneous. You never know if you may need them in a future scene!

If you really want to remove an object from your world, make sure that there are no instances of it in any of the world's scene. Then, remove its definition from the cml world definition file and, if it has actions and specific behaviour, remove them from the Configure method of the world configuration class.

1.3.6- Create a new object

You can create new objects either from scratch or by taking as a reference existing objects. The latter way is faster and convenient whenever you need to create new objects of a same family with different aspect and name but with similar behaviour, for instance an apple having already a pear.

1.3.6.1- Create a different object similar to an existing one

Now, imagine that you need to create a rugby ball as a different object from the already existing football, or, your world has tomatoes, but you need peppers. A pepper is essentially the same object as a tomato, it has the same actions (to pick, to drop...) but with different name and aspect. Let's create an object using another as reference.

1. Create the new 3D model
2. Unpack the textures that are needed for the new object and save them in the folder *textures* within the object's folder.
3. Copy the file *definition.xml* of the object that is similar to yours and modify it to adapt it to your object. Save the file in the object's folder.
4. Import the object to *Unity3D*:
    - As soon as *Unity3D* will open, the folder *Materials* will be created that contains the materials that the object uses.
    - Save it as a .prefab in the folder of the new object.

1.3.7- Create a new kind of object

1. Check that the type of object that you need is not currently provided by SeniorLudens GameKit. Recall that several different objects can be of the same type. For instance, there may be different objects of type "door" or "chair".
2. Create a suitable name to the new type of object. Try to be precise, thinking that even now your object is the first of a category, may be you'll add others of the same category later on, so don't call it after its category but after its intrinsic nature. For instance, if you put the name "fruit" to an apple, then you will not be able to distinguish between fruits. Call it apple.
3. Add to the object definition file all the states and attributes it has. Modify accordingly the *MyProjectWorld.xml* file. For instance:

```
<objectdef id="OBJECT">

<states default="DEFAULT_STATE">

        <state name="STATE_1">
                <action                                 name="ACTION_1"
                function="FUNCTION_ON_SENTENCE"/>
                <action                                 name="ACTION_2"
                function="FUNCTION_ON_SENTENCE"/>
                <action />
        </state>

        <state name="STATE_2">
                <action                                 name="ACTION_1"
                function="FUNCTION_ON_SENTENCE"/>
                <action/>
        </state>


</states>

</objectdef>
```

Add the description of the new object type to the documentation.

1.3.8- Create a component

The scope of a component defines if the component can be used only in one object or if it can be re-used in other objects. In the former case, the component is called *local*, otherwise it is called *global*. This section covers how to work with *local* components. To create a *global* component, please go to Section Use an object as component.

The definition of a local component is inserted directly in the definition of the object that contains it. To add a local component to an object definition just add a *component* element with the attribute “name” with the name of the component:

```
<component name="[name]">
        ...
</component>
```

Within the component block put the definition of the component which is like any other object definition.

If your component is *local* but it appears in your object multiple times with different names, you will write its definition only once within a *component* element as described above, and for the other occurences you will use a *componentref* to refer to this complete definition of the component.

```
<componentref name="[name]" ref="this.[reference name]"/>
```

1.3.8.1- Use an object as component

1.3.8.1.1- Add the component to the object definition

To use an object as component in the definition of another object you have to use the *componentref* element:

```
<componentref name="[name]" ref="[object id]"/>
```

1.3.8.1.2- Add the visual object as a component of your object [unity]

To use an existing object as a component you have to include the object as a piece of the new object and modify its properties to act as a component:

1. First of all create a instance of the object to use as component inside the main object (drag the prefab of the object acting as component from the warehouse to the main object in the scene).
2. Change the script of *VisualObjectInstance* unity component of the new instance for a *VisualComponentInstance* script.
3. Complete the *Component Name* and the *Component Form* fields with the values that you have defined in the *componentref* element of the object definition. For example:

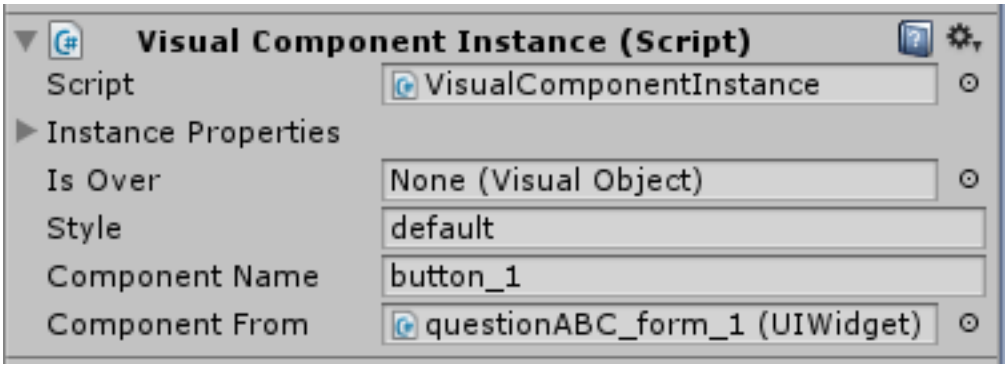

Figure 5 – Add visual object to a component

1.3.8.2- Create a form

Forms are used to display messages, questions and other similar 2D panels. They consist of a 2D layout on top of which different elements can be organized: images, text, input areas etc. The warehouseSL provides standard forms that you can use directly or that you can take as a basis to create new ones. Before creating a new form, check if already exists in warehouseSL (*objects -> GUI -> UI -> forms*). If so, follow the instructions of Add a form to your scene. Otherwise, keep reading this section.

1.3.8.2.1- Create a new form from scratch

Design a form

We recommend you to design first your form in a piece of paper and after that, think about how the user will interact with it, then you can start create the definition.

Create the definition

The organization of the different elements that compose a form is based on the concept of Components. A form is an object composed of different *components*, on which the user or the trainer (system) can interact independently. For instance, a basic existing form is the Message Form. It has four components: the background, the panel, the title, the content (text) and a confirm button.

As you have already designed your form on paper in the previous step, now and before starting to write the object definition you have to detect if each element composing the form is a component or not. For this you have to follow the following rules:

Does the element change if the user interacts with it?

Yes, it does -> It is a component or it is the form object itself.

No, it does not.

Can it be changed by the trainer (system)?

Yes, it can -> It is a component.

No, it cannot -> It is a graphical element of the form

Now that you know which elements are the part of form and which are components, you can start writing the definition in the same way as for any other object in the platform. Before creating any new component, check if it is already provided in the warehouseSL. If so, go to the section *Add a component to a form*, otherwise, go to the section *Create a new form component*.

Create a new form component When you create a new form component, you have to decide wether it will it be a component exclusive for this form or if it can be shared with future forms. If you're not sure about that, we recommend you to make it exclusive for this form.

If you have decided to make it exclusive to this form, follow the instructions Create a new component. Otherwise, read Create a new object and after Use an object as component.

Note: The components provided by the *SeniorLudens* warehouse are built on top of Unity's UI.

Add a component to a form In the warehouseSL there are also some examples of components ready to add to your forms (*objects -> GUI -> UI -> forms -> widget_examples*), for instance:

- 'form_label'
- 'form_image'
- 'form_button'

Once you find the object to use as component, you have to follow the section Use an object as component to add the component to your form before continuing with these instructions.

Create the visual representation of a form in *Unity 3D* Now you have the definition for your form, it is time to create the interface in *Unity 3D*, your form is composed by components and some other graphical elements from the object itself.

Create a new form from an existing one

You can create a new form by adding an existing one, modifying it and storing it in the warehouse as a new one. You will find existing forms in theSeniorLudens warehouse: *objects -> GUI-> UI -> forms.* Besides, in *objects -> GUI-> UI -> forms->questionsABC_form->examples* you will find examples of scenes and of tasks using the *questionABC* form.

To add an existing form, follow the steps described in see Add a form to your scene. You can modify it by removing or adding components (see *Add a component to a form*) and modifying the visual representation of the object in *Unity 3D.*

Once you have the new form, drag it to the warehouse (in order to create the prefab) and create the definition file with all its components (as in Add an object to a world).

1.3.8.3- Add a form to your scene

Note: SeniorLudens Warehouse provides some forms. You can add them to your scene as they are or modify them to create new ones.

To add a form to your scene, drag the prefab [16] to the Canvas of your scene. Once you add a form you can adapt it to your scene look and feel, you can modify its appearance changing the images of the panels and buttons or the size of the elements. If you modify it remember to create a new style of the object in the warehouse! Then, add the definition of the object into your world description and add the action fillFormAction in your world configuration file (see Add an object to a world to learn how to to these two steps).

Note: If your scene does not have a canvas:

Create it in Unity: *GameObject → UI → Canvas*

Change the *Render Mode* of the *Canvas* component to *Screen Space - Camera* and assign the GUI Camera of your scene.

1.3.9- Modify a scene

Use the *Scenario Editor* to create configurations of the scene: create, remove and move object instances of your world.

If you want to modify structural elements of the scene, that you cannot edit in the *Scenario Editor*, follow the steps of Section add new objects.

1.3.10- Add a new scene to your world

This section explains how to register new scenes. Currently, SeniorLudens allows you to have only one scene aside from the introduction and conclusion, so be aware that this documentation can change a lot.

1. First, create the scene. Do not forget to register it.
2. Then in the menu File -> Build Settings... -> drag the new scene from the Project panel to Scenes in build.

Note: If the scene is the beginning or the end of a task, you should add it to the *ProjectApp* component: With the object *ProjectApp* selected, in the *Inspector* panel, add the scene to the suitable component of *ProjectApp* (e.g. to *End Level* if it is an end-task scene)

1.3.11- Build a game

To release a new version of a game you have to build it for each supported platform. However, *SeniorLudens Platform* only has support for games built for Unity Webplayer, the *SeniorLuends Gamekit SDK for Unity3D* provides also support to build webgl games. [17]

SeniorLuends Gamekit SDK for Unity3D can be used to build the game. First go to Window → SeniorLudens → SDK window, then choose the build version (development, production) for each platform:

- Webplayer: builds the game as unity webplayer.
- WebGL (experimental): builds the game as webgl.

Important: If the game is build using *File → Build Settings* or *File → Build & Run*, then the world descriptor will not be generated.

After choosing a platform the game will be built and stored on the *SeniorLudens Gamekit:*

`webcontent/static/games/project_id/version_platform`

Or for development builds:

`webcontent/static/games/project_id/version_platform_dev`

Note: Development or Production? A game for production normally provides a small executable, but it's harder to debug. Our recommendation is to build the game first for development, test it (see *test a game*) and then build it and publish it on *SeniorLudens Platform*, see *upload a game*.

1.3.11.1- Test a game

Games built with the *SeniorLudens Gamekit* need a server to work, this server could be the *SeniorLudens Platform* or the server provided by the *SeniorLuends Gamekit SDK*.

The game will be tested twice, first with the *SeniorLuends Gamekit SDK* and after, if everything works fine, it will be uploaded to the platform to do the second test.

For the first test, the game needs to be built for webplayer using the development build, as explained in *build a game*. Then:

1 Open the SDK website, usually at http://localhost:5000
2 Create a new scenario configuration with the scenario editor.
3 Test the game with more than one task

If all the steps went fine, it is time to upload the game to the game to *SeniorLudens Platform*.

1.3.11.2- Upload a game

Please, refer to the *SeniorLudens Platform deliverables* to know in detail how to upload a game.

1.4- Migrate a Unity Scenario

If you have already built a *Unity3D scenario* and you want to use it in SeniorLudens, it may be better to start first by creating a very simple world from scratch in order to understand the processes and concepts: *Create a new world* section. Once you'll have done this first experiment, you'll be ready to follow these instructions:

1. First install the platform: *Installation*
2. Apply the basic settings: *Basic setting for the creation of a new world*
3. Create the projects files: *SeniorLudens project files*
4. Add an empty world object and define the world: *Define the world*
5. Create an empty scene object and define it: *Create a scene*
6. Create a new warehouse for your project: warehouses.
7. Add the objects of your scenario to the warehouse: for every object, create a prefab (if needed), store it in the warehouse following SeniorLudens pattern and create the corre- sponding definition files: *Create a new object*.
8. For all the prefabs add the *VisualObject* component: Unity_objects.
9. Add the definition of all the objects in the world definition file.

2- Scenario Editor

2.1- Introduction

The Scenario Editor is part of the SeniorLudens GameKit. Users that create scenario configurations are Trainers with the corresponding permissions. The Scenario Editor is implemented as a SeniorLudens game. Thus, it does not require programming skills.

The Scenario Editor is the SeniorLudens tool used to create different scenario configurations out of a predefined SeniorLudens World. There is one Scenario Editor Game for each SeniorLudens World. Each configuration is based on the same structure (World) and differs only by the set of tangible objects that are located within the scenario. Variations introduced by scenario configurations are essential to avoid player's boredom and to promote adherence to the games but also allow different learning goals in the same environment.

2.2- Installation

The Scenario Editor is available as a prefab at the SeniorLudens Warehouse, in order to include it in your world drag the prefab into your scene.

Now you have to configure some options of the scenario editor, as the scenario editor is build using a Canvas element you have to define the camera and the render mode; we recommend you to use the camera named GUI from the avatar and a distance of 80.

2.3- Access

The Scenario Editor is integrated in the SeniorLudens platform and therefore can be accessed from within it. The user needs a username and password with the correct access rights (role) set to access it.

The steps that users must follow to access the Editor are the following:

1. To access the Scenario Editor the user has to direct their browser to the url of the SeniorLudens platform where he can enter his credentials. In order to access the editors the user must sign in as a Manager (Checkbox).

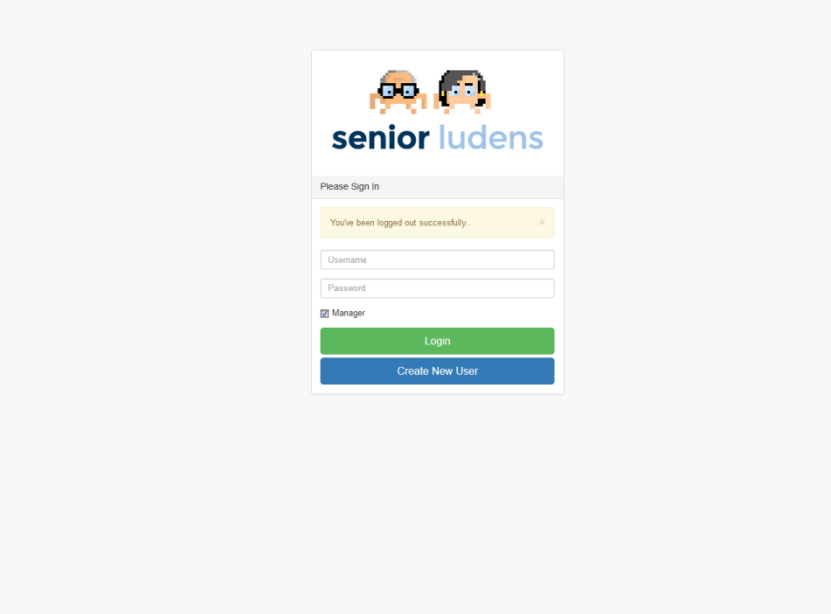

Figure 6 – Login View on SeniorLudens platform

2. After the sign-in process the SeniorLudens platform opens and the menu is shown on the left side of the screen. In the block Creation the ScenarioManger is listed.

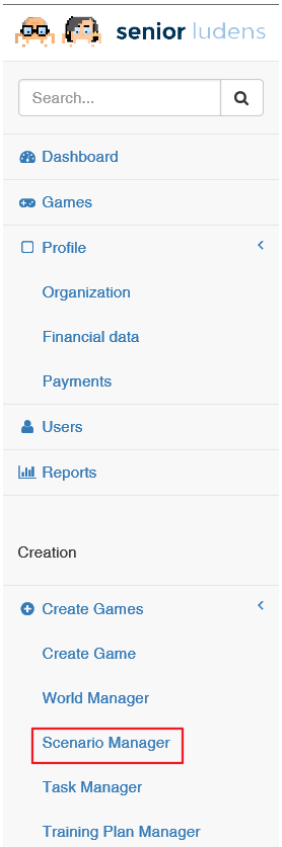

Figure 7 – Access to Task Manager

3. With a click on Scenario Manager, it is opened an overview on all scenarios that have been created and to which the user has access.

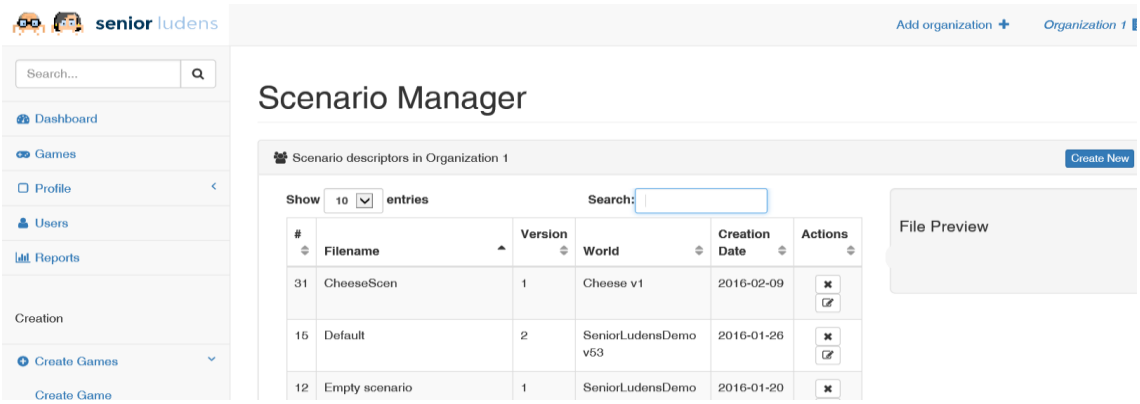

Figure 8 – Scenario Manager

4. Within the Scenario Manager the user can:

   - Create an new scenario from an existing world Create New
   - Edit an existing scenario by clicking on the icon.
   - Delete an existing scenario by clicking on the icon.

When the user is clicking on the Edit icon the Scenario Editor is started and the scene can be changed as described in Features.

Figure 9 – Scenario Editor

2.4- Features

2.4.1.1- Show and hide the Menu System

When the Scenario Editor is started the scene is shown in game mode (as it is seen by an end users). By pressing the Alt-Key the Menu System is activated.

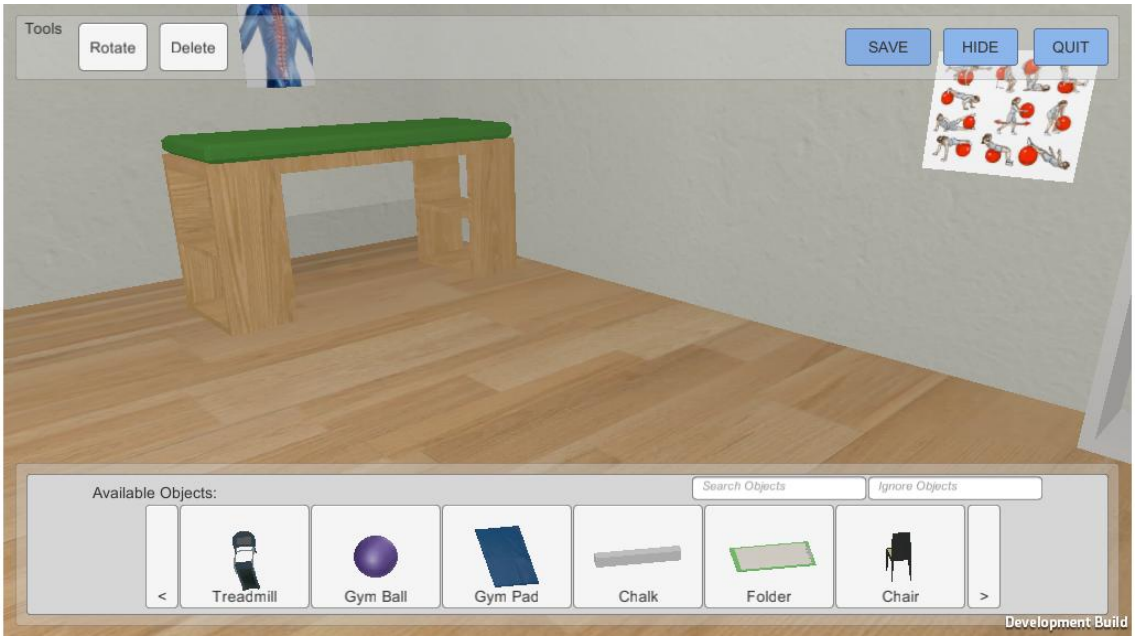

Figure 10 – Scenario Editor Menus

The Menu System is divided into two parts. An upper part with object related functions on the right side and Menu related buttons on the left side and a lower part where the object browser is shown.

The Menu System can be closed by clicking on the HIDE-Button.

2.4.1.2- Show and filter available items

When the Menu System is activated the object browser is shown at the bottom of the screen. The object repository might contain quite a lot of objects. You can browse through all the objects by using the arrows on the left and right.

Searching manually for an object might become a lengthy process. To speed this up the object browser has two filters.

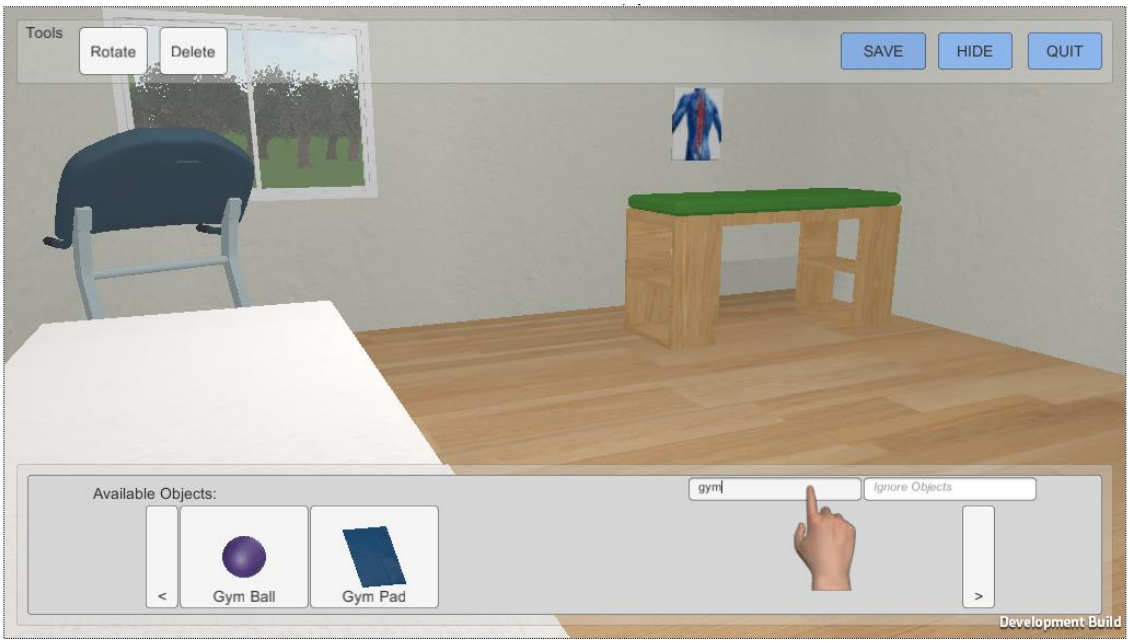

Figure 11 – Scenario Editor Search filters

The left filter is a search filter and shows all items that contain the entered characters.

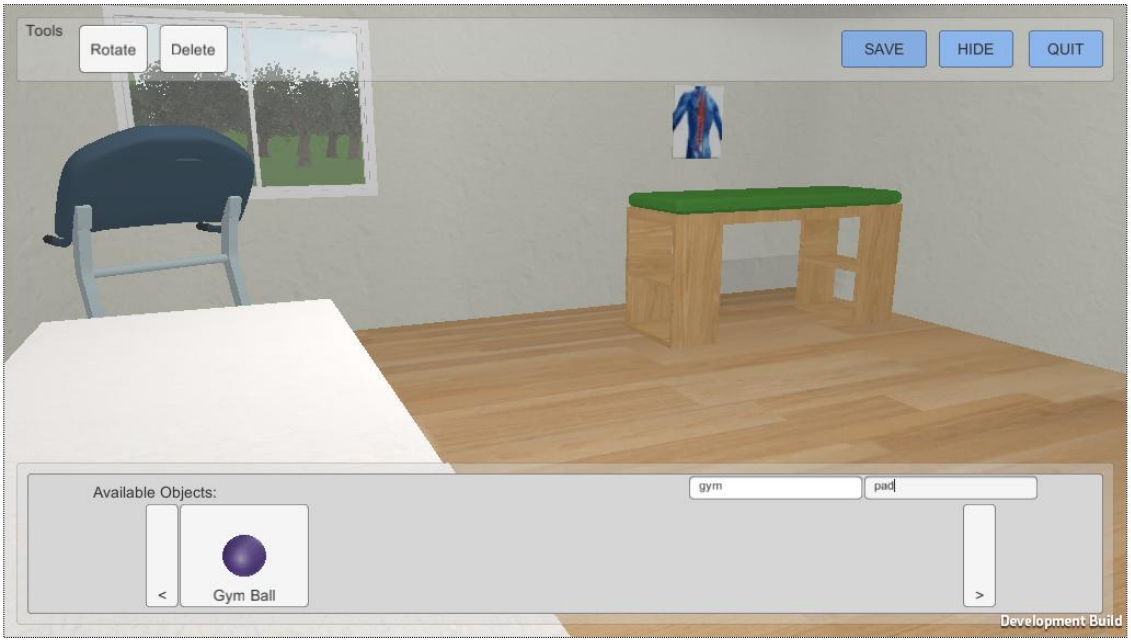

Figure 12 – Scenario Editor Negative filters

The right filter is a negative filter and is excluding all items that contain the entered text.

In order to find an object the two filters can be combined as seen in the two previous pictures. One filter is filtering the result set of the other one. This allows the user first to limit the available objects to a series of objects e.g. Gym related and then to exclude objects he does not want to see e.g. pad.

2.4.1.3- Select an item

Once an object that the user wants to be available in the scenario is found it can easily be grabbed by just clicking on it.

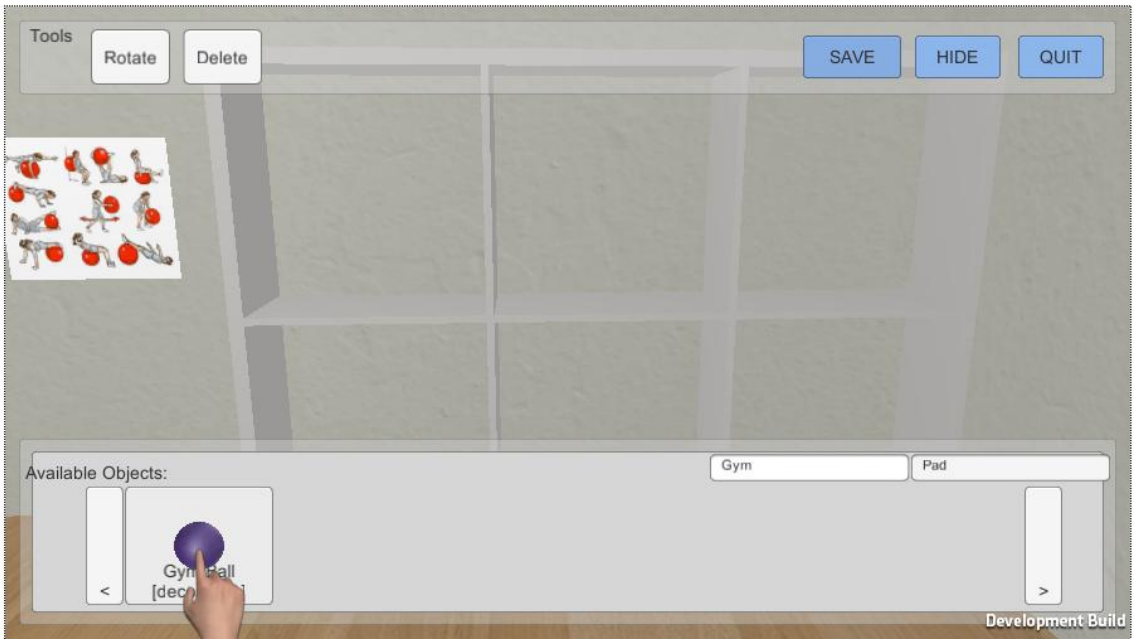

Figure 13 – Scenario Editor Item Selection

When an item in the object browser is clicked it appears in the middle of the screen and becomes the cursor. At the same time the menu system is automatically closed.

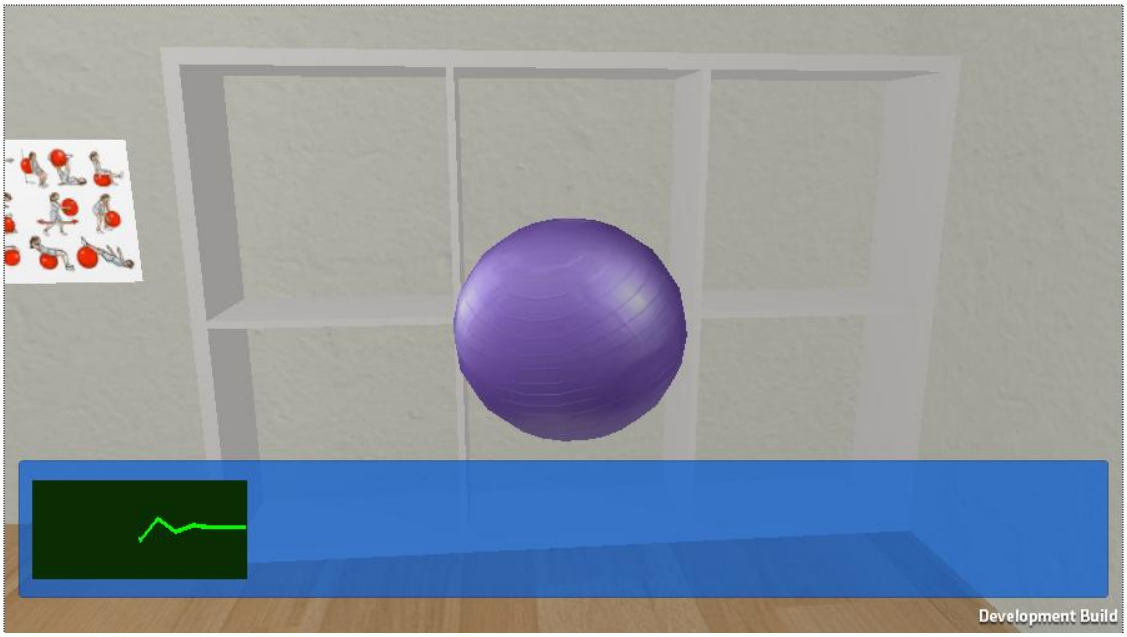

Figure 14 – Scenario Editor Item Release

It is recommended to decide where to put the object before it is picked from the object browser and to make sure that this position is visible on the screen beforehand.

2.4.1.4- Drop an item

When an object is selected as shown in the previous images, it can be positioned within the scene by clicking on the position where it should be.

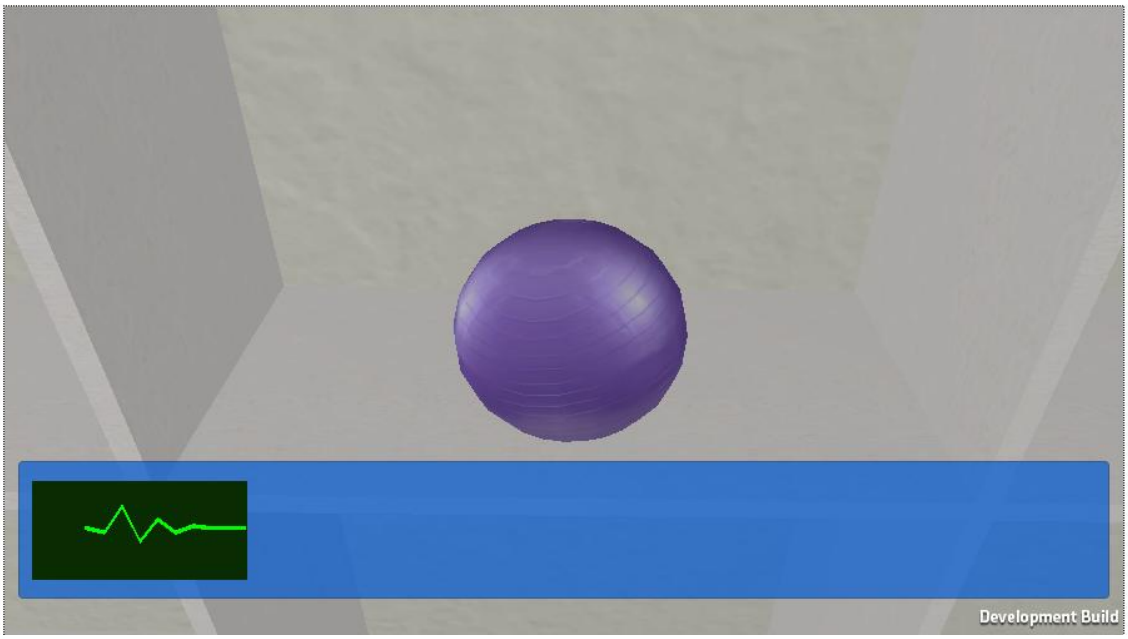

Figure 15 – Scenario Editor Drop Item

The object is automatically moved to the position on which the user has clicked.

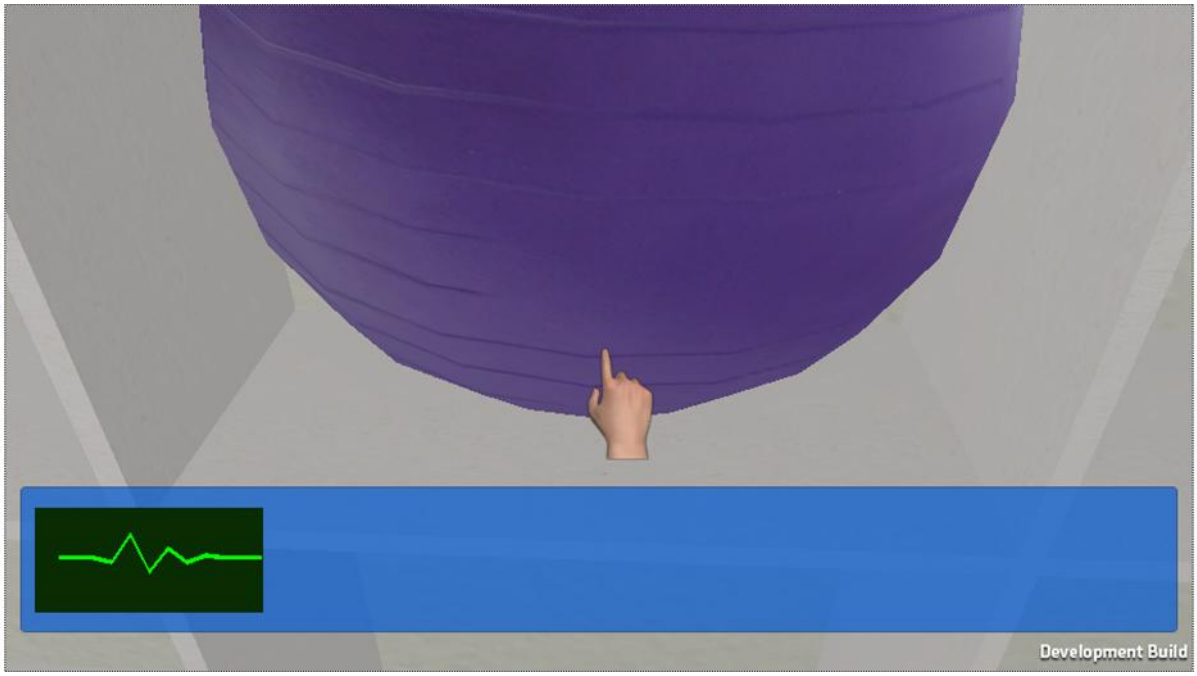

Figure 16 – Scenario Editor Drop Release

When it is successfully positioned the cursor changes back to its hand shape.

2.4.1.5- Rotate an item

When the rotation of an object needs to be changed the user first has to make sure that the object is visible on the screen. If this is not the case the view within the scenario can easily be changed by using the mouse.

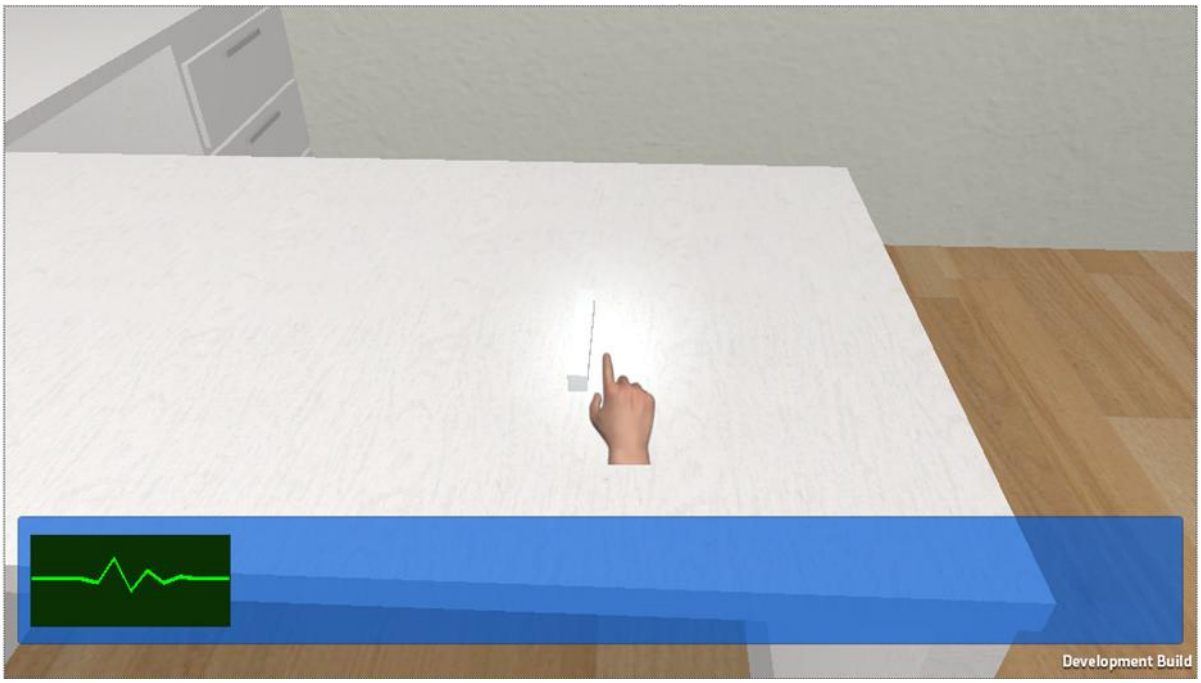

Figure 17 – Scenario Editor Rotate Item

Once the item is visible on the screen the menu system must be activated by clicking on the Alt-Key.

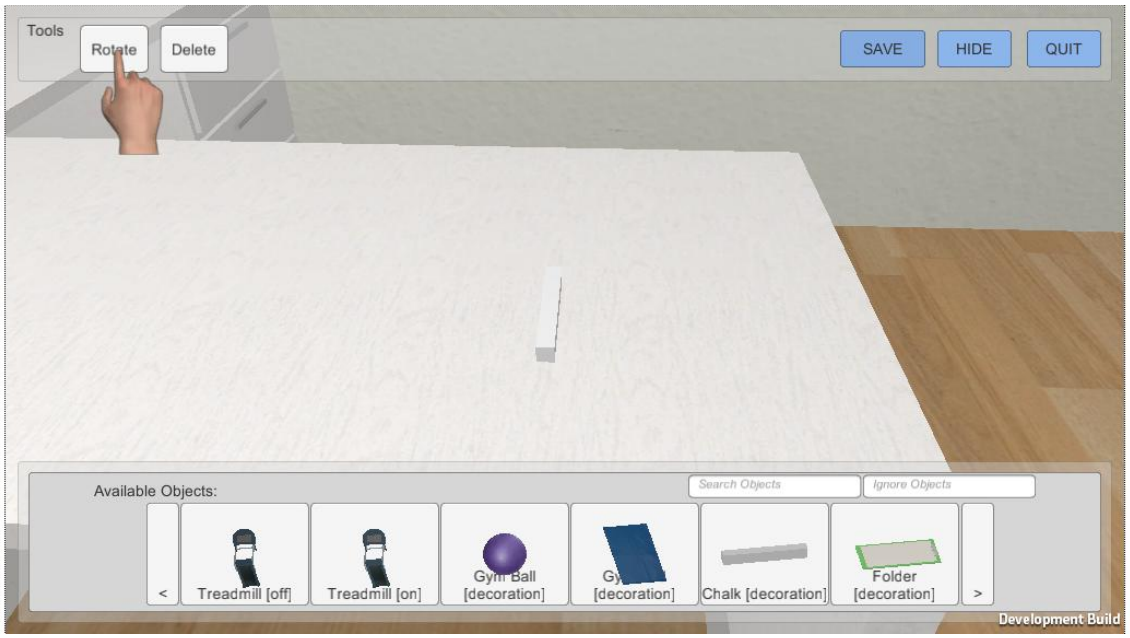

Figure 18 – Scenario Editor Rotate Action

In the top bar of the menu system the user must click on the button Rotate.

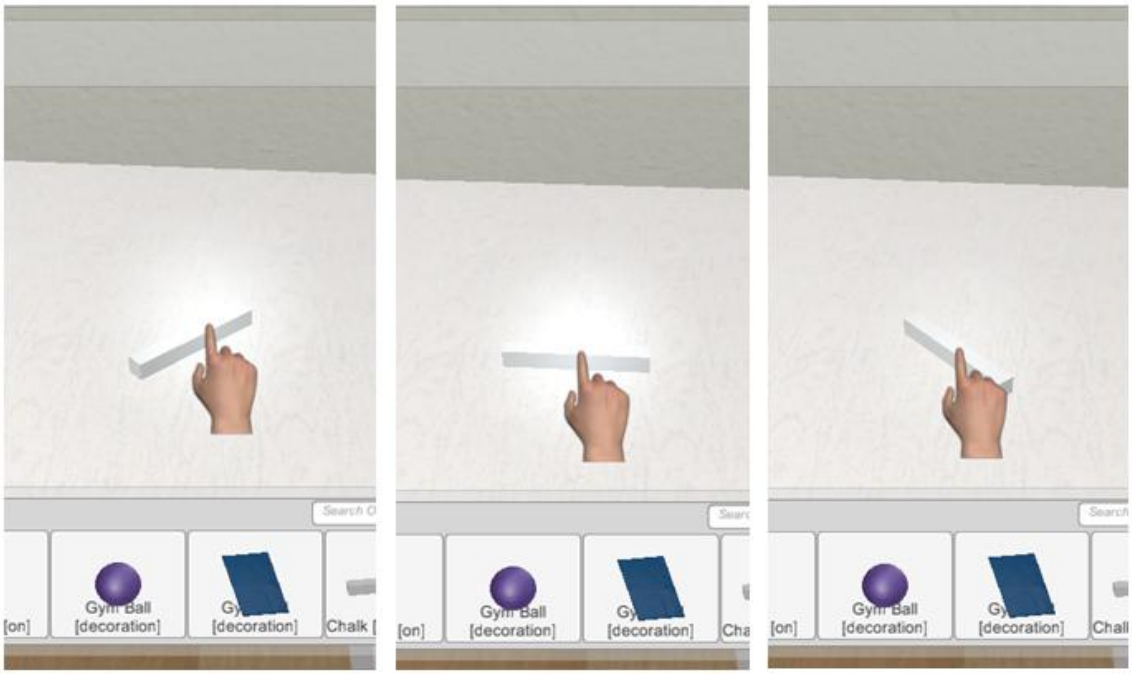

Figure 19 – Scenario Editor Rotate Action on item

When Rotate is activated, each click on the object rotates it by 45 degrees.

2.4.1.6- Delete an item

When an object needs to be deleted from the scene the user first has to make sure that the object is visible on the screen. If this is not the case the view within the scenario can easily be changed by using the mouse.

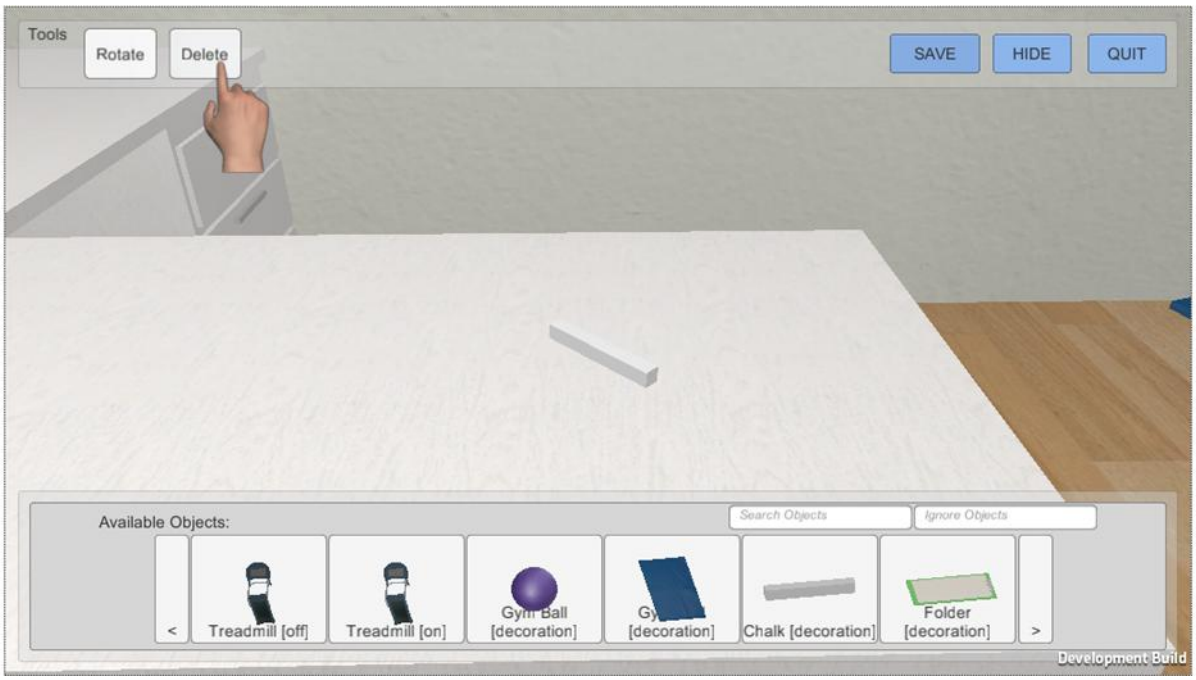

Figure 20 – Scenario Editor Delete Item

Once the item is visible on the screen the menu system must be activated by clicking on the Alt-Key and the delete action can be activated by clicking on the Delete button in the upper bar of the menu system.

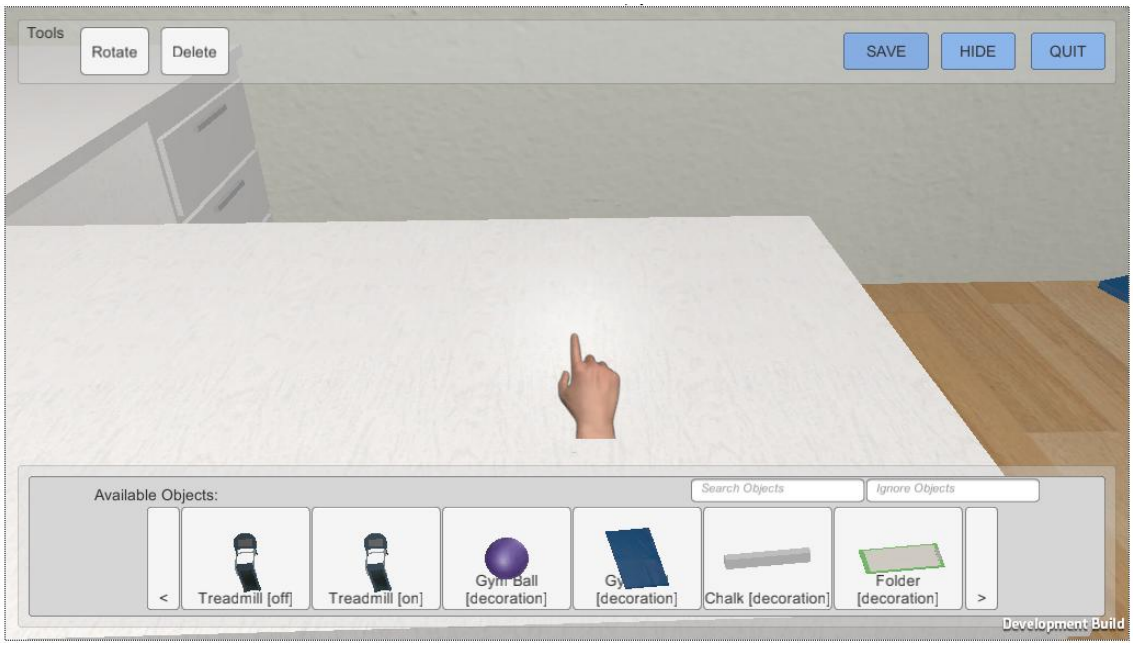

Figure 21 – Scenario Editor Delete Action

Once the delete action is activated and the user clicks on the item that he wants to disappear from the scene, the item is deleted.

2.4.1.7- Save the configuration

From time to time the changes done within a configuration should be saved.

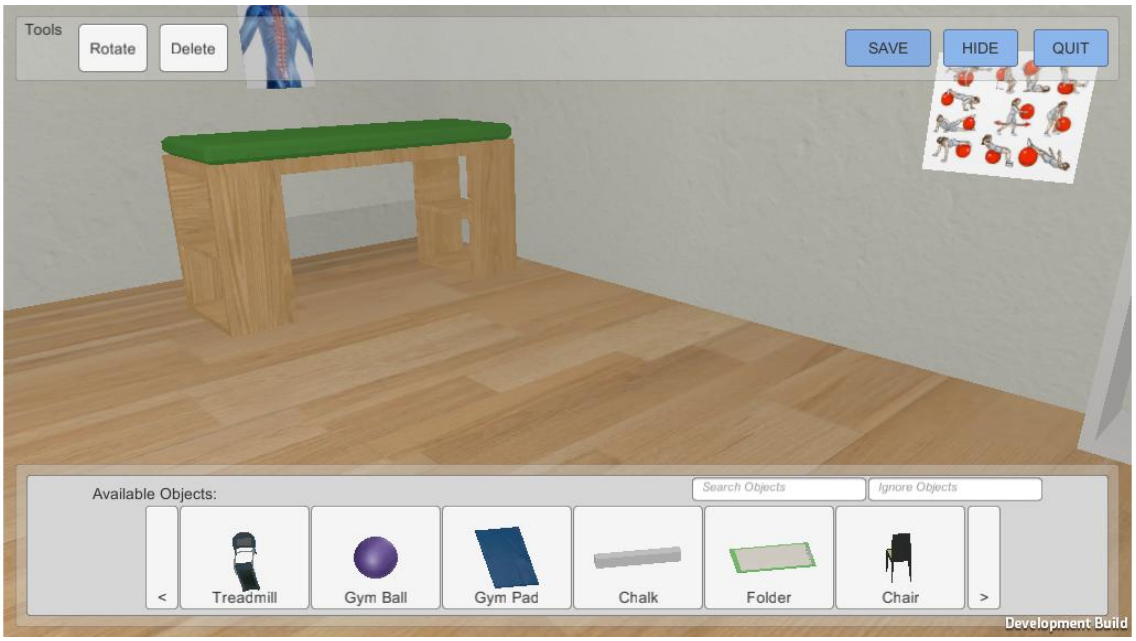

Figure 22 – Scenario Editor Save configuration

In order to save a configuration the menu system must be activated by pressing the Alt-Key. A click on the Save button in the upper right corner of the menu system saves the actual status of the scene.

3- Task Editor

3.1.1- Reference site

The task editor is deployed in an intermediate server located at: http://selte.cbim.it/**.** However it is integrated inside the SeniorLudens Platform infrastructure by using the SeniorLudens Storage Server. The Editor is accessible through the platform url: http://demos-innovation-labs.com/sl/ .

3.1.2- Introduction

The task editor is the tool used by the trainer to design the reference task for the trainee and define the different roles of the characters.

Deploying the full state diagram of all possible user actions is very tedious and prone to errors. Therefore, the task editor tool will require trainers to define only the reference task, this is the correct way of doing things.

For the reason Task Editor Tool makes use of Blockly as Visual Editor that allows users to write flows by plugging blocks together.

The reference task is defined in terms of actions structured as sequential or parallel compositions. Sequential compositions mean that the actions must be done one after the other, and parallel compositions mean that a subset of the actions of the bloc must be done no matter in which order. During the game play, all user interactions are interpreted as action queries. The action queries are evaluated in comparison to the reference task to know if they are correct or no. If they are correct, they are done. Otherwise, they can be done and evaluated as incorrect or forbidden to provide a free-of error learning process.

3.1.3- Access

Task Editor is integrated in SeniorLudens platform, so the primary access is needed to be performed through it. The user needs to validate its credentials with the system, in order to have granted access into the SeniorLudens Security system. By doing so, the users identify themselves with the specific roles that they have in the organization. The steps that users must follow to access the Editor are the following:

5. To access Task Editor the users introduce their credentials into login view, as can be seen in the figure below. It is important to tick in the manager checkbox, because the Editor is only accessible through the Management Portal. The users must own a manager role in the system to access the Management Portal.

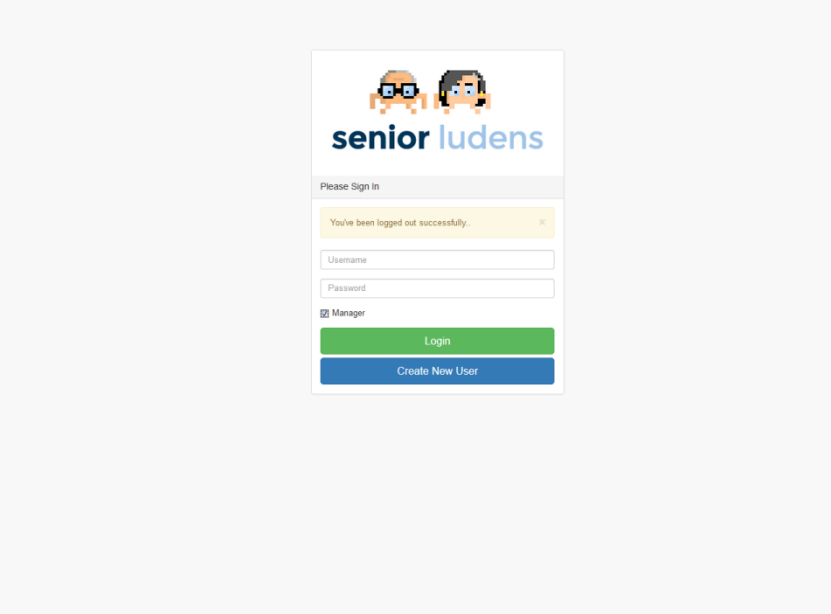

Figure 23 – Login View in Platform

6. Once the user has been validated in the system as manager, they can visualize the dashboard of the organization. After this step, they can click on the Task Manager option located in the side menu.

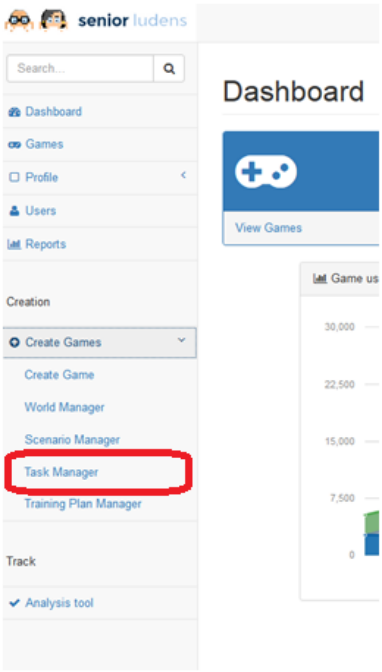

Figure 24 – Access to Task Manager

7. After selecting this option, the Task Manager is open. This view permits the managers to organize all the task descriptors defined in the system. Among the features of the view, it permits two essential actions directly related with Task Editor:

   - Edit existing descriptors: This action is performed when the user selects the button edit included in each row of the manager. If the user clicks on this button, the Task Editor will be loaded with the specific information of the selected task descriptor.

- Create new Task descriptors: This action is activated by clicking on the button create, located on top of the Task Manager View.

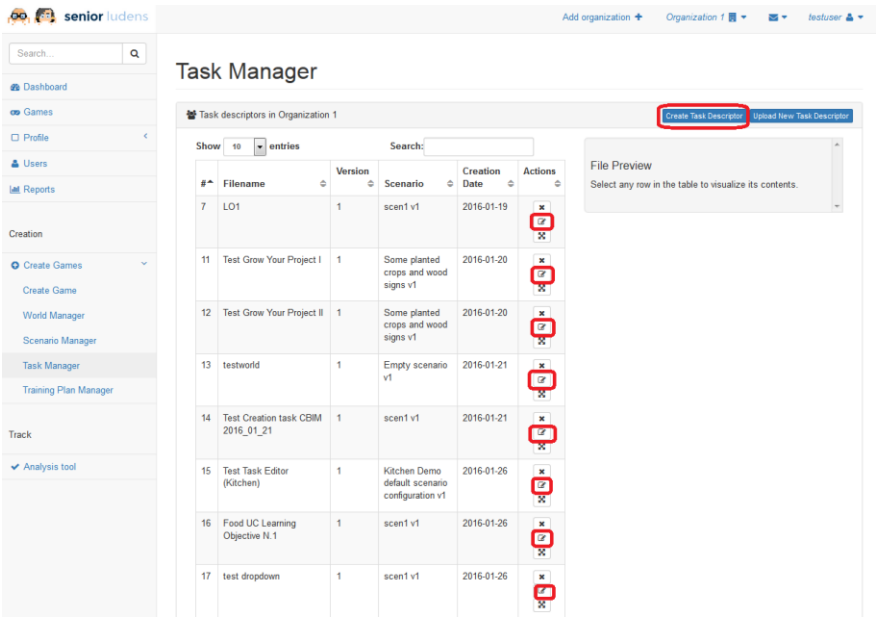

Figure 25 – Task Manager

When the creation option is selected, the system requests the user to choose an existing scenario descriptor in which the new task descriptor will be based. This step is only requested in the creation not in the edition.

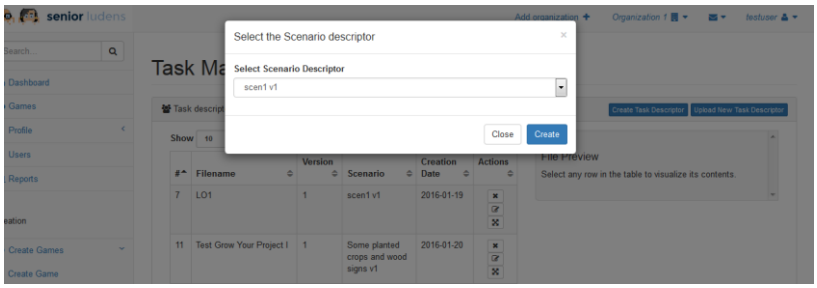

Figure 26 – Create new Task Descriptor

The last step is the visualization of the Task Editor embedded into the platform. Depending on the option chosen (edit or create), the editor is adapted and loads the needed descriptors from the SeniorLudens infrastructure.

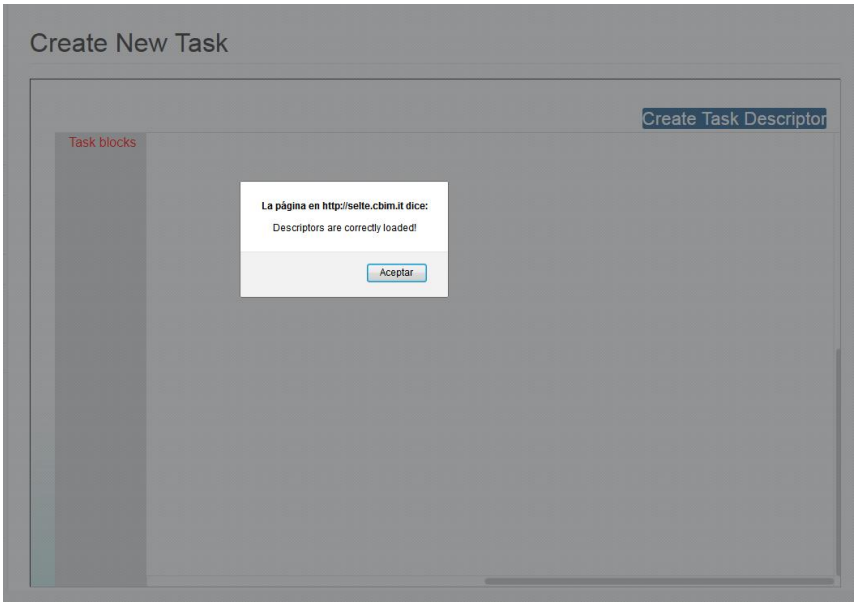

Figure 27 – Task Editor loaded

3.1.4- Features

3.1.4.1- How include new blocks

You can find the existing set of blocks in the toolbox (Task blocks) as follow:

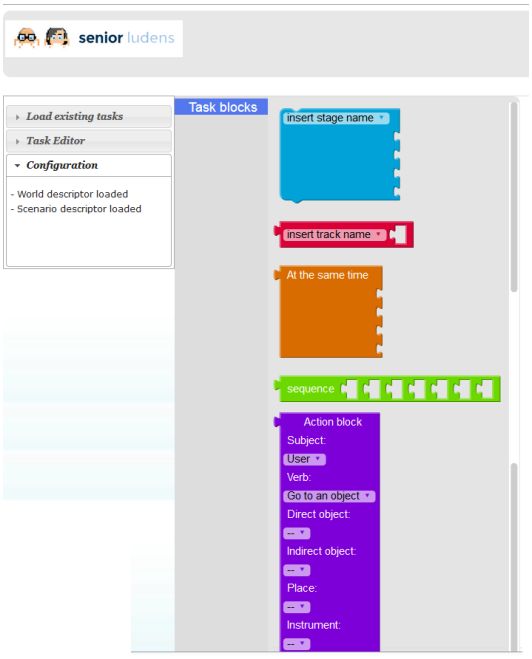

Figure 28 Toolbox which include complete set of blocks

The mandatory type blocks are as follow:

- Stage
- Track
- Sequence or At the same time
- Action

3.1.4.2- Workspace

Workspace is the section which include the blocks used to model the task

Figure 29 Task Editor working area

To inject a block into workspace is enough select the desired block from toolbox and drag it on workspace

3.1.4.3- Modify within the block

3.1.4.3.1- Duplicate

This feature provides to duplicate the workspace selected block.

3.1.4.3.2- Delete

This feature provides to remove the workspace selected block.

3.1.4.3.3- Run a contextual description of block

Example:
We can try to put into the workspace the Action Block and with right click of mouse on the block area, testing the functions as listed above:

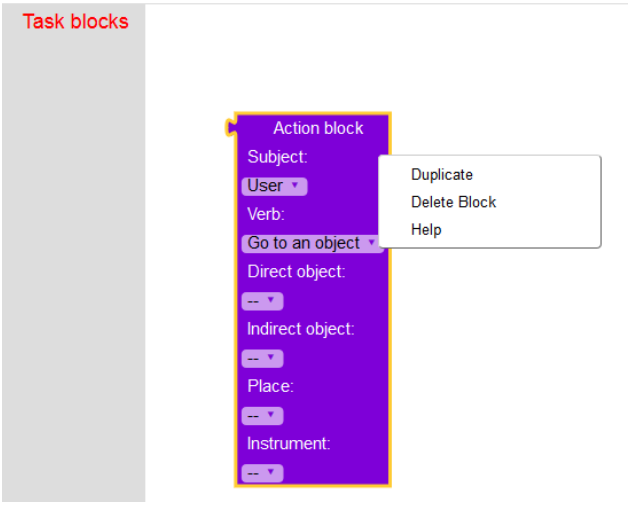

Figure 30 How modify an action block

Meanwhile, only for types Stage and Track, you can "rename" the title of the block.

For example, we can try to put into the workspace the Stage Block and with left click of mouse on the dropdown area, testing the function "New variable":

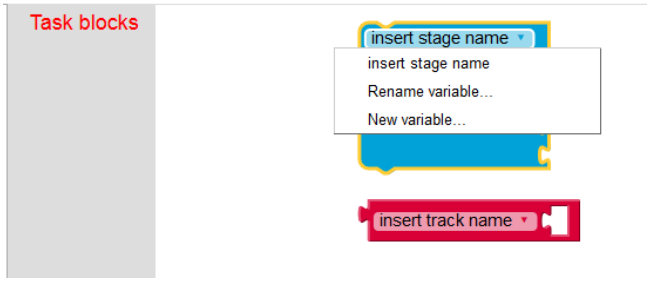

Figure 31 How modify stage/track block

3.1.4.4- Load existing task descriptor

You can load an existing task by clicking on "go to an object" within left menu (Load existing tasks) as follow:

Figure 32 How load existing task

3.1.4.5- Create new task descriptor

After inserting or modified an existing task, you can create and show new task descriptor simply clicking “Creating the new TE descriptor” within the left menu (task editor) as follow:

Figure 33 How create new TE descriptor

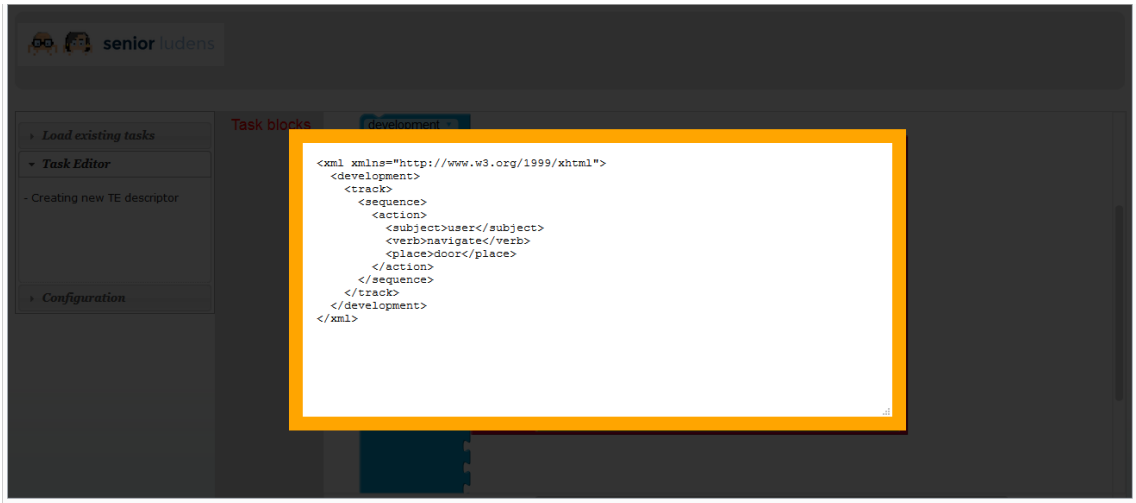

Figure 34 Visualization of the new TE descriptor XML

3.1.4.6- Put action modules in parallel

Task editor is able to manage the action block also in parallel to communicate to the Training Program Module how the action should be execute, at the same time or in sequence.

Example:

We can try to put a parallel block into a clean workspace and insert two actions block into this one as follow:

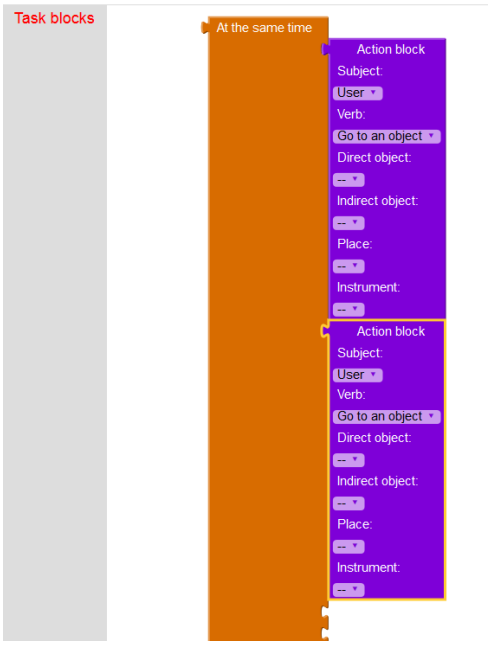

Figure 35 How manage the action blocks in parallel

4- Training Plan Editor

4.1.1- Introduction

This Editor is in charge of the training plan descriptor management. This fact means that it is placed in the last step of the descriptors structure. Because of this, training program descriptor is based on a set of task descriptors, and it is considered as the last descriptor needed to identify a Serious Game in SeniorLudens.

The Training Plan Editor has the objective of creating a list of Tasks configured with a determined level of difficulty. This goal is achieved by a process in which the user is able to configure Task with difficulty Level, so we can distinguish the following elements:

- *Task*: This element coincides with a Task descriptor included in the current organization. These elements are considered the basis of the training plan, as it is composed by an ordered set of them with a specific configuration.
- *Level*: This element is the specific configuration in which a Task can be arranged. This difficulty level is defined by the values of some predefined fields connected internaly with SeniorLudens Game Engine. These fields have been detailed in D2.2[4].

The level of difficulty of a Task is considered by the conjunction of a Level and by the specified number of repetitions needed to be completed by the player user.

Training Plan Editor has been fully developed, and partially integrated in the platform. It is still pending for the needed connections with SeniorLudens Storage Server, which will ensure the access to the created descriptors.

4.1.2- Access

This editor has been developed on top of the SeniorLudens infrastructure and it uses the same development stack. It has paved the way to the current integration level inside the SeniorLudens ecosystem. The editor can be accessed through the platform, using the users' role system and the security layer implemented as main base of the Platform. Based on this, it can be accessed through the platform url: http://demos-innovation-labs.com/sl/ .

Going into details of the access to the Training Plan Editor, it can be done through the following steps:

1. To access Training Program Editor, the users must validate their credentials using the platform login view. The editor is only accessed through the Management Portal, so the users must tick in the manager checkbox in the login form. To validate inside the Management Portal, the user must possess any of the management roles (See D1.1[5]).

[4] D2.2 – Scene, task and gamability editors.

[5] D1.1 – Users requirements

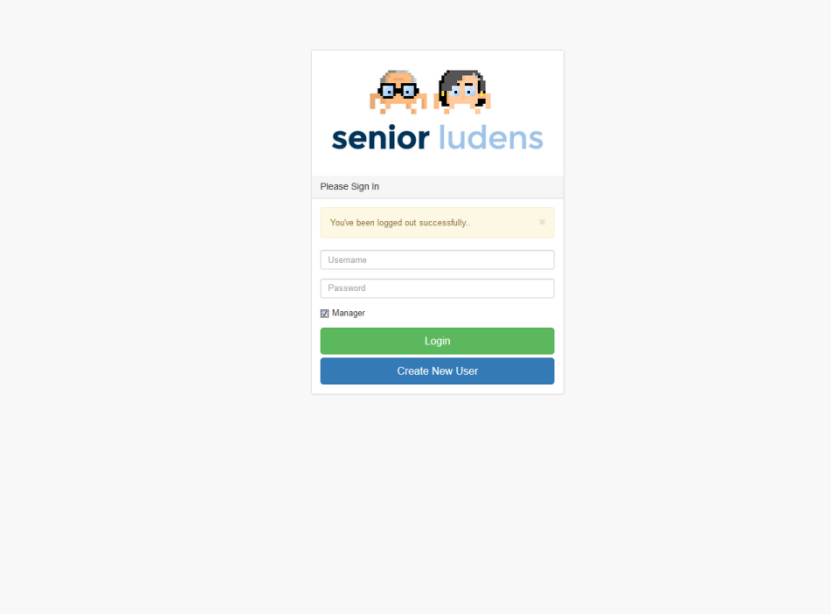

Figure 36 – Platform Login View

2. After the users validate successfully in the platform, they have access to the management dashboard. In consequence they have access to the different tools involved in the game definition in the selected organization. The users have to select Training Plan Manager in the side menu, located under Create Games parent menu. It is catalogued in this way, becauser this menu comprises all the tools used by managers for game creation in SeniorLudens.

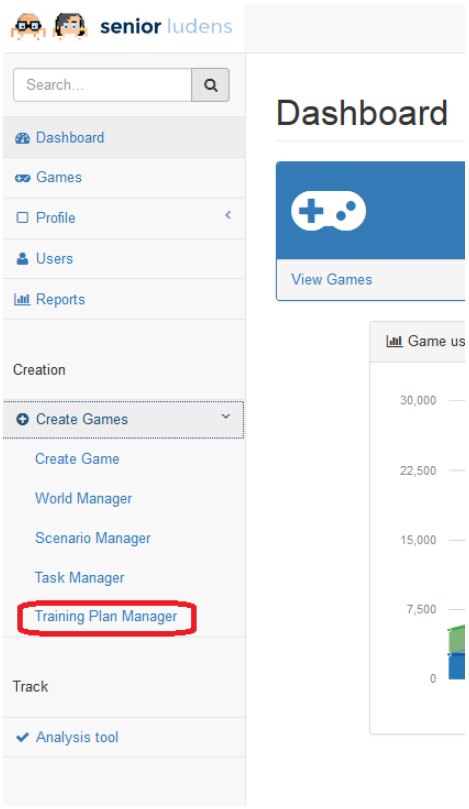

Figure 37 – Create Games Menu

3. Hence the Training Plan Manager is visualized in the platform. This manager is in charge of administering all the training plan descriptors in the selected organization. For this reason it shows a different row per each one of the existing training plan descriptors in the system. If the user desires to create a new training plan descriptor using the Training Plan Editor, they have to click on Create New button, which is located in the top bar of Training Plan Manager, as it is shown below.

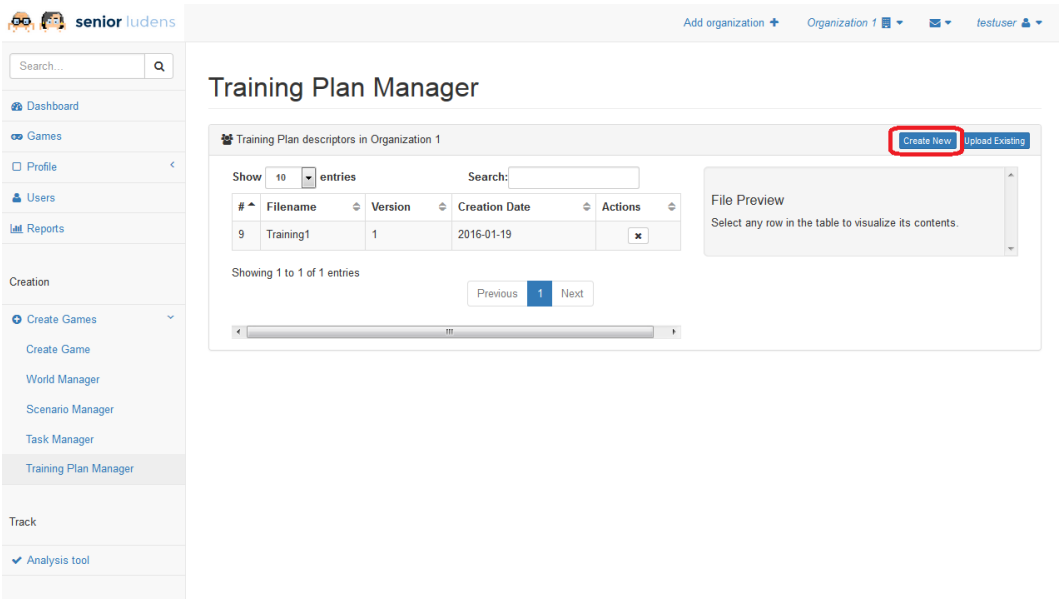

Figure 38 – Training Plan Manager

4. Once the user has selected this option, the platform loads Training Plan Editor and it is visualized to the user transparently to the user. In contrast with what was shown in Task Editor, the creation of a new Training Plan descriptor does not imply the selection of any task descriptor (because task descriptor is the lower level), because the Training Plan descriptor can use any Task descriptor previously uploaded into the organization to create the final descriptor. Because of this, the information is not needed and neither requested.

4.1.3- Features

Training Plan Editor pursues the objective of creating an ordered list of tasks with a specific difficulty configuration. This difficulty is defined as the union of the number of repetitions and the specific Level defined in the Editor. The users can create so many Levels with different configurations as they desire. There is a default Level, called Level 0, which defines the difficulty base for any task in the system. This default level cannot be deleted or modified.

It is important to clarify, that Training Plan Editor is fed from the Task available in the current organization, so they are loaded asynchronously when the editor is loaded. Only descriptors in the current organization are shown, and are accessible to be used in the editor. In the same way, the resulted Training Plan descriptor will be stored in the current organization in which it is being created.

The main operation available in Training Plan Editor consists on dragging and dropping the Task Elements into the Main Canvas. By this simply action the Tasks are included inside the Training Plan descriptor model.

It is possible to include a same task as many times as the user desires. All of these replicas will be considered to have an individual configuration.

In the current integration only creating has been deployed, but will be adapted to include the editing mechanisms together with the fully integration with SeniorLudens System.

4.1.3.1- Main View

The editor is composed by a set of individual elements with different function in the Training Plan definition. The appearance of the view is divided into three columns. The left column is reserved for the input elements, which are considered as the source elements in the editor. The central column is the canvas in which the list of configured tasks is created. The right side is reserved to show the details of the different elements and also to their internal management.

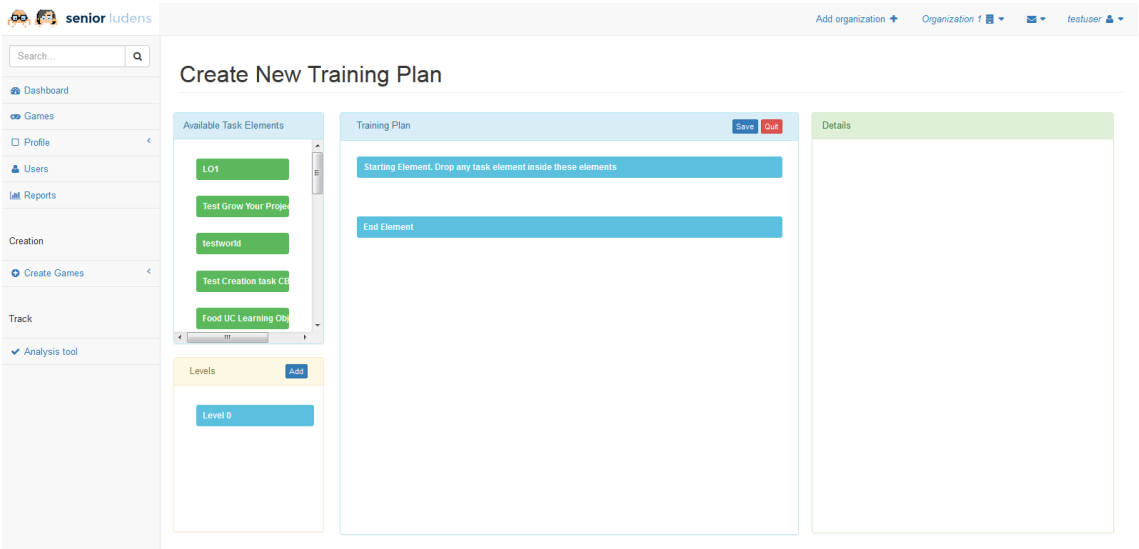

Figure 39 – Training Plan Editor

We are going to make a review on the different components that compose the editor, defining the different features of each of them.

4.1.3.2- Source elements

The source elements are defined by the union of the two different actors involved in the training plan definition: Levels and Tasks. These both elements are the basic bricks in which the descriptor is based.

4.1.3.2.1- Task

This component is located in the upper left side of the editor. It is responsible to load asynchronously the Task descriptors available in the current organization. These descriptors are represented as individual rows in the component. These rows are intended to be dragged into the main canvas to compose the descriptor structure.

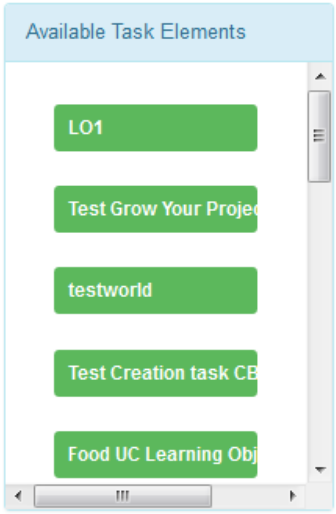

Figure 40 – Task Elements

4.1.3.2.2- Level

This component is located under the Task component. It is in charge of the management of the different Levels defined in the current training plan. In the same way the Task component does, it represents the different levels as individual rows, but they are not draggable because they are considered simply configuration modules for the tasks included inside the training plan.

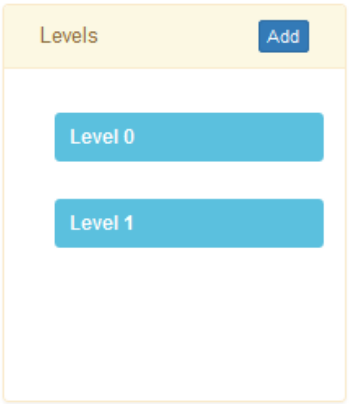

Figure 41 – Level Elements

The component also includes the Add Level action. It is located on top of the component and it permits to create new customized levels. The creation is performed through the Details View. After the creation in Details View, the new Level is represented as another row in Level Component.

The component also permits to visualize the configuration of the level by examining the value of the different fields which compose it. This information is shown in the Details View, which includes options to modify and update the individual fields.

The component includes a default Level called Level 0. This level is considered the default level of difficulty for all tasks from the moment in which a new Task is included inside the canvas. The default level cannot be modified.

4.1.3.3- Training Plan Main Canvas

This component represents the main work space in the editor, as it is the place in which the tasks are dropped and included inside the training plan descriptor under creation.

When a Task is dropped, the element is transformed to a configurable Task. It means that it is appended the number of repetitions and the level of difficulty. By default the number of repetitions is set to 1 and the default level of difficulty.

Once the Task has been included it can be reorder in the canvas by simply dragging and dropping to the new location. It will adjust the other tasks accordingly.

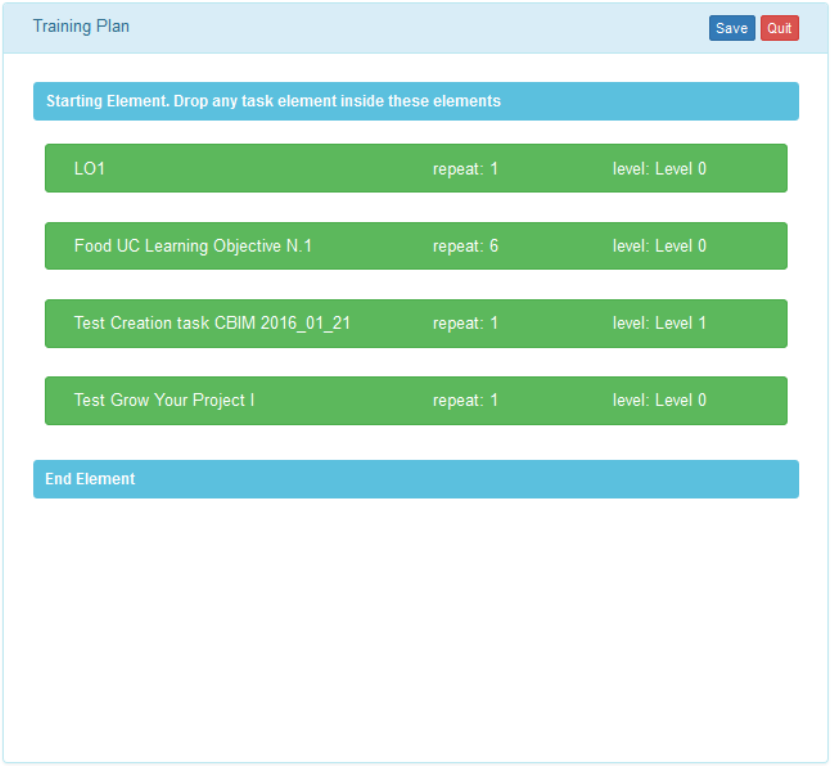

Figure 42 – Training Plan Main Canvas

A Task can be replicated as many times as the user desires with same or different configuration. It means that the user has full control over the training plan definition.

The user can modify the configuration of each individual task included by just clicking on it. It will display the information in the detail view. This information is also updatable from this view. If any change is performed, it will be updated in the canvas immediately.

The canvas also visualizes the general actions for the editor in the top bar. These general actions are save and quit. For the moment the connection with Storage Server has not been fully integrated so the generated descriptors are not visualized in the Training Plan Manager.

4.1.3.4- Detail view

This component is in charge of visualizing all the information contained by the different elements that compose the training plan descriptor. This information is divided between the two source elements available in the editor: Task and Levels.

This component also provides the update mechanisms on these elements. After any change it is forwarded to the origin component responsible for the specific element.

4.1.3.4.1- Task Details

The task details are shown when the user selects any task included inside the Main Canvas. In the view is displayed all the configurable fields of the task: number of repetitions and the specific level of difficulty. Only the levels defined in the current training plan descriptor are selectable for the definition of the task difficulty.

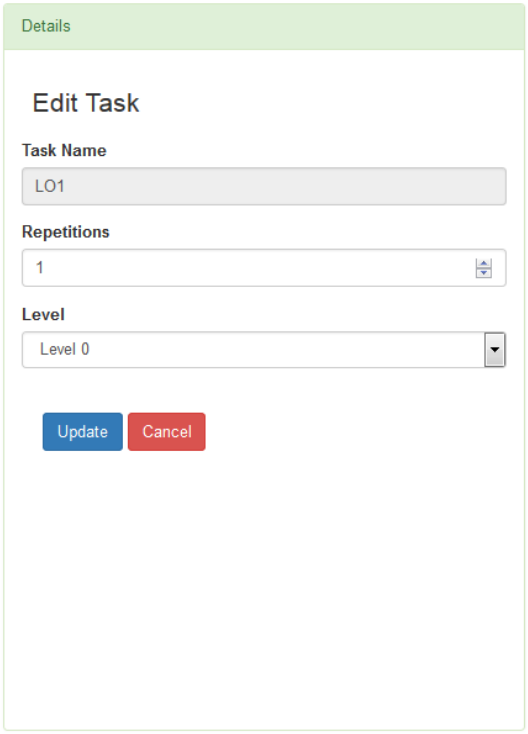

Figure 43 – Task Details

The view permits to update the task configuration. After any change, it is forwarded to the Main canvas which updates the information of the modified task. If the task is replicated in the training plan, only the edited task is modified.

4.1.3.4.2- Level Details

The level details are visualized when the user selects any Level in Level Component. It displays a tabbed view in which all the fields which define the Level are shown (these fields are detailed in D2.2[6]). These fields are organized in three categories:

- Metadata: It represents the general information of the level. It is important to highlight that in case of editing the Level, the id and the name are not modifiable to guarantee the consistency of the descriptor information. In case of creation, the id is modifiable neither, to avoid collisions in the internal data model.
- Factors: These fields are multipliers connected with internal fields defined in SeniorLudens Game Engine.
- Helpers: These helpers are intended to modify the behaviour of the game depending of the needed level of assistance requested for the users.

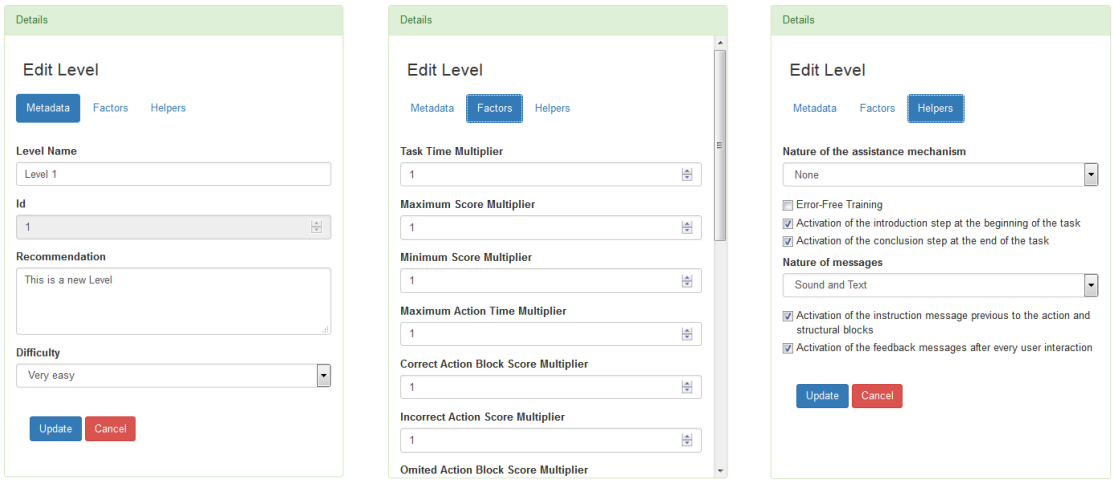

Figure 44 – Level Details

These fields shape the difficulty of the level. In case of the default level, it is not possible to modify the fields, as it is considered the base level in the editor.

[6] D2.2 Scene, task and gamability editors

Figures and tables

Acronyms

Acronym	Explanation
SL	SeniorLudens
SLGK	SeniorLudens Game Kit
SDK	Software Development Kit
TE	Task Editor