EngAGe: An Engine for Assessment in Games

Tutorial developers

EngAGe is a tool aimed at developers and teachers for the integration of assessment in educational games. It is composed of:

- A grammar for a configuration file describing the game and its assessment
- A set of web services to perform the assessment
- A database where the data from the various gameplays is logged
- A web interface for managing the games created and visualising the learning analytics

EngAGe allows for a separation of the assessment from the game mechanics, therefore creating a modularity that offer the possibility, for educators, to alter the game's assessment even after its distribution. This is done through an editor in the web interface.

This tutorial is aimed at developers that are interested by the concept and would want to see how the engine is integrated in practice within a Unity 2D game with C# scripting.

Contents

Tutorial – Part 1

0. Set up the environment

0.1 Unity

If you don't have Unity installed in your machine, install it from https://unity3d.com/get-unity this tutorial has been tested with Unity 4.6 and Unity 5.

0.2 The game code

You can download the game code or clone the GIT repository of the project https://github.com/yaelleUWS/eu_game/tree/without_engage.

0.3 EngAGe username and password

If you have been given developer's credential with this tutorial skip to the next step.

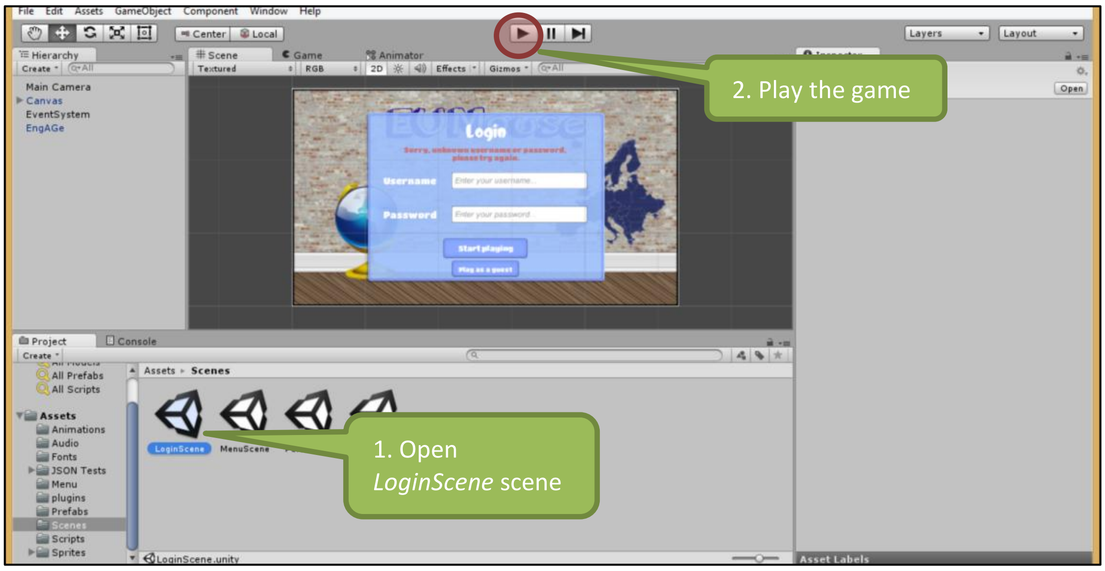
If you don't have a developer id, username and password, email me or use the default developer bellow. Be aware that, if you do so, anyone could have access to your work, update or delete it.

Id = ***404*** **-** Username = ***default*** **-** Password = ***engage***

1. Play the game

Start Unity and open the *EU mouse* project.

In the *Project* window, go to *Assets* > *Scenes* and open *LoginScene* with a double click.
Then play the game using the play button.



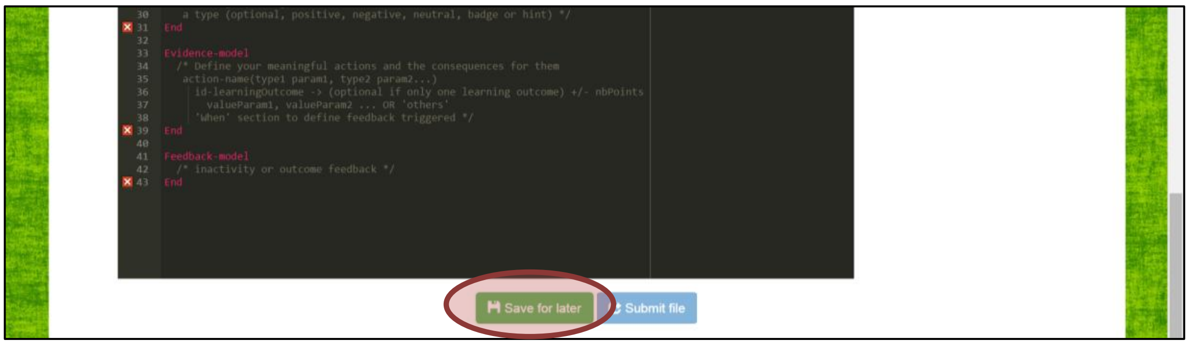
Get acquainted with the game's mechanics, there is no assessment in the game for now. No score or feedback is showed. You will add the assessment with EngAGe, as detailed in the next sections.

2. Create the configuration file

As a developer, the first thing that you have to do to use EngAGe in your game is to create its *configuration file*. This file has to be written using a pre-defined grammar and submitted via the engine's website. It defines the game's assessment in seven sections explained bellow. More information can be found on them here: http://146.191.107.189/documentation/doc.

Open http://146.191.107.189/serious_games/new, on this page you will find the configuration file editor. Use the online editor to write your configuration file, it will allow for syntax highlighting and will notify you if it encounters any error.

Tip - You can use the "Save for later" button at any time to save your work.



```
    a type (optional, positive, negative, neutral, badge or hint) */
End

Evidence-model
  /* Define your meaningful actions and the consequences for them
    action-name(type1 param1, type2 param2...)
      id-learningOutcome -> (optional if only one learning outcome) +/- nbPoints
        valueParam1, valueParam2 ... OR 'others'
      'When' section to define feedback triggered */
End

Feedback-model
  /* inactivity or outcome feedback */
End
```

2.1. *The game description*

The first section of the configuration describe your educational game. Here is an example:

```
Serious-game
  /* Compulsory */
  SGname: "My Serious Game"
  SGdeveloper: 1234

  /* Optional */
  SGdesc: "A small description of my serious game"
  SGageRange: 0-99
  SGlanguage: EN
  SGcountry: UK
  SGgenre: "platform"
  SGsubject: "Mathematics, Vectors"
  SGpublic: true
End
```

Update the example to describe the EU mouse game. Most of the parameters are optional here and you don't have to include them, but you do have to give your game a name (SGname) and specify your developer ID (SDdeveloper).

For more information about this section see: http://146.191.107.189/documentation/doc#SG.

2.2. *The player's data*

The second section lists the data you need from your players. This is particularly important when using learning analytics, you will want to refine your data. The section looks like:

```
Player
  name String "What’s your name?"
  gender Char "Are you a boy (b) or a girl (g)?"
End
```

Update the example to ask for the age (Int), gender (Char) and country (String) of the player.

For more information about this section see http://146.191.107.189/documentation/doc#Player.

2.3. The learning outcomes and scores

Here you will list the learning outcomes of your game, and its scores. Each item need to have a name (identifier), a description (string) and can have an optional starting value (integer, default value is 0). For example:

```
Learning-outcomes
  myOutcome1 "A description of my outcome1"
  myOutcome2 "A description of my outcome2" 10
End
```

Update the following example to list EU Mouse three scores:

1) eu_countries: the number of EU countries left to find, starts at 28.
2) eu_score: the number of EU countries found by the player, starts at 0.
3) lives: the number of lives left to the player, starts at 3.

For more information about this section see http://146.191.107.189/documentation/doc#LO.

2.4. The feedback

Here, you will list the feedback triggered by the game. A piece of feedback has a name (identifier) and a description (string), it can also have a type. The types available are: positive, negative, badge, hint, final, win and lose.
Note that you can use square brackets for accessing parameters that will be sent to the server. In our case there will only be one: *country*.
Update the following example to list EU Mouse feedback:

1) correctEU : that says that the country in indeed part of EU (you can use [country] to access the country selected by the player).
2) wrongEU: when the country selected is incorrect.
3) endWin: feedback that triggers the end of the game and the player wins.
4) endLose: feedback that is triggered when the player loses.
5) slowGame: triggered when the player is in difficulty
6) speedGame: triggered when the player is doing very good

```
Feedback-messages
  increaseDifficulty "You are good, let’s make it more challenging!" adaptation
  myFeedback1 "Well done, correct answer!" positive
  myFeedback2 "Not quite! [param1] did not go with [param2]. Try again!" negative
  youWin "Congratulations! You won!" win
  myBadge "Congratulations! You earn the level 1 badge!" badge
End
```

The game also has 8 badges, six of them are defined bellow.

```
Feedback-messages
  /* … feedback defined earlier */
  gold_medal "Well found 200 EU countries" badge
  silver_medal "You found 100 EU countries" badge
  expert_time "You played more than 60 minutes" badge
  master_time "You played more than 30 minutes" badge
  effort "You played 5+ times" badge
  performance "You won 10+ times" badge
End
```

Update the section adding the following two:

1) bronze_medal: when the player found 50 EU countries
2) novice_time: when the player spent more than 10 minutes practicing

For more information about this section see http://146.191.107.189/documentation/doc#Fdbck

2.5. The assessment logic

This section defines how the scores are updated and specifies the actions available to the player. Each action has a name, a list of parameters, and a description. After what, you need to list the consequences that the player choices have in terms of scores update. In the "When" part, feedback triggers are defined, feedback can be immediate or delayed (shown only in logs).

The EU mouse game is rather simple, there are only two actions that a player can do:

1) ***Select a country for the first time***: if the country is correct, 1 is substracted to eu_countries and 1 is added to eu_score. If the country is incorrect 1 is substracted from lives
2) ***Select a country that has been selected before***: if the country is correct, 1 is added to eu_score, otherwise, 1 is substracted from lives.

The following section describes the first action, update it (in the editor) to add the second one.

```
Evidence-model
  newCountrySelected (String country)
  "When a player selects a country for the first time"
    /* scores to be updated */
    eu_countries -> -1, eu_score -> 1
        /* list of choices */
        austria
        belgium
        bulgaria
        croatia
        cyprus
        czech_republic
        denmark
        estonia
        finland
        france
        germany
        greece
        hungary
        ireland
        italy
        latvia
        lithuania
        luxembourg
        malta
        netherlands
        poland
        portugal
        romania
        slovakia
        slovenia
        spain
        sweden
        united_kingdom
    End
    /* if the country selected was not in previous list (others), on life is lost */
    lives -> -1
      others
    End
    When
      /* for any positive point obtained, correctEU feedback is triggered */
      any(+) : correctEU
      /* if the country was incorrect, wrongEU feedback is triggered */
      others : wrongEU
    End
  End
End
```

For more information see http://146.191.107.189/documentation/doc#Actions.

2.6. The feedback model

This section defines feedback triggered by a score reaching a limit. The following feedback model includes the winning and speeding condition of the game. Add the following:

- When *lives* reaches 0 (inferior to 1) the game is lost (feedback endLose).
- When lives reaches 1 (inferior to 2) the game slows down (feedback slowGame)

```
Feedback-model
  eu_countries <  1 : endWin
  eu_countries < 6 : speedGame
End
```

For more information see http://146.191.107.189/documentation/doc#FdbckModel.

2.7. The across-gameplays feedback model

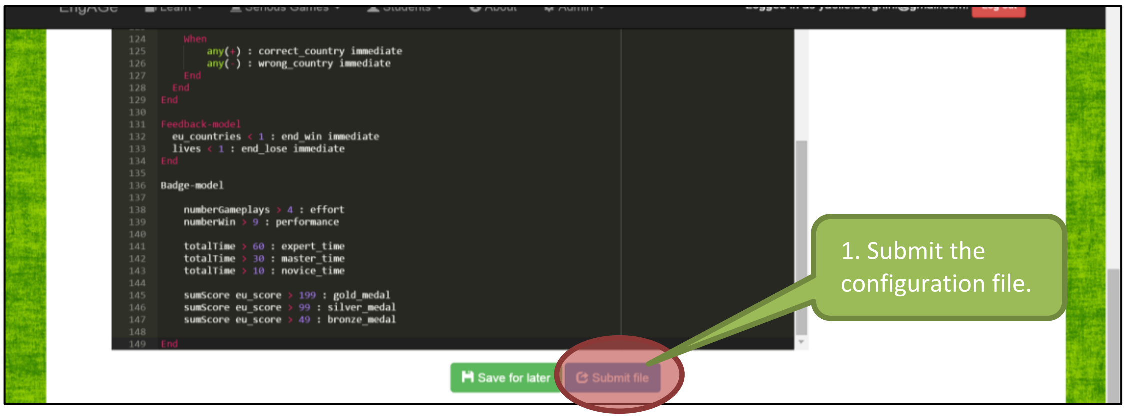
This section describes how the badges are awarded to the player, there are various function that can be used for that. Simple functions include: *'numberGameplays'*, *'numberWin'*, *'totalTime'* and *'averageTime'*. Other functions are associated to a score, they are: *'sumScore'*, *'averageScore'*, *'maxScore'*, *'minScore'*. Update the following section to describe all 8 badges of EU mouse.

```
Badge-model
  numberGameplays >  9 : effort
  numberWin >  9 : performance
  totalTime >  10 : novice_time
  sumScore eu_score >  49 : bronze_medal
End
```

For more information see http://146.191.107.189/documentation/doc#BadgeModel.

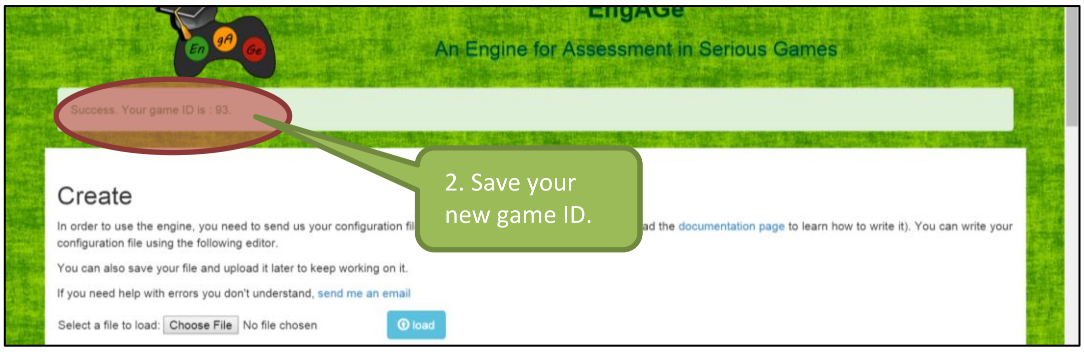
Congratulations!

You now have completed your configuration file. You can submit it. If the file is correct, you should receive an ID back, take a note of it, you will use it soon.



```
      When
          any(+) : correct_country immediate
          any(-) : wrong_country immediate
      End
    End
  End

  Feedback-model
    eu_countries < 1 : end_win immediate
    lives < 1 : end_lose immediate
  End

  Badge-model

      numberGameplays > 4 : effort
      numberWin > 9 : performance

      totalTime > 60 : expert_time
      totalTime > 30 : master_time
      totalTime > 10 : novice_time

      sumScore eu_score > 199 : gold_medal
      sumScore eu_score > 99 : silver_medal
      sumScore eu_score > 49 : bronze_medal

  End
```

Save for later
Submit file
1. Submit the configuration file.



An Engine for Assessment in Serious Games
2. Save your new game ID.

Success. Your game ID is : 93.

Create

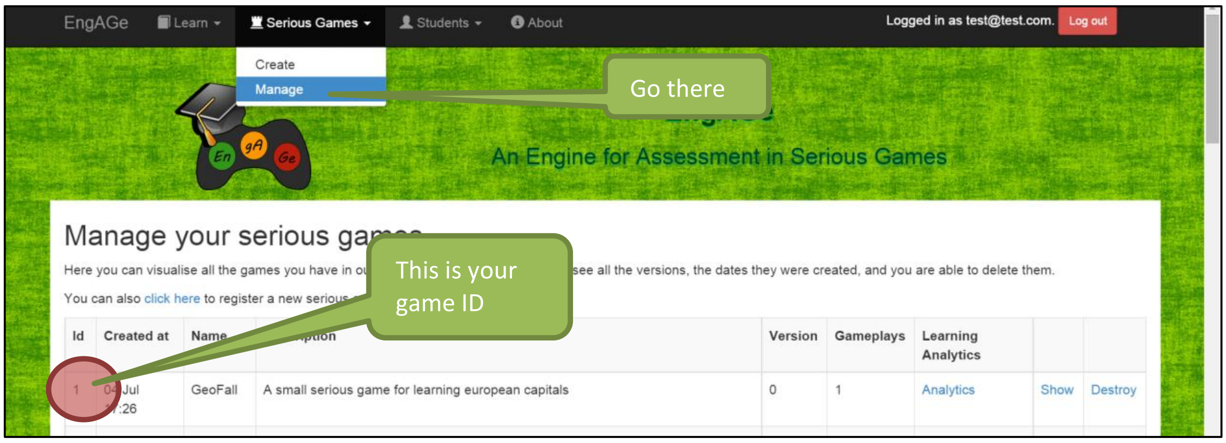
In order to use the engine, you need to send us your configuration fil ad the documentation page to learn how to write it). You can write your configuration file using the following editor.

You can also save your file and upload it later to keep working on it.

If you need help with errors you don't understand, send me an email

Select a file to load: Choose File No file chosen load

If you forgot your game ID, you can find it by logging into the interface with your username and password and going into *Serious Games > Manage*.



Manage your serious ga

Here you can visualise all the games you have in o see all the versions, the dates they were created, and you are able to delete them.

You can also click here to register a new serious

Id	Created at	Name	ption	Version	Gameplays	Learning Analytics		
1	04 Jul :26	GeoFall	A small serious game for learning european capitals	0	1	Analytics	Show	Destroy

If you are having trouble with the configuration file, you can cheat and have a look at the file situated at the root of the Git folder.

https://github.com/yaelleUWS/eu_game/blob/without_engage/configFile_engage.txt

Tutorial – Part 2

3. Integrate the assessment

EngAGe offers a set of web services to perform the assessment in your game based on how you defined it in the configuration file. Using them you will be able to:

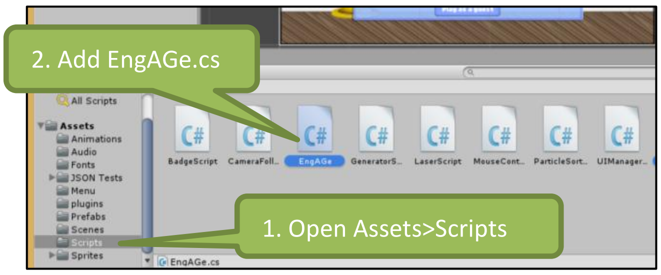
- Access the game information
- Log a player
- Start a gameplay
- Assess an action
- Retrieve feedback and scores associated to a gameplay
- Adapt the game
- End a gameplay
- Retrieve the badges earned by a player
- Retrieve the game leader board
- Retrieve learning analytics information about a game

So, now that EngAGe knows about your game, let's use these web services to update EU mouse and create some assessment.

3.0 EngAGe C# script for Unity

Luckily for us, using Unity, you don't need to code the calls to the web services yourself, you can use a script file that will do it for you. Download the *EngAGe.cs* file from:
http://146.191.107.189/documentation/downloads.

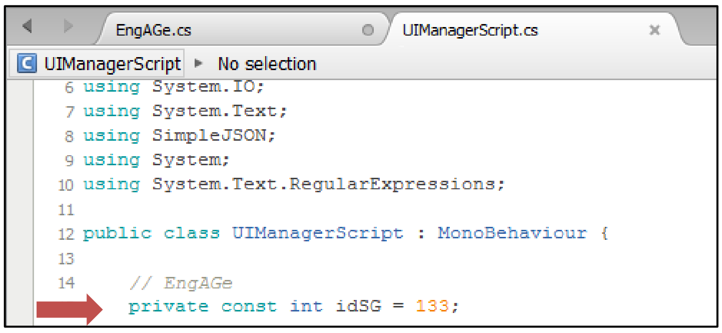
1. Add the script to the Unity project, drag and drop it in the *Project* window, in *Assets > Scripts*.



2. Attach the script (drag and drop) to the Event System Game object ***In all four scenes***

Add your game ID

As you will need your game ID when you communicate with the engine, you should save it as a private constant in UI manager script.
In *UIManagerScript.cs, a*dd a following line: `private const int idSG = XXX;` to the class. Obviously, replace XXX by your own game ID.



```
using System.IO;
using System.Text;
using SimpleJSON;
using System;
using System.Text.RegularExpressions;

public class UIManagerScript : MonoBehaviour {

    // EngAGe
    private const int idSG = 133;
```

3.1 Use EngAGe functions - LoginScene

The first thing a player does when he/she start the game is to log in – or play as a guest. We want to ask EngAGe if the player credentials exists, and if he/she hasn't played before, we need to ask a few questions (age, gender and country based on the configuration file).

For this task you will use the ***SGaccess*** web service.

URL	`http://146.191.107.189:8080/SGaccess`
Method	`POST`

Consumes	JSON with three key/value
	`{`
	`        "idSG": 92,`
	`        "username": "test",`
	`        "password": "password"`
	`}`

Produces	JSON with:
	1. loginSuccess: username/password exists,
	2. params: list of information to ask about the player, if any,
	3. version: version of the game to be played by the player
	4. idPlayer: if the player has already played before
	5. student: basic info about the student logged in
	
	Example of a correct login with a student that had never played before:
	`{   "student": {`
	`        "id": 1,`
	`        "username": "test",`
	`        "idSchool": 1,`
	`        "dateBirth": "2001-01-01"`
	`    },`
	`    "loginSuccess": true,`
	`    "params": [`
	`        {   "name": "age",`
	`            "question": "How old are you",`
	`            "type": "Int"`
	`        },`
	`        {   "name": "gender",`
	`            "question": "Are you a boy (b) or a girl (g)",`
	`            "type": "Char"`
	`        }`
	`    ],`
	`    "version": 0`
	`}`

For more information visit: http://146.191.107.189/documentation/doc#GP.

But we don't have to call the web service ourselves as *EngAGe.cs* does it for us; we only need to call the function "*loginStudent*" or "*guestLogin*" using the EngAGe.E singleton. The function contains asynchronous calls, so it needs to be started in a coroutine

(http://docs.unity3d.com/Manual/Coroutines.html)

Guest login

The player might want to play as a guest, in which case we need to check that the game is public and if so ask the guest for more information (age, gender and country again).

1. In *UIManagerScript.cs,* find the "*GetStartedGuest*" function. It's the function called after a click on the "*Play as a guest*" button. For now the function only loads *ParametersScene*.
2. Replace the Application.LoadLevel line by the following line:

```
StartCoroutine(EngAGe.E.guestLogin(idSG, "LoginScene", "ParametersScene"));
```

This will check if the game is public, and load either:

a. *LoginScene*, if the game is not public. In that case, EngAGe saves the error (202).
b. *ParametersScene*, if the access is granted, you will want to ask the player questions.

User Login

If the player provides a username and password

1. In *UIManagerScript.cs,* find the "*GetStarted*" function. It's the function that is called after a click on the "*Start playing*" button. For now the function saves the username and password and loads the *ParametersScene*.
2. Replace the **`Application.LoadLevel`** line by the following:

```
StartCoroutine(EngAGe.E.loginStudent(idSG, username, password, "LoginScene",
"MenuScene", "ParametersScene"));
```

This will try to log the player, and load either:

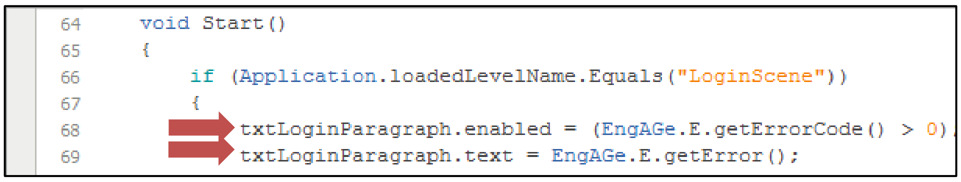
a. *LoginScene*, if the login failed (wrong username or password). In that case, EngAGe saves the error (201 = login failed, 0 = no error) and an error message.
b. *MenuScene*, if the login was successful and that the player already played the game (therefore already gave his/her information, no need to ask again).
c. *ParametersScene*, if the login was successful but the player is new to the game, you will want to ask him/her the questions.

Error Handling

If you try running the game now, everything should work as described. You can try to log with a default student ("test", "test1234"), you should be taken to the parameters screen. A guest login also works if you defined your game as public in the configuration file.
But, if you try to log in with incorrect username and password, although you are taken back to the Login scene, the game doesn't say anything to guide you. You need to add an error message.

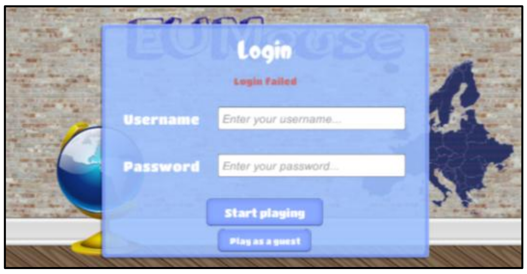
1. In *UIManagerScript.cs,* go to the "*Start*" function. For now, if the current scene is *LoginScene*, the text "*txtLoginParagraph*" that corresponds to the error paragraph is systematically disabled. Change the line to enable it if engage's error code is superior to 0. Also add another line to set the text content to engage's error message, this will be useful later. Here is the code:

```
txtLoginParagraph.enabled = (EngAGe.E.getErrorCode() > 0);
txtLoginParagraph.text = EngAGe.E.getError();
```



```
void Start()
{
    if (Application.loadedLevelName.Equals("LoginScene"))
    {
        txtLoginParagraph.enabled = (EngAGe.E.getErrorCode() > 0)
        txtLoginParagraph.text = EngAGe.E.getError();
```

Try again with incorrect credentials, you will see a red text appear like so:



1.2 Use EngAGe functions – ParametersScene

For now, the parameters scene doesn't ask for anything. We need to make it create one text field per information needed from the player. Of course, we could use Unity to create 3 text inputs for the age, gender and country. But, in case we want to add, delete or modify these player's characteristics in the future, we'll write some more generic code.

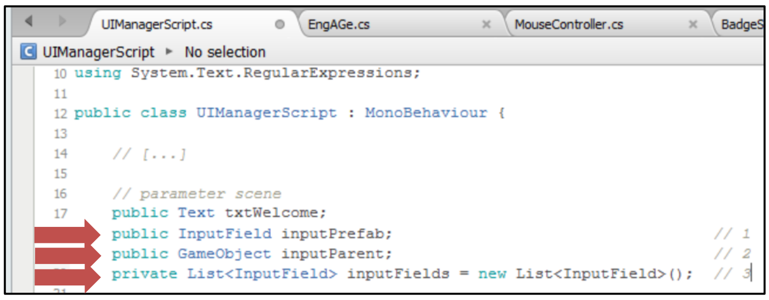
Adding variables for the scene

For this section, we will need to add three variables to the *UIManagerScript.cs*:

1. An example of an input field (one input will be created for each player's characteristic).
2. A game object that will be the inputs' parent element.
3. A private list of all the inputs that have been created so we can have easy access to them.

Here is the code:

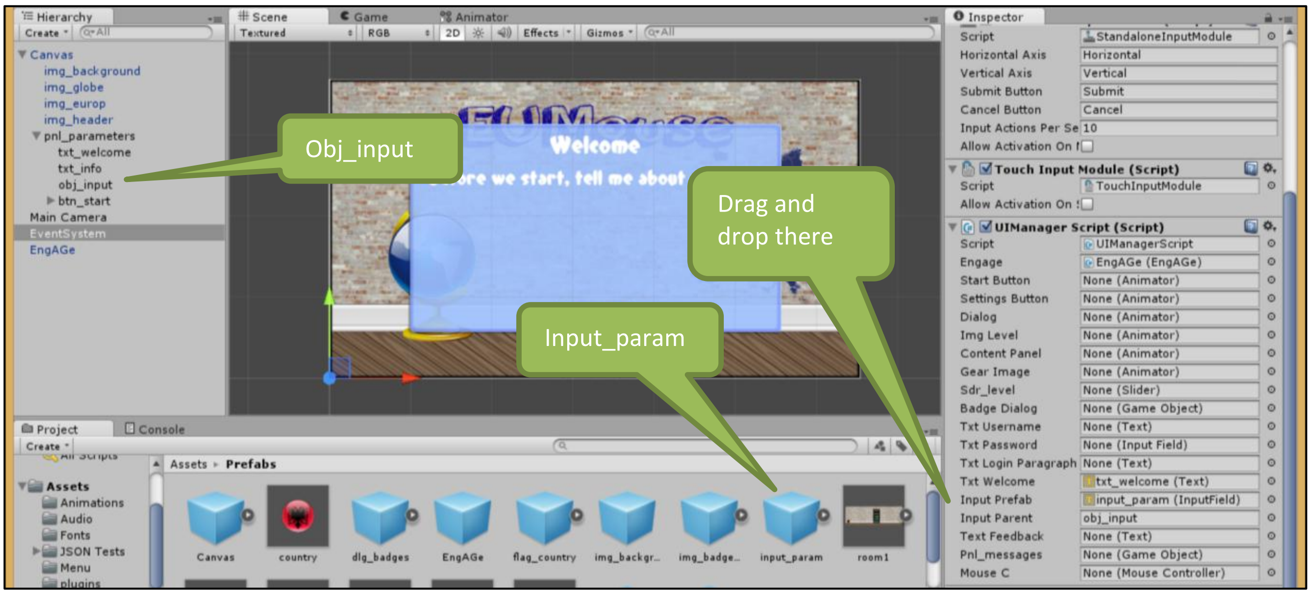
```
public InputField inputPrefab;
public GameObject inputParent;
private List<InputField> inputFields = new List<InputField>();
```



```
using System.Text.RegularExpressions;

public class UIManagerScript : MonoBehaviour {

    // [...]

    // parameter scene
    public Text txtWelcome;
    public InputField inputPrefab;                                    // 1
    public GameObject inputParent;                                    // 2
    private List<InputField> inputFields = new List<InputField>();   // 3
```

We now need to attach actual game objects to the public variables.

1. Open the *ParametersScene* scene.
2. Select *EventSystem* and drag and drop
   a. The prefab (*Assets > Prefabs*) *input_param* into *Input Prefab*
   b. The game object *obj_input* (Canvas > pnl_parameters) into *Input Parent*



Script	StandaloneInputModule
Horizontal Axis	Horizontal
Vertical Axis	Vertical
Submit Button	Submit
Cancel Button	Cancel
Input Actions Per Se	10
Allow Activation On I	☐
Touch Input Module (Script)	
Script	TouchInputModule
Allow Activation On S	☐
UIManager Script (Script)	
Script	UIManagerScript
Engage	EngAGe (EngAGe)
Start Button	None (Animator)
Settings Button	None (Animator)
Dialog	None (Animator)
Img Level	None (Animator)
Content Panel	None (Animator)
Gear Image	None (Animator)
Sdr_level	None (Slider)
Badge Dialog	None (Game Object)
Txt Username	None (Text)
Txt Password	None (Input Field)
Txt Login Paragraph	None (Text)
Txt Welcome	txt_welcome (Text)
Input Prefab	input_param (InputField)
Input Parent	obj_input
Text Feedback	None (Text)
Pnl_messages	None (Game Object)
Mouse C	None (Mouse Controller)

3. Save the scene (Ctrl – S or File > Save scene).

Display information needed as text fields

When the scene loads, we need to create one input for each characteristic, to do so:

1. In *UIManagerScript.cs* "*Start*" function find the section called in *ParametersScene* is loaded.
2. Update it with the following code. It goes through all the characteristics EngAGe received earlier (retrieved with engage.getParameters()) and for each one:
    a. A text field is created based on the *input_param* prefab
    b. Its text is set to prompt the player to input its data in the correct format (more code would be needed to check these fields but won't be covered in this tutorial).
    c. The fields are positioned, aligned vertically (again, this is a quick fix that works for 1-5 characteristics, for more, a little bit of thought should be put into the positioning).
    d. Each textfield is stored into the *inputFields* array for easy access later.

```
else if (Application.loadedLevelName.Equals("ParametersScene")) {
  txtWelcome.text = "Welcome " + username ;
  int i = 0;
  // loop on all the player's characteristics needed
  foreach (JSONNode param in engage.getParameters())
  {
    // creates a text field in the panel parameters of the scene
    InputField inputParam = (InputField)Instantiate(inputPrefab);
    inputParam.name = "input_" + param["name"];
    inputParam.transform.SetParent(inputParent.transform);
    inputParam.text = param["question"] + " (" + param["type"] + ")";

    // position them, aligned vertically
    RectTransform transform = inputParam.transform as RectTransform;
    transform.anchoredPosition = new Vector2(0, 20 - i*50 );

    // save the input in the input array
    inputFields.Add(inputParam);
    i++;
  }
}
```

Save the player's answer

When the player clicks on the Start button, the value of the text fields should be sent to EngAGe.

1. In *UIManagerScript.cs* "*GoToMenu*" function. It is called after a click on the start button.
2. All it does is load the menu scene, update the code as follows to save the inputs value.

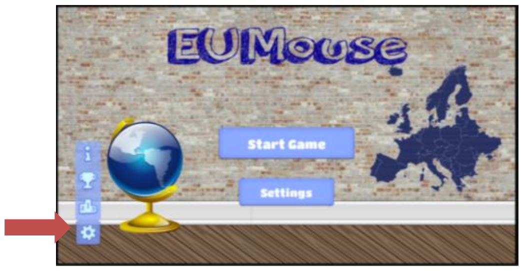
```
public void GoToMenu() {
        // for each parameter required
        foreach (JSONNode param in engage.getParameters())
        {
                // find the corresponding input field
                foreach (InputField inputField in inputFields)
                {
                        if (inputField.name == "input_" + param["name"])
                        {
                                // and store the value in the JSON
                                param.Add("value", inputField.text);
                        }
                }
        }
        Application.LoadLevel("MenuScene");
}
```

You can now test this scene. If you load the parameters scene directly, you will receive an error message, indeed some variables haven't been initialised, you will need to play the login scene first (open *LoginScene* and click the play button).

Tutorial – Part 3

3.3 Use EngAGe functions – MenuScene

If you don't have much time, you can skip the 3 first sections and go straight to Start a gameplay .
Play the menu scene (open it and click the play button), notice in the bottom left corner, a gear button. If you click on it, you can see a menu with three icons.



1. The information icon opens a window displaying a description of the game.
2. The trophy icon opens a window displaying the badges gained by the player.
3. The podium icon opens a window displaying the leader board for the game

The next three sections will populate each of these windows.

Access to the game information

For now, the information window doesn't display much. We need to ask EngAGe to retrieve the game data and display it there.

For this task you will use the ***seriousgame*** web service.

URL	`http://146.191.107.189:8080/seriousgame/<idSG>/version/<version>`
Method	`GET`
Consumes	/

Produces	JSON representing the configuration file
	
	Example of the seriousgame section :
	`"seriousGame": {`
	`"genre": "Runner",`
	`"idDeveloper": 1,`
	`"ageMin": 10,`
	`"ageMax": 99,`
	`"description": "This is a mini game that trains you to identify the countries that form the European Union",`
	`"subject": "geography",`
	`"name": "EU mouse",`
	`"public": true,`
	`"lang": "EN",`
	`"country": "UK"`
	`}`

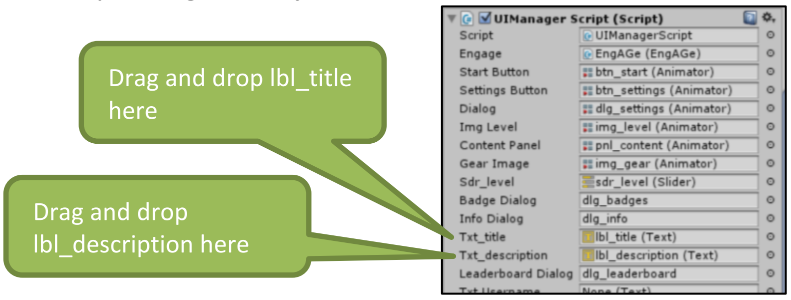
1. In *UIManagerScript.cs*, find the "*Start*" function. If the scene loaded in *MenuScene*, we want to call engage.getGameDesc(idSG). This will set engage's *seriousGame* variable to the json containing your configuration file parsed.

```
else if (Application.loadedLevelName.Equals("MenuScene"))
{
        // retrieve EngAGe data about the game
        StartCoroutine(engage.getGameDesc(idSG));

        /* [...] */
}
```

2. Add two public variables to the class, two text elements. One displays the title of the game, the other that displays its description.

```
public Text txt_title;
public Text txt_description;
```

3. In the Unity editor, in the *MenuScene*, attach the appropriate text labels (*Canvas>dlg_info>lbl_title* and *lbl_description*) to the newly created fields in the *EventSystem* game object. Here is how it should look like:


Drag and drop lbl_title here
Drag and drop lbl_description here

UIManager Script (Script)

Script	UIManagerScript
Engage	EngAGe (EngAGe)
Start Button	btn_start (Animator)
Settings Button	btn_settings (Animator)
Dialog	dlg_settings (Animator)
Img Level	img_level (Animator)
Content Panel	pnl_content (Animator)
Gear Image	img_gear (Animator)
Sdr_level	sdr_level (Slider)
Badge Dialog	dlg_badges
Info Dialog	dlg_info
Txt_title	lbl_title (Text)
Txt_description	lbl_description (Text)
Leaderboard Dialog	dlg_leaderboard
Txt Username	None (Text)

4. In the UI Manager Script, find the "*OpenInfo*" function, for now it only opens the information window. Add the following code to update the text labels with the game name and its description:

```
public void OpenInfo()
{
        // get the seriousGame object from engage
        JSONNode SGdesc = engage.getSG () ["seriousGame"];

        // display the title and description
        txt_title.text = SGdesc["name"];
        txt_description.text = SGdesc["description"];

        // open the window
        infoDialog.SetActive (!infoDialog.activeSelf);
}
```

If you play the game, and click the "about" button, you will see a window containing the information you gave in the configuration file.

Access to the player's badges

For now the badges window only displays "locked" badges, no matter how much you play, you can never unlock them. We need to retrieve the badges earned by a player from EngAGe.

For this task you will use the ***badges*** web service.

URL	`http://146.191.107.189:8080/badges/all/seriousgame/`**`<idSG>`**`/version/`**`<version>`**`/player/`**`<idPlayer>`**
Method	**`GET`**
Consumes	/
Produces	JSON array containing all the badges available in the game and specifying if the player earned it. For example: `[` `{`

```
            "message": "You played 10+ times",
            "name": "effort",
            "earned": false,
            "goalNum": 10,
            "playerNum": 8
        },
        {
            "message": "You found 50 EU countries",
            "name": "bronze_medal",
            "earned": true,
            "goalNum": 50,
            "playerNum": 76
        }
    ]
```

In the Unity editor, every badge already has a BadgeScript component that specifies an active image (for when the badge is unlocked). We need to attach EngAGe to this badge script and use it to retrieve the badges earned.

1. In *UIManagerScript.cs*, in the "*Start*" function. If the scene loaded in *MenuScene*, we want to call engage.getBadgesWon(idSG). This will set engage's *badges* variable to a json array containing the badges earned.

```
else if (Application.loadedLevelName.Equals("MenuScene"))
{
        // retrieve EngAGe data about the game, the badges earned
        StartCoroutine(engage.getGameDesc(idSG));
        StartCoroutine(engage.getBadgesWon(idSG));

        /* [...] */
}
```

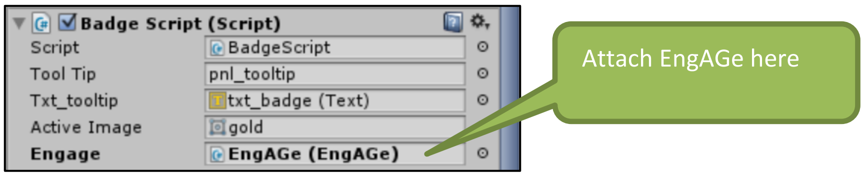
2. Open *BadgeScript.cs*
3. Add the following public variable `public EngAGe engage;` to the class.
4. Find the "Update" function and add the following code. It will get the badges earned from EngAGe and display the active image if the badge is part of the list.

```
void Update () {
        // get name of the badge represented
        string badgeName = this.name.Replace ("img_badge_", "");

        // if the badge is in EngAGe returned list, use the active image
        foreach (JSONNode b in engage.getBadges())
        {
                if (string.Equals(b["name"], badgeName) && b["earned"].AsBool)
                {
                        this.GetComponent<Image>().sprite = activeImage;
                }
        }
}
```

5. Find the "*OnPointerEnter*" function, it displays a custom tooltip describing the badge. For now, it only displays the default "description not available" text. Add the following code to display the badge message from your previously defined configuration file instead:

```
public void OnPointerEnter(PointerEventData data)
{
        // get the configuration file parsed in json format
        JSONNode sg = engage.getSG ();

        // get name of the badge represented
        string badgeName = this.name.Replace ("img_badge_", "");

        // update the description to the message defined in the config file
        // if no message is found the tooltip will display default message
        string desc = "description not available";
        if ((sg ["feedback"] != null) && (sg ["feedback"][badgeName] != null))
        {
                desc = sg ["feedback"] [badgeName] ["message"];
        }
        showToolTip (data.position, desc);
}
```

6. In Unity Editor, in the *MenuScene*, attach the EngAGe element to the new field ***for all eight badges***. Select a badge from the Hierarchy in *Canvas > dlg_badges > img_badge_xxx* and drag and drop EngAGe in the appropriate field in Badge Script. This is what the Inspector should look like for each badge:



Badge Script (Script)

Script	BadgeScript
Tool Tip	pnl_tooltip
Txt_tooltip	txt_badge (Text)
Active Image	gold
Engage	**EngAGe (EngAGe)**

If you play the game (starting from the Login scene), and click the "achievements" button, you should be able to view the badges description (on mouse over), all badges will still be locked as you haven't played the game while logged in yet.

Access to the game's leader board

For now, the leader board window doesn't display anything. We need to ask EngAGe to retrieve the gameplay data and display it there.

For this task you will use the ***leaderboard*** web service.

URL	`http://146.191.107.189:8080/learninganalytics/leaderboard/seriousgame/<idSG>/version/<version>`
Method	**GET**
Consumes	/

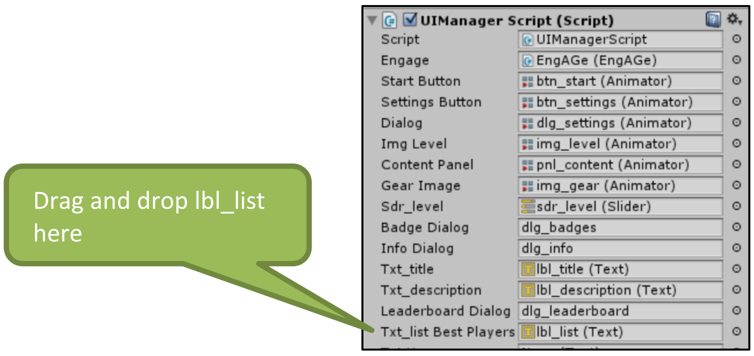
Produces	JSON object containing an array of performance for each score of the game. The performances are in descending order and are composed of a name and a score. Time is also represented (in seconds). For example:
	`{     "eu_countries": [`
	`        {    "name": "Anonymous",`
	`             "score": 28`
	`        },`
	`        {    "name": "yaelle",`
	`             "score": 0`
	`        }`
	`     ],`
	`     "eu_score": [`
	`        {    "name": "yaelle",`
	`             "score": 129`
	`        },`
	`        {    "name": "Anonymous",`
	`             "score": 112`
	`        }`

```
    ],
    "lives": [
        {   "name": "yaelle",
            "score": 3
        },
        {   "name": "Anonymous",
            "score": 0
        }
    ],
    "longestGameplays": [
         {
             "name": "Yaelle",
             "score": 328
         },
         {
             "name": "Anonymous",
             "score": 320
         }
     ],
     "shortestGameplays": [
         {
             "name": "Anonymous",
             "score": 32
         },
         {
             "name": "Anonymous",
             "score": 41
         }
     ]
}
```

1. In *UIManagerScript.cs*, in the "*Start*" function. If the scene loaded in MenuScene, we now want EngAGe to retrieve the leader board as well, update the code as follows:

```
else if (Application.loadedLevelName.Equals("MenuScene")) {
        // retrieve data about the game, the badges earned and the leader board
        StartCoroutine(engage.getGameDesc(idSG));
        StartCoroutine(engage.getBadgesWon(idSG));
        StartCoroutine(engage.getLeaderboard(idSG));

        /* [...] */
}
```

   This will set engage's leaderboard variable to a JSON containing the game best players and their scores.
2. Add one public variable to the class: `public Text txt_listBestPlayers;`, a text element that will display the list of the best players.
3. In the Unity editor, attach the appropriate text label (*Canvas>dlg_leaderboard>lbl_list*) to the newly created field. To do so, select *EventSystem* in the Hierarchy and drag and drop the text label in *Txt_list Best Players*. Here is how the Inspector of *EventSystem* should look like:



UIManager Script (Script)

Script	UIManagerScript
Engage	EngAGe (EngAGe)
Start Button	btn_start (Animator)
Settings Button	btn_settings (Animator)
Dialog	dlg_settings (Animator)
Img Level	img_level (Animator)
Content Panel	pnl_content (Animator)
Gear Image	img_gear (Animator)
Sdr_level	sdr_level (Slider)
Badge Dialog	dlg_badges
Info Dialog	dlg_info
Txt_title	lbl_title (Text)
Txt_description	lbl_description (Text)
Leaderboard Dialog	dlg_leaderboard
Txt_list Best Players	lbl_list (Text)

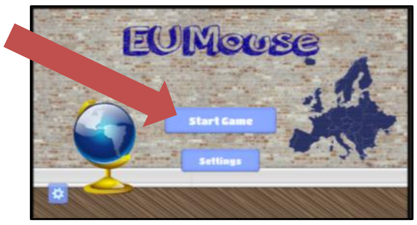
4. In *UIManagerScript.cs* "*OpenLeaderboard*" function, for now it only opens the leader board window. Add the following code to update the text label with a list of the best players:

```
public void OpenLeaderboard() {
        // get the leaderboard object from engage
        JSONNode leaderboard = engage.getLeaderboardList ();

        // look only at the eu_score
        JSONArray euScorePerf = leaderboard ["eu_score"].AsArray;

        // display up to 10 best gameplays
        int max = 10;
        txt_listBestPlayers.text = "";
        foreach (JSONNode gameplay in euScorePerf) {
                if (max-- > 0)
                {
                        // each gameplay has a "name" and a "score"
                        float score = gameplay["score"].AsFloat ;
                        txt_listBestPlayers.text += score + " - " +
                                gameplay["name"] + "\n";
                }
        }
        // open the window
        leaderboardDialog.SetActive (!leaderboardDialog.activeSelf);
}
```

If you play the game (from Login scene), and click the “leaderboard” button, you will see a window containing the list of the best players of your game (though, no-one played your game yet so you have an empty list).

Start a gameplay

When the player clicks on the “*Start Game*” button, we want EngAGe to know that the gameplay is linked to the player logged in (or the guest that gave player information). EngAGe should also initialise the scores based on the starting values defined in the configuration file.



For this task you will use the ***startGameplay*** web service.

URL	`http://146.191.107.189:8080/gameplay/startGP`
Method	**`PUT`**

Consumes	JSON object containing game and player data
	Example of JSON for a known player:
	`{ "idSG": 92,`
	`"version": 0,`
	`"idPlayer": 2`
	`}`
	Example of JSON for a guest:
	`{ "idSG": 92,`
	`"version": 0,`
	`"idStudent": 0,`
	`"params": [`
	`{ "name": "age",`
	`"type": "Int",`
	`"value": 26`
	`},`
	`{ "name": "gender",`
	`"type": "Char",`
	`"value": "f"`

```
        }]
}
```

Produces

JSON representing the ID of the gameplay created and ID of the player:

```
{
    "idGameplay": 183,
    "idPlayer": 28
}
```

EngAGe.cs include a “startGameplay” function, just call it with the game ID and the name of the scene to load. In *UIManagerScript.cs*, find the “*StartGame*” function; it’s the function called on click on the “Start Game” button. For now, it simply loads GameScene, replace the line as follows and it will initiate a gameplay with the student or guest details and save its ID.

```
public void StartGame() {
     StartCoroutine (engage.startGameplay(idSG, "GameScene"));
}
```

3.4 Use EngAGe functions – GameScene

When you play the game and select a country, at the moment, the country disappears but nothing else happens, we need three things:

1. Check if the country is correct (part of EU) and whether it has been found previously.
2. Update the scores and lives
3. Display a feedback to the player in the feedback panel (see icon in the bottom left)

Assess a player’s action

Because you already listed the EU countries in the configuration file, the good news is that there no need to hard code them! Handy no? Especially if the EU list changes in the future.

For this task you will use the ***assess*** web service.

URL	`http://146.191.107.189:8080/gameplay/<idGameplay>/assessAndScore`
Method	**PUT**

Consumes

JSON object containing the action name and a JSON of values for the parameters of the action. For example:

```
{
  "action": "newCountrySelected",
  "values": { "country": "france" }
}
```

Produces

A JSON object with two components, feedback (feedback triggered by the action) and scores (scores updated after the action). For example:

```
{
    "feedback": [
        {   "message": "Yes, france is indeed part of the EU",
            "name": "correct_country",
            "type": "POSITIVE"
        }
    ],
    "scores": [
        {   "startingValue": 28,
            "description": "countries of the EU left to find",
            "name": "eu_countries",
            "value": 27
        },
        {   "startingValue": 3,
            "description": "number of lives the player has",
            "name": "lives",
            "value": 3
        },
        {   "startingValue": 0,
            "description": "correct countries identified",
            "name": "eu_score",
            "value": 1
        }
    ]
}
```

1. Open *UIManagerScript.cs* (In *Assets > Scripts*) and create a new empty function

```
public void ActionAssessed(JSONNode jsonReturned) {}
```

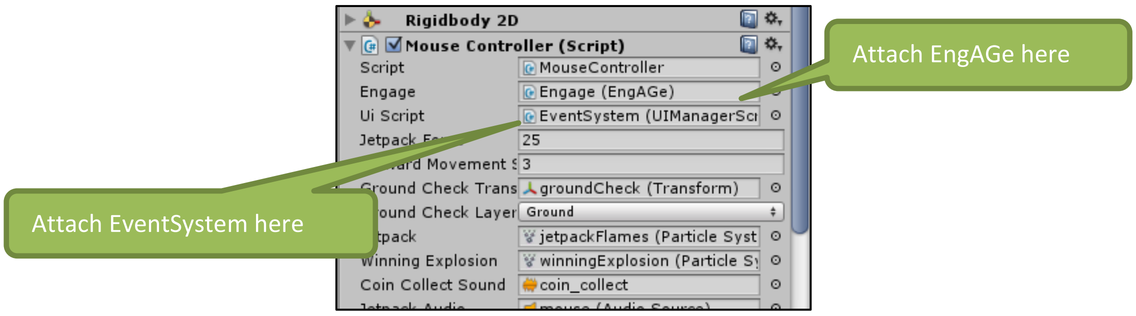
2. Open *MouseController.cs* (In *Assets > Scripts*)
3. Add the following public variables to the class

```
public EngAGe engage;
public UIManagerScript uiScript;
```

4. Find the "CollectFlag" function. This function is called when a collision is detected between the mouse (player) and a country flag.
5. Update the function as follows. It will check if the country selected is new, if so, call engage's *"assess"* function with *"newCountrySelected"* action, otherwise with "*countryReSelected*". Once EngAGe has assessed the action, it will call *ActionAssessed* from *UIManagerScript* with the JSON produced by the web service.
   Note - Make sure the names correspond to what you defined in your config file.

```
void CollectFlag(Collider2D flagCollider) {
        AudioSource.PlayClipAtPoint(coinCollectSound, transform.position);

        // get the name of the country selected
        Sprite spr_flag = flagCollider.gameObject.GetComponent<SpriteRenderer>().sprite;

        // country already selected
        if (countriesFound.Contains(spr_flag.name)) {
                // create a JSON with key/value "country" (only parameter in config file)
                JSONNode vals = JSON.Parse("{\"country\" : \"" + spr_flag.name + "\" }");
                // ask EngAGe to assess the action based on the config file
                StartCoroutine(engage.assess("countryReSelected", vals,
                                                        uiScript.ActionAssessed));
        }
        // country selected for the first time
        else {
                JSONNode vals = JSON.Parse("{\"country\" : \"" + spr_flag.name + "\" }");
                // ask EngAGe to assess the action based on the config file
                StartCoroutine(engage.assess("newCountrySelected", vals,
                                                        uiScript.ActionAssessed));
        }
        // save country selected
        countriesFound.Add (spr_flag.name);

        flagCollider.gameObject.SetActive (false);
}
```

6. In the Unity Editor, open the *GameScene*, attach the EngAGe element to the new field. Select the mouse game object from the Hierarchy, and drag and drop EngAGe and EventSystem in the appropriate fields. This is what the Inspector should look like:



Rigidbody 2D
Mouse Controller (Script)

Script	MouseController
Engage	Engage (EngAGe)
Ui Script	EventSystem (UIManagerScr
Jetpack F	25
ard Movement S	3
Ground Check Trans	groundCheck (Transform)
ound Check Layer	Ground
tpack	jetpackFlames (Particle Syst
Winning Explosion	winningExplosion (Particle Sy
Coin Collect Sound	coin_collect

Congratulations, the actions are now correctly assessed! Only you can't see the reaction as the scores are not updated in the GUI (The *ActionAssessed*() function is empty). Which is what we will work on next…

Update the scores

So, let see how we can update:

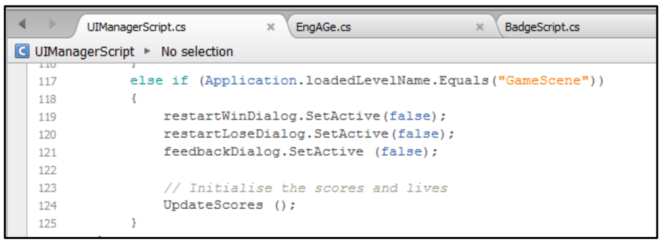
1. The numbers of EU left to find
2. The overall score (total EU found)
3. The lives left to the player

We need to update the GUI scores both in the start function and after an action has been assessed.

1. In *UIManagerScript.cs*, create an "*UpdateScores*()" function as follows, it will update the three scores mentioned earlier based on the score retrieved after the web services call:

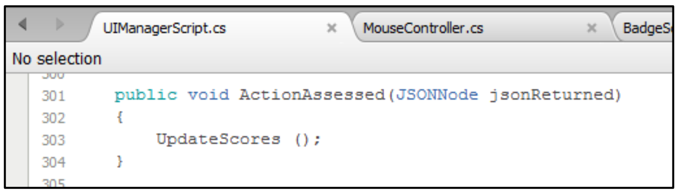
```
public void UpdateScores()
{
        foreach (JSONNode score in engage.getScores())
        {
                string scoreName = score["name"];
                string scoreValue = score["value"];

                if (string.Equals(scoreName, "eu_score"))
                {
                        pointsLabel.text = float.Parse(scoreValue).ToString();
                }
                else if (string.Equals(scoreName, "eu_countries"))
                {
                        euLabel.text = float.Parse(scoreValue).ToString();
                }
                else if (string.Equals(scoreName, "lives"))
                {
                        float livesFloat = float.Parse(scoreValue);
                        int lives = Mathf.RoundToInt(livesFloat);

                        life3.gameObject.SetActive(lives > 2);
                        life2.gameObject.SetActive(lives > 1);
                        life1.gameObject.SetActive(lives > 0);
                }
        }
}
```

2. Call that function in "*Start*()" (when the scene loaded is *GameScene*) so that the GUI is correctly initialised at the start of the game.



```
        else if (Application.loadedLevelName.Equals("GameScene"))
        {
            restartWinDialog.SetActive(false);
            restartLoseDialog.SetActive(false);
            feedbackDialog.SetActive (false);

            // Initialise the scores and lives
            UpdateScores ();
        }
```

3. Now find the "*ActionAssessed*" function that you created earlier.
7. This function is called by EngAGe after the engine successfully assessed a player's action, so it should call the *UpdateScores* previously created to update the score GUI:



```
public void ActionAssessed(JSONNode jsonReturned)
{
    UpdateScores ();
}
```

Play the game (from LoginScene) and check that the scores are updating. Now, if you played long enough you probably noticed that, even if you run out of lives, you keep going. We will fix that, and more, in next section.

Display and use feedback

In the bottom-left corner of the game scene, there is a button that opens a feedback window. We want to log here every feedback triggered in the game.

1. Create a public variable *txtFeedback* in *UIManagerScript.cs* to tell the game where to write the feedback (`public Text txtFeedback;`).
2. In Unity Editor, attach the *txt_feedback* gameobject (you can find it in the hierarchy *Canvas>dlg_feedback > img_feedback > txt_feedback*) to the field.
3. In *UIManagerScript.cs,* create an "*UpdateFeedback*" function. It will take the feedback received from EngAGe in parameter and will display them in the feedback window, positive feedback will be green, negative red and others will be black.

```
public void UpdateFeedback(JSONArray feedbackReceived)
{
        foreach (JSONNode f in feedbackReceived)
        {
                // set color of line
                string color = "black";
                if (string.Equals( f["type"], "POSITIVE"))
                        color = "green";
                if (string.Equals( f["type"], "NEGATIVE"))
                        color="red";

                txtFeedback.text += "<color=\"" + color + "\">" +
                                        f["message"] + "</color>\n";
        }
}
```

4. The function should also check if one of feedback triggered means the end of the game, or an adaptation of it. We had two adaptation feedback speedGame and slowGame, add this code in the foreach loop.

```
if (string.Equals(f["final"], "lose"))
{
        // tell the mouse it lost the game (to activate the die animation)
        mouseC.loseGame();
        // open a dialog window to go to menu or restart game
        restartLoseDialog.SetActive(true);
}
else if (string.Equals(f["final"], "win"))
{
        // tell the mouse it won the game (to activate the win animation)
        mouseC.winGame();
        // open a dialog window to go to menu or restart game
        restartWinDialog.SetActive(true);
}
else if (string.Equals(f["type"], "ADAPTATION"))
{
        if (string.Equals(f["name"], "speedGame"))
        {
                mouseC.forwardMovementSpeed += 1;
        }
        else if (string.Equals(f["name"], "slowGame"))
        {
                mouseC.forwardMovementSpeed -= 1;
        }
}
```

5. Call UpdateFeedback from "*ActionAssessed*" as follows.

```
public void ActionAssessed(JSONNode jsonReturned)
{
        UpdateScores ();
        UpdateFeedback (jsonReturned["feedback"].AsArray);
}
```

End a game

The last thing you need to do with EngAGe, is to send a request to end a game properly.

For this task you will use the ***endGameplay*** web service.

URL	`http://146.191.107.189:8080/gameplay/<idGameplay>/end/<win|lose>`
Method	`POST`
Consumes	/
Produces	An integer: - 1 if the game was correctly ended, - 0 if it was already ended, - -1 if the gameplay doesn't exist.

We will do that in the same *ReceiveFeedback()* function, when we find a feedback that triggers the end of the game. We will call `StartCoroutine (engage.endGameplay(boolWin));` with a Boolean to specify whether the game was won or lost. Here is the final *ReceiveFeedback()* function's code:

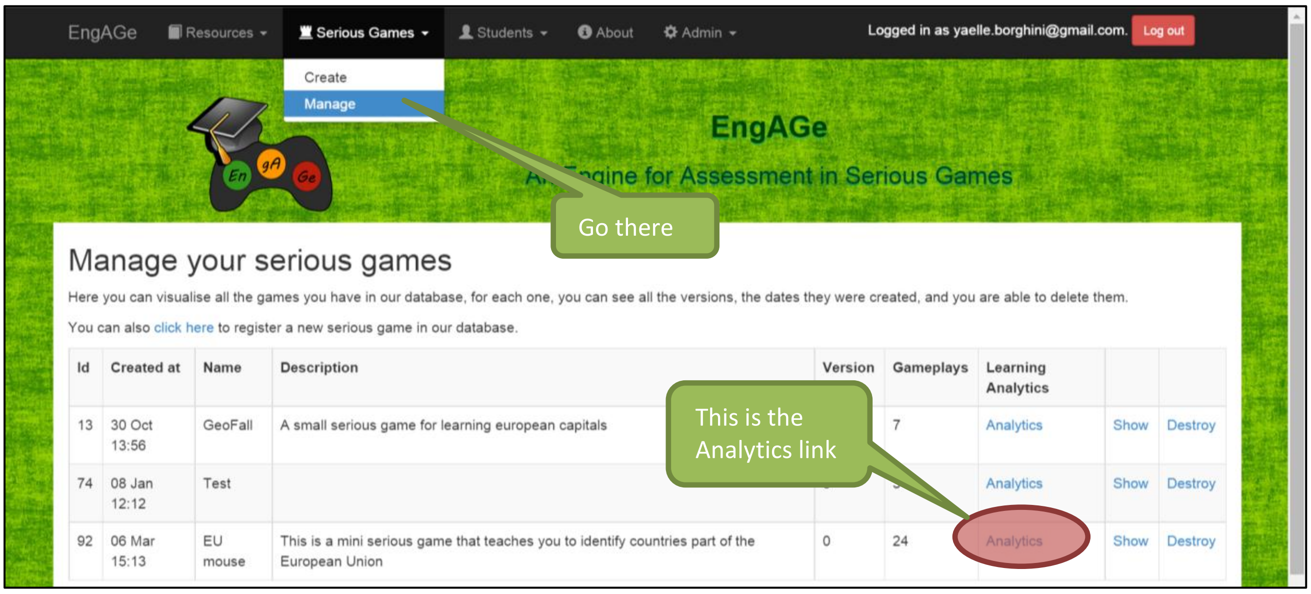
```
public void UpdateFeedback(JSONArray feedbackReceived)
{
        foreach (JSONNode f in feedbackReceived)
        {
                // set color to write line into
                string color = "black";
                if (string.Equals( f["type"], "POSITIVE"))
                        color = "green";
                if (string.Equals( f["type"], "NEGATIVE"))
                        color="red";
                txtFeedback.text += "<color=\"" + color + "\">" +
                                        f["message"] + "</color>\n";
                // trigger end of game?
                if (string.Equals(f["final"], "lose"))
                {
                        // tell EngAGe it’s the end of the game (lost)
                        StartCoroutine (engage.endGameplay(false));
                        // tell the mouse it lost the game
                        mouseC.loseGame();
                        // open a dialog window to go to menu or restart game
                        restartLoseDialog.SetActive(true);
                }
                else if (string.Equals(f["final"], "win"))
                {
                        // tell EngAGe it’s the end of the game (won)
                        StartCoroutine (engage.endGameplay(true));
                        // tell the mouse it won the game
                        mouseC.winGame();
                        // open a dialog window to go to menu or restart game
                        restartWinDialog.SetActive(true);
                }
                else if (string.Equals(f["type"], "ADAPTATION"))
                {
                        if (string.Equals(f["name"], "speedGame"))
                        {
                                mouseC.forwardMovementSpeed += 1;
                        }
                        else if (string.Equals(f["name"], "slowGame"))
                        {
                                mouseC.forwardMovementSpeed -= 1;
                        }
                }
        }
}
```

Congratulations!

You are done, everything is well integrated, you can play the game to check everything works fine. Remember to start from the login scene. If you are having problem, you can find the final code on the EngAGe branch of the git repo here:
https://github.com/yaelleUWS/eu_game/tree/with_engage.

4. Learning Analytics

Now that you have integrated EngAGe, you have access to its learning analytics dashboard, simply log onto the website, visit Serious Games > Manage, and click on the learning analytics link.


EngAGe
Resources
Serious Games
Students
About
Admin
Logged in as yaelle.borghini@gmail.com.
Log out
Create
Manage
EngAGe
An Engine for Assessment in Serious Games
Go there
This is the Analytics link

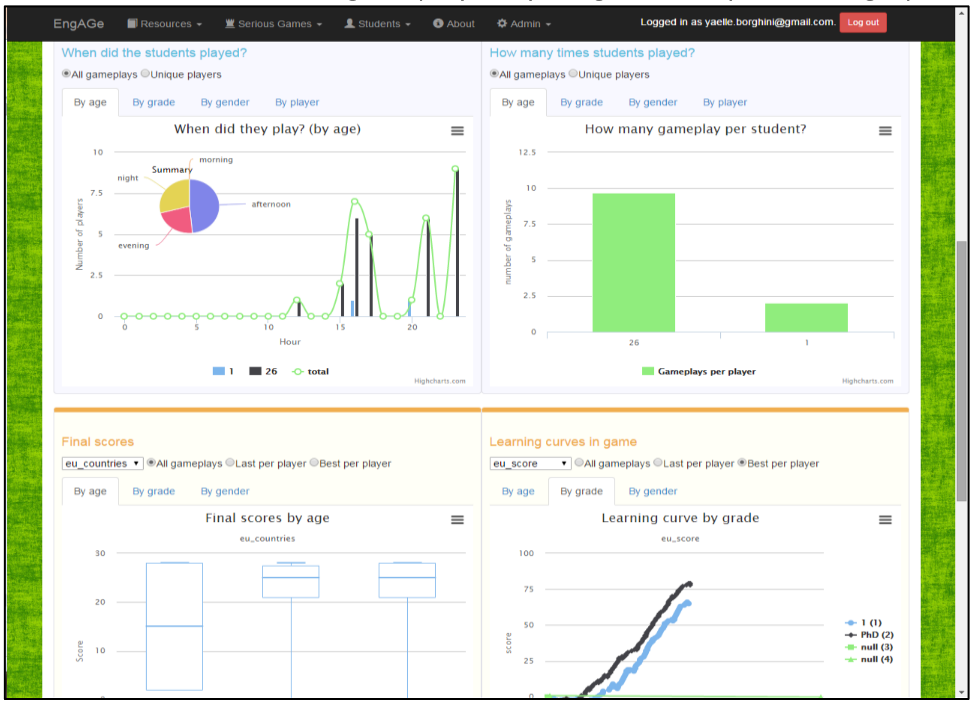
Manage your serious games

Here you can visualise all the games you have in our database, for each one, you can see all the versions, the dates they were created, and you are able to delete them.

You can also click here to register a new serious game in our database.

Id	Created at	Name	Description	Version	Gameplays	Learning Analytics		
13	30 Oct 13:56	GeoFall	A small serious game for learning european capitals		7	Analytics	Show	Destroy
74	08 Jan 12:12	Test				Analytics	Show	Destroy
92	06 Mar 15:13	EU mouse	This is a mini serious game that teaches you to identify countries part of the European Union	0	24	Analytics	Show	Destroy

This will retrieve all the data about the gameplays of your game and put it into graphics.



If you need more data, or would like to develop your own learning analytics dashboard, you can access the JSON with a GET request to: http://146.191.107.189:8080/learninganalytics/seriousgame/<yourGameID>/version/0

I hope you found the tutorial useful and easy enough to follow. Please take a moment to fill out EngAGe usability and usefulness questionnaire, it there: http://goo.gl/forms/UEpV2U9YwO It should take around 5-10 minutes to complete.

Thank you very much!