SmartWall SWGamePackage Manual

Embedded Fitness - 2023

Table of contents

Word of welcome

Hello, welcome to the Smartwall SWGamePackage manual. In this manual you'll find information about how to use the core features and tooling needed for the development of games that can run on the SmartWall. This guide assumes you've installed a version of the Unity game engine and created a blank project, have imported the SWGamePackage package into it, and have an Integrated Development Environment (IDE) like Visual Studio.

This document is a continuation of a previous document written by Joël van Huijkelom, and is currently being maintained by Sven Janssen. As I expect the toolset to be improved over time, this manual will be a living document and be updated periodically, which means it might be out of date at points. If there are features that will be added, changed or removed that are relevant to the existing documentation I'll write some commentary in [ ]-brackets.

Latest document update: 15-11-2023

User input

SWGamePackage contains code that allows the SmartWall input detection framework to interface with Unity games, and send inputs to be handled by the game. [I_SmartwallInteractable is the only script from the core features that you actually use in development, everything else gets taken care of for you.]

In order to make a game interact with player input, it first has to initialize the core processing features. There's an Init scene in Utils/Initialization which has the necessary components to start detecting hits. This scene has to be the first scene in the build order, and will automatically load the second scene once the initialization is ready. If you have any code that needs to be initialized on game boot as well, we recommend doing it in this scene.

The core functionality consists of the following scripts:

I_SmartwallInteractable

This interface has to be implemented in classes of objects that need to do something when they get hit. Note that in order to get hit, objects need to have a Collider (2D or 3D depending on the input type in BlobInputProcessing). When the object gets hit it calls the OnHit() function, which passes the position of the hit as a Vector3. The Z-axis of this vector does not get used with the 2D detection, but with the 3D detection it determines the depth of the hit object because it is a 3D space.

BlobDetectionGateway

This class establishes an tcp connection to the SmartWall detection code. The detection software then sends input data through this connection. This data gets processed into a list of Blob objects. This list is public and static, so other classes can easily access it from anywhere.

The class also has a NewBlobData event that will send a list of the new Blobs every time the class receives new data. Keep in mind that you will not receive new data every frame, hence the public List. You can subscribe to the event if you want to make sure a blob only gets processed once. It allows you to do a bit more with the blobs than only have objects respond on hit.

Blob

This class serves as a data container for input data, and contains the following data:

Type	Name	description
int	Id	The id of the input.
(UnityEngine.)Vector2	Position	The normalized screen space position of the blob.
float	Width	The normalized width of the ball.
float	Height	The normalized height of the ball.

BlobInputProcessing

This class takes the Blob data and translates it to interactions.
The class uses a cylinder cast / overlap circle the size of which is determined by the size of the ball as detected by the system. If AccountForBallSize is set to false a raycast from the center of the ball will be used instead.
All gameobjects with colliders in the casted path will have the Hit() method called on all their scripts implementing the I_SmartwallInteractible interface.

The InputType variable decides what the input gets processed as in the game; 2D, 3D or both at once.
This class also has functionality for processing mouse inputs into interactions for debugging purposes.

DevInitSceneSwitcher

DevInitSceneSwitcher is a tool that automatically switches the game to the initialization scene as you enter play mode. This allows the game to boot as it does normally, which allows you to test the game without constantly switching to the init scene yourself. To use the DevInitSceneSwitcher, simply place the Prefab with the same name in the scene you're working in.

Initialization

SWGamePackage contains an Init scene that initalizes various input and management tools. It should be the first scene in the build order, and automatically loads the next scene when fully initialized. The scene contains the following tools:

Level Manager

The Level Manager is a staticily available class that allows the game to switch between scenes using a scene transition animation. It gets instantiated in the Init scene.

Loading levels

- public void **LoadLevel**(int **index**, Transition **transition**)
- public void **LoadLevel**(string **name**, Transition **transition**)

Use LoadLevel() to load a new level, where the index is the index of the scene in the build order, and transition is an enum value of the specific animation. Every transition animation takes 2 seconds (1 second fade out, 1 second fade in), except Transition.None, which switches between scenes without playing an animation.
A level can also be loaded by scene name by replacing the int with a string.

```
0 references | 0 changes | 0 authors, 0 changes
public void LoadLevel()
{
    int index = 4;
    LevelManager.Instance.LoadLevel(index, Transition.Crossfade);
}
```

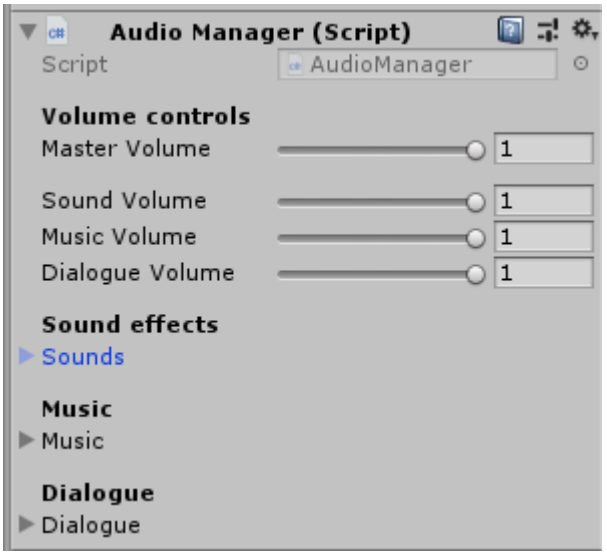
- public void **LoadNextLevel**(Transition **transition**)

You can also load the next scene in the build order by using the LoadNextLevel() function, where transition is the same enum value of the specific animation.

```
0 references | 0 changes | 0 authors, 0 changes
public void LoadNextLevel()
{
    LevelManager.Instance.LoadNextLevel(Transition.Crossfade);
}
```

Audio Manager

The audiomanager allows users to easily play and manipulate sound effects in code, while also providing a single place where the volume of the game can be managed.
The script consists of 3 categories of sound types (sounds, music & dialogue), each with a slider that allows the volume to be changed in real time. The Master volume slider changes the volume of all sliders at once.



Sounds

Each category contains a list of Sounds. A sound has the following Values:

Type	Name	description
string	Name	The name of the sound. Gets used to find the specific sound when changing sound settings in code.
float	Volume	The maximum level the sound effect plays at. A value between 0 and 1.
float	Pitch	The pitch the sound effect plays at. A value between 0 and 3.
bool	Loop	Makes the sound automatically restart when it's finished playing
bool	PlayOnAwake	Starts playing the sound the moment the game gets booted.
bool	SilentPlay	Plays the sound at 0% volume when started. Tends to be useful for dynamic music.
List<AudioClip>	Clips	A list of clips to play. Plays the first one by default but can be used to randomize between different variations of sounds.

Sound instancing

You can create and destroy instances of sounds during runtime, for example to create multiple instances of a sound for copies of objects, in order to prevent multiple objects trying to access the same sound at once.

- public string **CreateSound**(string **name**)

Create a copy of an existent sound, using the original's name as a unique identifier. Returns the name of the copy sound as a string.

```
private string SoundInstance;

0 references | 0 changes | 0 authors, 0 changes
public void CreateNewSoundInstance()
{
    string sound = "JumpSound";
    SoundInstance = AudioManager.Instance.CreateSound(sound);

    AudioManager.Instance.Play(SoundInstance);
}
```

- public void **DestroySound**(string **name**)
- public void **DestroySound**(string **name**, float **delay**)

Destroys an instance of a sound after a certain delay, using the sound's name as a unique identifier.

```
private string SoundInstance;

0 references | 0 changes | 0 authors, 0 changes
public void DestroySoundInstance()
{
    float delay = 1.2f;
    AudioManager.Instance.DestroySound(SoundInstance, delay);
}
```

Playing

Start playing a sound with a certain name. You can add an index to specify a specific variation clip from the Clips list (as specified in Sounds). Alternatively you play a specific (external) sound clip by adding an AudioClip. You can also play the sound after a delay by adding a float.

- public void **Play**(string **name**)
- public void **Play**(string **name**, int **index**)
- [Obsolete] public void **Play**(string **name**, AudioClip **clip**)
- [Obsolete] public void **Play**(string **name**, string **path**)
- public void **Play**(string **name**, float **delay**)

This method is currently the main way to randomize the sound clip that gets played. Note: the second parameter of random.Range is exclusive, make sure you plug in the correct data!

```
0 references | 0 changes | 0 authors, 0 changes
public void PlaySound()
{
    string soundName = "JumpSound";
    int clipIndex = UnityEngine.Random.Range(0, 4);
    AudioManager.Instance.Play(soundName, clipIndex);
}
```

The method loads a specific AudioClip from the resources folder, and starts playing it. It automatically adds Assets/Resources so in this example the path so your assets can be found at [project directory]/Assets/Resources/Sounds/Sound01.wav
[this one gets practically never used, isn't tested very well].

```
0 references | 0 changes | 0 authors, 0 changes
public void PlaySound()
{
    string soundName = "JumpSound";
    string path = "Sounds/Sound01";
    AudioManager.Instance.Play(soundName, path);
}
```

Stopping

Stops playing a sound with a specific name. A time variable can be added to stop the sound after a certain amount of time. Not to be confused with FadeOut().

- public void **Stop**(string **name**)
- public void **Stop**(string **name**, float **time**)

```
0 references | 0 changes | 0 authors, 0 changes
public void StopSound()
{
    string soundName = "JumpSound";
    float delay = 1.2f;
    AudioManager.Instance.Stop(soundName, delay);
}
```

Pausing

Pauses or unpauses a sound with a certain name.

- public void **Pause**(string **name**)
- public void **Unpause**(string **name**)

Volume

Sets the volume of a sound with a certain name to a certain volume. The volume is a float between 0 and 1. A float time variable can be added to change the volume over an amount of time.

- public void **SetVolume**(string **name**, to)
- public void **SetVolume**(string **name**, to, time)

FadeIn() and FadeOut() fade the volume of a sound with a certain name from 0 to 1 or from 1 to 0 depending on the method used. FadeOut has a stop variable that decides whether the sound stops playing once the volume reaches 0, or keeps playing at 0% volume.

- public void **FadeIn**(string **name**, float **time**)
- public void **FadeOut**(string **name**, float **time**, bool **stop** = false)

Pitch

Sets the pitch of a sound with a certain name, or randomizes the pitch between 2 values.

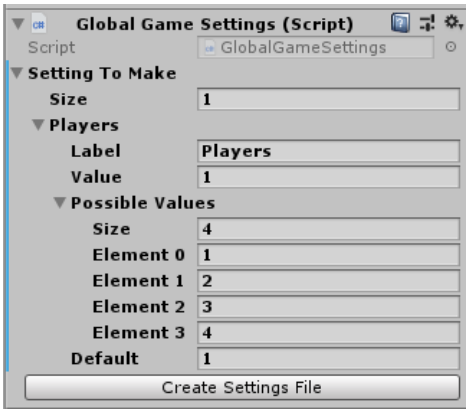
- public void **SetPitch**(string **name**, float **value**)
- public void **RandomizePitch**(string **name**, float **min**, float **max**)

Looping

Sets whether or not the sound with a specific name automatically starts playing again after it's done playing.

- public void **SetLoop**(string **name**, bool **state**)

Global game settings



The Smartwall is controlled using an android app on a tablet. In this app users can start games and alter the gameplay by selecting various options. These options are made by developers and vary from game to game. The Init scene contains a GlobalGameSettings script on the GameMaster object which allows you to easily generate possible options.

The options consist of a list with option items with the following values:

Type	Name	description
string	Label	The name of the option. This name only gets used internally, and gets translated in the app to match the user language and tone of voice.
string	Value	The currently selected option. This string should match one of the strings in PossibleValues.
List<string>	PossibleValues	A list of all selectable option values.
string	Default	The default option value. [This doesn't currently get used, the default value is whatever is in Value.]

In order to use the settings in game, Load in the setting in the relevant script using GlobalGameSettings.GetSetting(string nameOfSetting)
Create an if-else statement or switch case and go through the potential values and execute code based on the loaded value.

```
0 references | Sven Janssen, 26 days ago | 2 authors, 5 changes
void Start()
{
    string setting = GlobalGameSettings.GetSetting("Players");
    if (setting == "4")
    {
        // do things for 4 players
    }
    else if (setting == "3")
    {
        // do things for 3 players
    }
    else if (setting == "2")
    {
        // do things for 2 players
    }
    else //if (setting == "1")
    {
        // do things for 1 player
    }
```

Lerp Curves

LerpCurves allow you to add some animation to otherwise static runtime animations. For example: if you have an object that has to move from point A to B, you can use a coroutine and lerpcurve to easily smooth out or add weight to the movement.

Methods

- public float **Linear**(float **time**)
- public float **EaseIn**(float **time**)
- public float **EaseOut**(float **time**)
- public float **EaseInOut**(float **time**)
- public float **Bounce**(float **time**)

```
0 references | 0 changes | 0 authors, 0 changes
private IEnumerator _MoveObject(Transform obj, Vector2 from, Vector2 to)
{
    float progress = 0;
    while (progress < 1)
    {
        progress = Mathf.Clamp01(progress + Time.deltaTime);
        obj.position = Vector2.Lerp(from, to, LerpCurves.Instance.EaseInOut(progress));

        yield return null;
    }
}
```

Player colour container

The PlayerColourContainer class allows you to get the color and icon that is associated with a specific player during runtime.

- public static Colour **GetPlayerColour**(int **playerNumber**)
- public static Sprite **GetPlayerIcon**(int **playerNumber**)

[This component is pretty outdated and needs to be reworked. I recommend spawning Prefabs that come with specific player colours and icons instead of spawning a neutral Prefab that then has to load the correct player data.]

Scores

The score scene is a scene that's included in SWGamePackage, and allows you to easily tally scores and showcase the winner of a round. In order to use the score scene, Add the scene to the build order and call the MoveToScores() function:

- public static void **MoveToScores**(
  List<int> **scores**,
  int **levelIndex** = 1,
  int **sceneIndex** = 1,
  bool **ignore0Scores** = true)

```
0 references | 0 changes | 0 authors, 0 changes
public void GoToScoreScreen()
{
    List<int> scores = new List<int>();
    scores.Add(12);
    scores.Add(8);
    scores.Add(14);
    scores.Add(3);

    int levelIndex = 1; //I still don't know what this means
    int SceneToReturnTo = 1;

    bool Ignore0Scores = true;

    ScoreScreenController.MoveToScores(scores, levelIndex, SceneToReturnTo, Ignore0Scores);
}
```

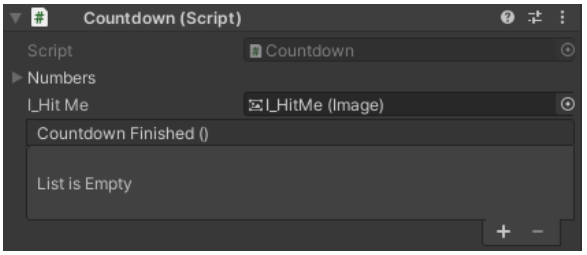
The **scores list** is a list of ints representing the amount of points each player scored during the round of the game.
The **LevelIndex** parameter allows you to save different sets of highscores for different levels or difficulty settings.
The **sceneIndex** decides the build order index of the scene that the game returns to if the replay button gets hit.
Ignore0Scores decides whether or not the scorescreen should display players that didn't score any points, or not. This functionality exists mainly because users sometimes boot up the game with the wrong number of players, and this makes the score screen a bit more legible.

All values except the score list have default values, so they can be left empty if needed.

Gameplay elements

SWGamePackage contains a couple of Game elements; small tools of features that we noticed got used across multiple games, and made reusable code for. The package contains the following elements:

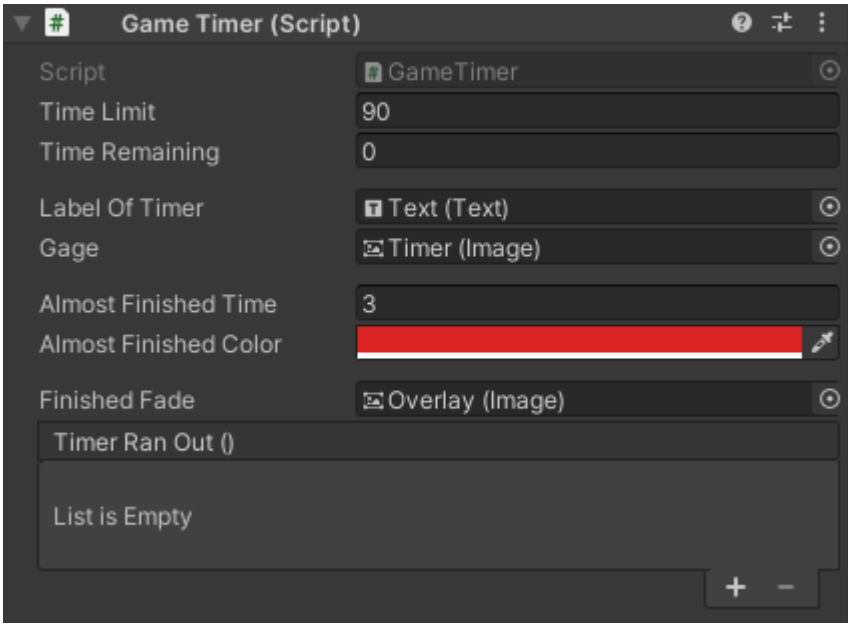
StartScreen



The startscreen is a screen at the start of your gameplay scene that adds a countdown before the game starts, while preventing any input to other gameplay elements (because it is in front of them and ‘catches’ the input check). To use the feature, drag the P_StartCountdown prefab into your gameplay scene. The countdown gets started when a player hits the screen, as indicated with the moving icon. After the countdown is finished, the startscreen gets disabled so inputs can be made on other game elements, and the CountdownFinished() event gets called, so you can start your game logic.

GameTimer

Game timer is, as the name implies, a timer that determines the length of a round by counting down. To use a game timer, drag the Timer prefab into the scene.



After the timer runs out, the timer blocks any player input by enabling a semi transparent grey box (similar to StartScreen), and invoke the TimerRanOut() event after a couple of seconds. This mostly gets used to transition to the Score scene after the game is over to create gameplay loop.

You can set the default time limit by changing the TimeLimit variable in the editor, and start the timer with the StartTimer() method. Alternatively you can use StartTimer(float time), which starts the timer with a custom time.

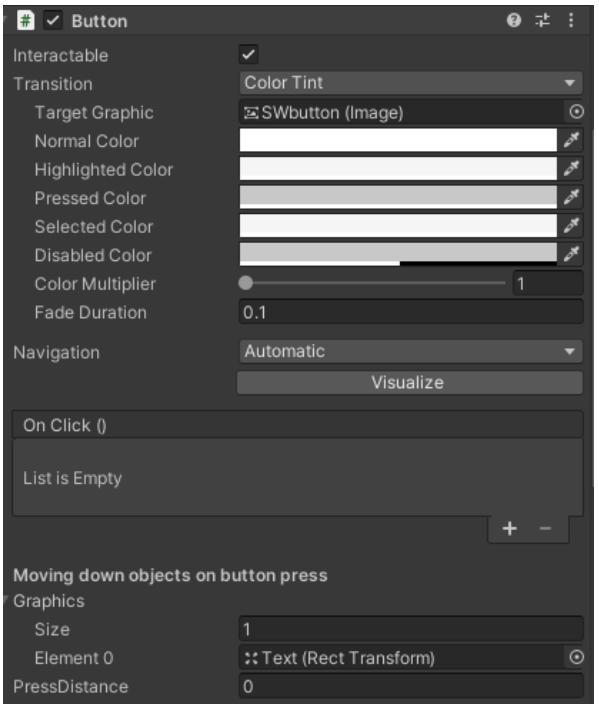
You can pause and unpause the timer with the PauseTimer(bool pause).

GameStopwatch

Like game timers, stopwatches measure the length of a round, but count up until the players fulfil a condition instead. You can stop the timer by calling the Finish() method, and read out the timer value from the TimeRemaining variable after the timer is finished.

SWButton

SWButton is an extension of the standard Unity button which adds functionality for it to also react to Smartwall input. To use the button, simply drag the SWButton prefab into your GUI (it's a Unity button, so it needs to be inside a Canvas). Besides the prefab having a 2D collider, the core functionality is the same as a standard unity button.



The main difference is the "Moving down objects on button press" section at the bottom of the script. This feature allows you to add a list of objects that need to be moved when you press the button, and how much they need to be moved on the y-axis. This way you can easily move child objects with the button graphic, so it looks like they're a single object.

SWLabel

SWLabel are standardized labels for displaying player scores. You can easily set up scorelabels for up to 4 player by dragging the Labels prefab into your scene. The prefab contains 4 color-coded labels with SWLabel scripts, and a horizontal layout group to manage their position in the scene while allowing you to turn off labels that aren't being used.

You can update the value of the label with the UpdateLabel(string value) function, and update the color of the label with the UpdateBackground(Color color) function.

Obsolete features

Language

The language components allow you to translate text labels into multiple languages, but has not been maintained for several years because we have shifted our approach away from language, and instead use elements like color or sound to convey meaning. This code is vestigial, and doesn’t work so don’t use it.