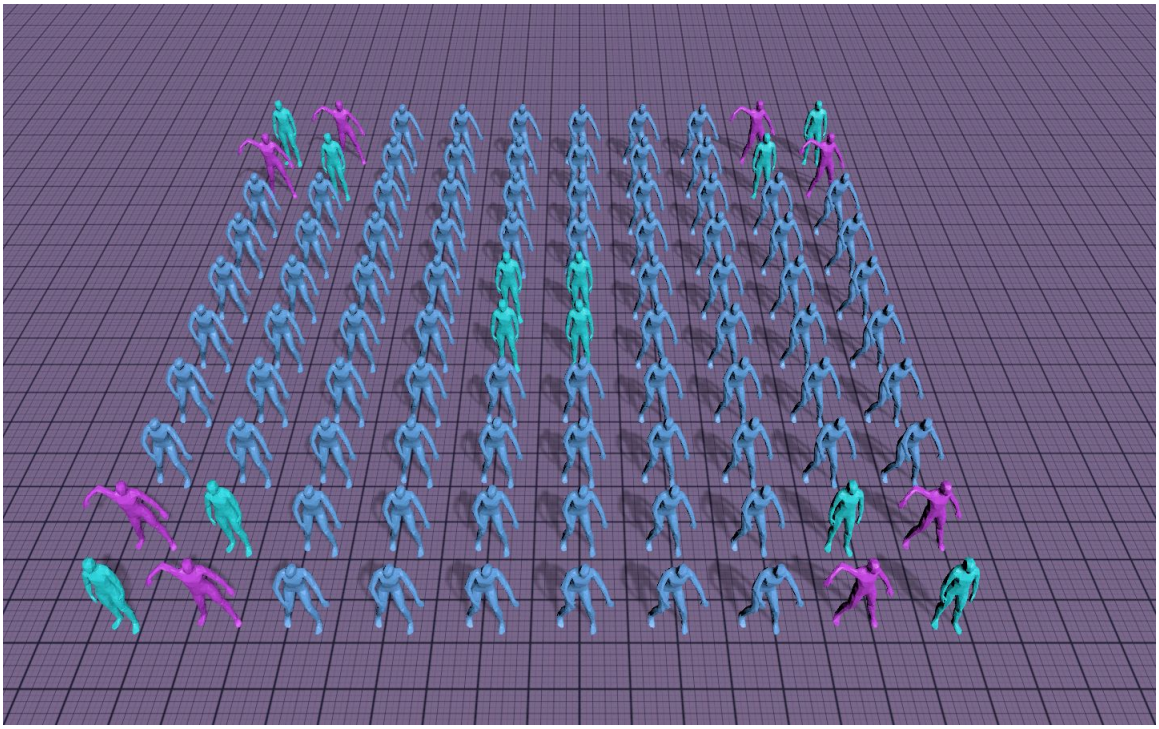
UDMSlua Manual

UDMSlua 2020.1



N. Katomeris and Ath. Kehagias

Thessaloniki, Greece
2020

Contents

1 Introduction **5**
- 1.1 What Is UDMSlua ? . . . 5
- 1.2 QuickStart . . . 6
- 1.3 Keyboard Shortcuts . . . 7
  - 1.3.1 Main Menu . . . 7
  - 1.3.2 Scenaria . . . 8

2 The Structure of UDMSlua **12**
- 2.1 Introduction . . . 12
- 2.2 The Components of UDMSlua . . . 12
- 2.3 Scenaria and Domains . . . 12
- 2.4 Creating Additional Domains . . . 14
- 2.5 Calling Domain Functions . . . 14
- 2.6 Rooms . . . 14
- 2.7 Lua scripting . . . 14

3 The Scenaria **19**
- 3.1 A Simple Attraction-Repulsion Scenario . . . 19
  - 3.1.1 The `settings.lua` Script . . . 20
  - 3.1.2 The `group.lua` Script . . . 22
  - 3.1.3 The `camera.lua` Script . . . 25
- 3.2 Epidemic Dances . . . 26
  - 3.2.1 Intuitive Description of the Model . . . 26
  - 3.2.2 Mathematical Description of the Model . . . 27
  - 3.2.3 Epidemic Dance on a Circle . . . 28
- 3.3 Game Theoretic Dances . . . 34
  - 3.3.1 Intuitive Description of the Model . . . 34
  - 3.3.2 Mathematical Description of the Model . . . 34
  - 3.3.3 Game Theoretic Dance on a Grid . . . 36
- 3.4 Logical Dances . . . 40
  - 3.4.1 Intuitive Description of the Model . . . 40
  - 3.4.2 Multivalued Logics and Logical Operators . . . 41
  - 3.4.3 Mathematical Description of the Model . . . 43
  - 3.4.4 Logic Dance on a Circle . . . 44

A UDMSlua API **48**
A.1 Lua Camera Object . . . 48
A.1.1 Lua Camera Object Properties . . . 48
A.1.2 Lua Camera Object Methods . . . 50
A.2 Lua Game Object . . . 53
A.2.1 Lua Game Object Properties . . . 53
A.2.2 Lua Game Object Methods . . . 54
A.3 Lua Group Domain . . . 57
A.3.1 Lua Group Domain Properties . . . 57
A.3.2 Lua Group Domain Methods . . . 57
A.4 Lua Group Object . . . 64
A.4.1 Lua Group Object Properties . . . 64
A.4.2 Lua Group Object Methods . . . 65
A.5 Lua Individual Domain . . . 72
A.5.1 Lua Individual Domain Properties . . . 73
A.6 Lua Room . . . 73
A.6.1 Lua Room Properties . . . 73
A.6.2 Lua Methods . . . 74
A.7 Scripting Examples . . . 80
A.7.1 Individual object controlling scripts . . . 80
A.7.2 Group object controlling scripts . . . 81
A.7.3 Using custom prefabs . . . 82
A.7.4 Create a checker board . . . 83
A.7.5 Print text on screen . . . 83
A.7.6 Play an animation from the available animations . . . 84
A.7.7 Access a IK limb game object . . . 84
A.7.8 Create a group of different objects . . . 84
A.7.9 Enable effects shortcuts . . . 84
A.7.10 Enable a LUT effect with a specific texture . . . 85

B UDMSlua Libraries **86**
B.1 Animations . . . 86
B.2 Camera Functions . . . 92
B.2.1 Basic Camera Functions . . . 92
B.2.2 General Camera Functions . . . 93
B.2.3 CineMachine Functions . . . 95
B.3 Group Functions . . . 96
B.3.1 Animation Functions . . . 96
B.3.2 Formations . . . 97
B.3.3 GCA Functions . . . 98
B.3.4 Group Functions . . . 99
B.3.5 Group Member Functions . . . 100
B.3.6 Inverse Kinematics Functions . . . 105
B.3.7 Navigation Functions . . . 106
B.3.8 Trail Renderer Functions . . . 107

B.4 Logic Functions 108
B.5 Luts 109
B.6 Neighborhoods 110
B.7 Object Functions 111
B.7.1 Light Functions 111
B.7.2 Object Functions 112
B.7.3 Trail Renderer Functions 115
B.8 Room Functions 116
B.9 Utility Functions 118
B.10Visual Effects 120

C Asset Bundles **122**
C.1 Models 122
C.2 Textures 122
C.2.1 textures/dungeon 122
C.2.2 textures/ground 123
C.2.3 textures/scifi 125
C.2.4 textures/various 126
C.3 LUTs 127

D Extending UDMSlua **129**
D.1 Adding Lua Scripts 129
D.2 Adding Lua Libraries 129
D.3 Adding Streaming Assets (Music Files etc.) 131
D.4 Adding Resources 131
D.5 Adding C# Scripts 132

E Imported Assets Used in UDMSlua **135**
E.1 External Packages 135
E.2 Models 135
E.3 Animations 135
E.4 Music 135
E.5 Textures 136

Chapter 1

Introduction

1.1 What Is UDMSlua ?

UDMSlua (*Unity Dance and Motion Scripter in Lua*) is an application for designing and visualizing human movement in 3D. UDMSlua is being developed with the Unity engine and provides Lua scripting support for creating *dance scenaria*.

The scenaria programming interface of UDMSlua was designed to give the user maximum design freedom by providing access through Lua scripts to

1. most of the Unity Engine's native scripting API,
2. new methods that simplify the creation of dance scenaria.

It also facilitates Lua scripting by supporting the use of many different scripts for each scenario and supports some advanced runtime functionality, such as running code snippets or executing commands on selected domains of a scenario. The scripting API is also extensible by supporting user defined libraries.

The UDMSlua distribution contains Lua scripts which implement several categories of dance models that (hopefully) produce interesting and aesthetically pleasing results. Some of the included model categories are: *epidemic*, *game theoretic* and *logical* dances.

UDMSlua has been created and tested with Unity2019.4.10f1 on Windows (7, 8 and 10). The project, can be built and run on linux (x86_64) and MacOS. However, it hasn't been tested on those platforms.

Nota Bene: This document is currently incomplete (in fact it is a never ending work in progress). This chapter is fairly complete and should give you all the information you need to start using UDMSlua with all the pre-installed scenaria. The following chapters will give you some (but not all) information to do the following.

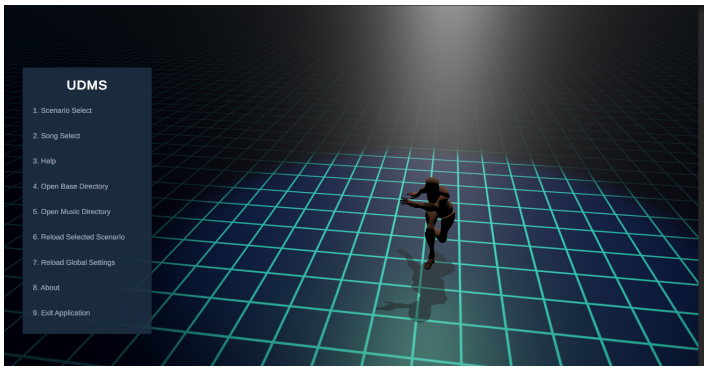
1. Create your own scenaria. To this end, in addition to Chapters 2 – B of the current document, you should study carefully the scenaria contained in the folder

   `../StreamingAssets/Scripts`.

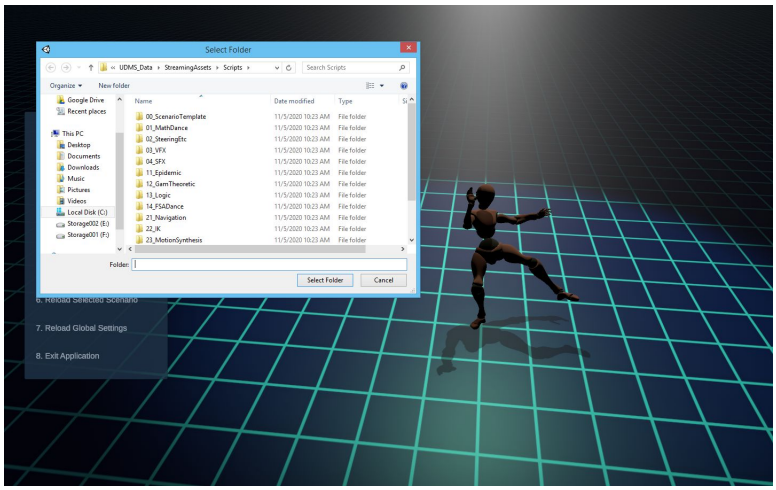
2. Modify UDMSlua. For this you should study the following chapters (especially D) and experiment with the UDMSlua project files.

1.2 QuickStart

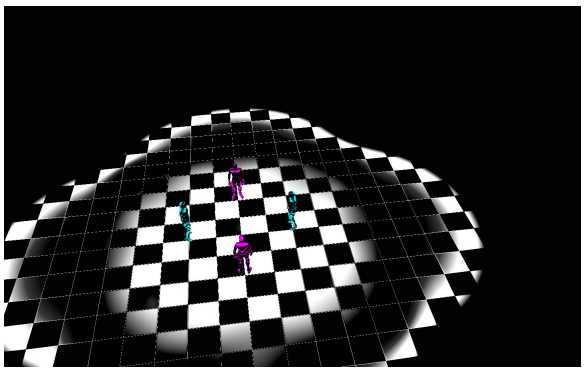
1. Start the application. You will see the Intro screen with the *Main Menu*:

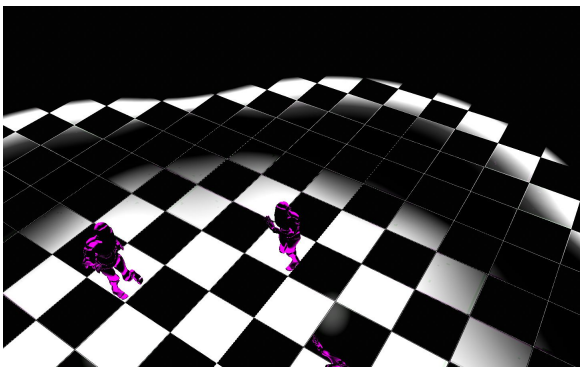


2. Click on Scenario Select; a file dialog will open and you will see this



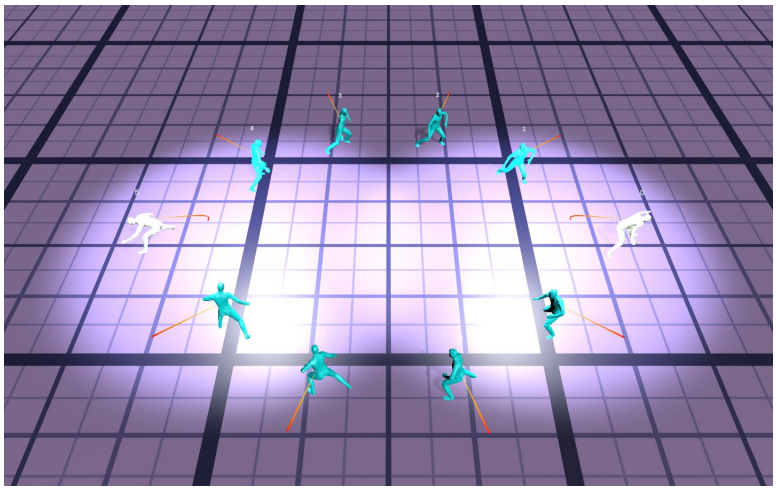
Open one of the folders and select one of the included subfolders. To be specific, open `13_Logic`, click on `LOG0091` and then click on Select Folder. You will see this (animated)





In this example you have no control over the camera (it follows a prespecified path).

3. Let's try another scenario. Hit Esc; the Main Menu will show up. Click again on Scenario Select and in the file dialog select `11_Epidemic` and then `EPI0001`. You will see this



In this (as in most) scenario you can control the camera with the mouse. The preselected behavior is a *free camera*, which behaves much like the Unity Editor camera: right mouse button orbits, middle mouse button pans and mouse wheel zooms in/out.There are six default camera behaviors, you activate each one by hitting one of buttons F1 to F6. You can program (script) additional camera behaviors.

4. You can exit the application by showing the Main Menu (hit the Esc key) and clicking on Exit Application.

5. You can show/hide the Main Menu by hitting the Esc key. When the Main Menu is visible, you can select one of the nine choices with the mouse; you can also use the key combinations Left Alt+1, ..., Left Alt+9, *whether the Main Menu is visible or not*. We will discuss choices 2 to 8 in later chapters, but feel free to experiment with each of them right now.

6. For the time being, let us just mention that the most interesting choice is Left Alt+4 (Open Base Application). It opens a file dialog which starts in the Scripts folder. Here you can go to a scenario folder, open the Lua script files with a text editor and modify the corresponding dance scenario. Or you can create a new folder and write your own scripts.

1.3 Keyboard Shortcuts

In this section we give a brief description of the default keyboard shortcuts.

1.3.1 Main Menu

You can show/hide the Main Menu by hitting Esc. If the Main Menu is visible you can select with the mouse (and if the Main Menu is either visible or invisible with Alt+1, ..., Alt+8) the following actions.

1. Scenario Select. Opens a file dialog in the Scripts (base) folder; here you can select a scenario subfolder, which will be played by the application.
2. Song Select. Opens a file dialog in the Music folder; here you can select a song, which will be played by the application. Songs have to be in the `*.ogg` format.
3. Help. Opens the help panel which contains links to the online docs, this help file (`UDMSluaHelp.pdf`) and a Quick Start guide.
4. Open Base Directory. Opens a file dialog in the Scripts (base) folder. Now you can select a scenario subfolder, open the included Lua scripts and modify them.
5. Open Music Directory. Opens a file dialog in the Music folder.
6. Reload Selected Scenario. Reloads and runs the last selected scenario.
7. Reload Global Settings. Reloads the global game settings.
8. About. Version info and credits panel.
9. Exit Application. Terminates the application.

In addition to the above you can also bring up the *console* by hitting the tilde key ~. The use of the console is covered in Chapter 2 of the UDMSlua Manual.

1.3.2 Scenaria

When you are in one of the Scenaria you can use the following keys.[1]

1. Esc: Show/hide the **Main Menu**.
2. $<$ or , : Decreases the **music volume**.
3. $>$ or . : Increases the **music volume**.
4. F1-F6: **Camera Keys**; each one activates a certain camera behavior, as described below.
5. LeftShift+1, ..., LeftShift+0: **Target Selection Keys**, described below.
6. RightShift+1, ..., RightShift+0: **Visual Effects Keys**, described below.
7. ~ (the tilde key): Show/hide the **Console**. The use of the console is covered in Chapter 2 of the UDMSlua Manual.

[1] Provided that the shortcut listener library functions are being called in an update()

Camera State Selection Keys

1. F1: activates a user-controlled *free camera*. It is similar to the Unity Editor Camera, controlled with the mouse.

   RMB: Orbit.

   MMB: Pan.

   Wheel: Zoom in/out.

2. F2: activates a user-controlled *free camera*. You can use the following keys.

   UpArrow: Move camera forward.

   DownArrow: Move camera backward.

   PgUp: Rotate camera upwards around x-axis.

   PgDn: Rotate camera downwards around x-axis.

   W: Increase X coordinate.

   S: Decrease X coordinate.

   A: Increase Z coordinate.

   D: Decrease Z coordinate.

   E: Increase Y coordinate.

   Q: Decrease Y coordinate.

3. F3: activates a *top-down view* camera following the *camera target* (a target is **required** and selected by the *target selection keys*, see below). You can use the following keys.

   UpArrow: Decrease Y coordinate.

   DownArrow: Increase Y coordinate.

4. F4: activates a user-controlled *top-down view* camera. You can use the following keys.

   W: Increase X coordinate.

   S: Decrease X coordinate.

   A: Increase Z coordinate.

   D: Decrease Z coordinate.

   E: Increase Y coordinate.

   Q: Decrease Y coordinate.

5. F5: activates a camera which *follows* the *camera target* (a target is **required** and selected by the *target selection keys*, see below). You can use the following keys.

   UpArrow: Move camera forward.

DownArrow: Move camera backward.

PageUp: Increase Y coordinate.

PageDown: Decrease Y coordinate.

6. F6: activates a camera which *orbits* around the *camera target* (a target is **required** and selected by the *target selection keys*, see below). You can use the following keys.

   UpArrow: Decrease orbit radius.

   DownArrow: Increase orbit radius.

   PageUp: Increase Y coordinate.

   PageDown: Decrease Y coordinate.

Target Selection Keys

Some camera behaviors require a *camera target*, which can be any of the dancers / agents. Camera targets can be set by script (see Chapter 2 of the UDMSlua Manual) or by the following keys (for the first 9 agents).

1. LeftShift+1: set the target to Agent No.1.
2. LeftShift+2: set the target to Agent No.2.
3. LeftShift+3: set the target to Agent No.3.
4. LeftShift+4: set the target to Agent No.4.
5. LeftShift+5: set the target to Agent No.5.
6. LeftShift+6: set the target to Agent No.6.
7. LeftShift+7: set the target to Agent No.7.
8. LeftShift+8: set the target to Agent No.8.
9. LeftShift+9: set the target to Agent No.9.
10. LeftShift+0: Remove the camera target.

Visual Effects Keys

A large number of visual effects can be applied by script (see Chapter 2 of the UDMSlua Manual). Some of these have keyboard shortcuts, as indicated below.

1. RightShift+1: activate Sobel Edge Enhancement.
2. RightShift+2: activate MotionBlur.
3. RightShift+3: activate Negative.

4. RightShift+4: activate Thermal Vision.
5. RightShift+5: activate Posterization.
6. RightShift+6: activate GrayScale.
7. RightShift+7: activate DuoTone.
8. RightShift+8: activate Colorization.
9. RightShift+9: activate Emboss.
10. RightShift+0: deactivate all visual effects.

Chapter 2

The Structure of UDMSlua

2.1 Introduction

UDMSlua is essentially a Lua scripting extension to Unity. The UDMS initials denote *Unity Dance and Motion Scripter*. The main goal of UDMSlua is the easy modeling and visualization of dance behaviors. The Lua binding is effected through xLua.

2.2 The Components of UDMSlua

When UDMSlua is started, the user is presented with a *Main Menu* which allows, as explained in Chapter 1, the selection and execution of a *scenario*. The "minimum" scenario is a folder which contains a `settings.lua`; this, in turn, contains a single function `SetUpFun`.

A scenario (folder) can be expanded by including as many `*.lua` files as you wish. Each file must be connected to some *object* or *object group* in order to be called. An object can be any Unity *gameObject*.

Every `*.lua` file is executed inside a *domain*. This means that the code contained in the file has access to some variables and functions which aim to give the user easy access to both the objects contained in the environment and to most of the Unity functionalities, as well as to both Lua *standard libraries* and to UDMSlua *customized libraries*. We will give more details on domains in Section 2.3.

In addition to the Main Menu, a *console* is also included, which is useful for debugging and executing additional code during runtime. Hitting the ~ key shows/hides the console. When the console is visible, you can enter any of the *console commands* listed in Table 2.1.

It is worth emphasizing that, in addition to the scenaria, the overall application interface can be extended and modified using Lua code in the file `gameSettings.lua`.

2.3 Scenaria and Domains

A *domain* is an environment in which Lua code can be executed. Domains are created to receive objects, inside which symbols (variables / functions) are independent. Hence Lua scripts created for one scenario can also be used, without changes, in a different domain of

basedir	Opens a File Dialog in the main folder.
combine	Selects and runs a script inside a chosen domain.
delete	Deletes the selected domain.
deselect	Deselects a selected environment.
domains	Prints a list of all the room domains.
globalsettings	Reloads (and executes) the global settings.
menu	Opens/closes the Main Menu.
musicdir	Opens a File Dialog in the music folder.
objects	Prints a list of all registered objects of the room.
reload	Reloads (and executes) the selected scenario.
reloaddomain	Reloads (and executes) the selected domain.
room	Opens the scenario selection panel.
run	Runs Lua code directly from the command line.
script	Opens the script selection panel.
select	Selects one of the room domains.
song	Opens the music selection panel.

Table 2.1: Console commands

the same or another scenario. In Figure 2.1 we see inside the blue frames the Lua domains which always exist in the application.

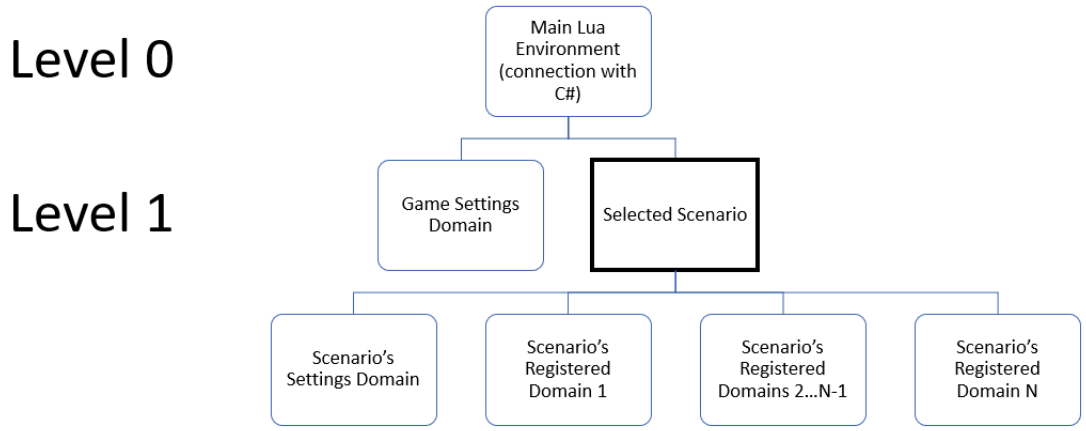


Figure 2.1: The domains of a scenario

The domain levels of Figure 2.1 correspond to the domain meta-tables which have been defined. Domains of Level-1 have access to domain properties of Level-0. This means that, if a symbol has not been defined in the domain of a scenario, it will looked for in the Level-0 domain.

At the lowest level (Level-0), the application, using xLua connects the Unity (C# API) to Lua. In this manner Lua files have access to variables and functions of the C# environment.

At the next level, exist the domains of UDMSlua global settings, as well as the selected scenario. When a scenario is selected a domain is created for its local settings as well as additional domains defined in the scenario.

In most cases, each scenario domain is used by a `*.lua` file. This is not a limitation, because it is also possible to execute any number of files in the same domain. However this choice requires more complex coding (to ensure compatibility between various domain scripts); for this reason it is only offered as a console command.

2.4 Creating Additional Domains

Rather than using a single `*.lua` file for the complete control of a scenario, we have the option of declaring (in the main scenario file) additional domains for particular objects or object groups; to control objects inside a domain, additional `*.lua` files will be executed. We have achieved this by creating special *components* which, when attached to objects make them members of a Lua domain. Depending on the component type, an object can be incorporated in a domain specific to this single object or to one designed to control a group of objects. To every domain corresponds one or more `*.lua` scripts.

Practically speaking, the preferred approach is to create a single separate file and domain for each independent object or for each group of objects. However it is also possible to control any object of the scenario from any script and domain of the scenario.

2.5 Calling Domain Functions

In a video application we usually have a "master loop" in which the application output is specified for each frame. Such a loop indeed exists for all Unity applications and its basic parts are presented in Figure 2.2.

The same loop is applied to the components controlling the execution of Lua code. In these components we have defined all the functions available to the user; each function's main role is to call the corresponding script function defined in the scenario domain. In other words, if a component is added to an object, this component will call the corresponding Lua functions when these are called from C#. Tables 2.2 and 2.3 list functions for objects and object groups.

2.6 Rooms

To select and execute scenaria which are independent of each other we have created *rooms*. A room is for UDMSlua the analog of a scene for Unity; in fact each room is connected to a Unity scene. Unless otherwise declared, this will be a *new* empty scene, i.e., one which does not conatin objects or Lua domains. It is also possible to use pre-constructed scenes (the user can construct such scenes in the Unity editor and add them to the application.

2.7 Lua scripting

UDMSlua domains have some predefined symbols through which they can access the C# environment and, through this, the entire application. As already mentioned in Section 2.3 all

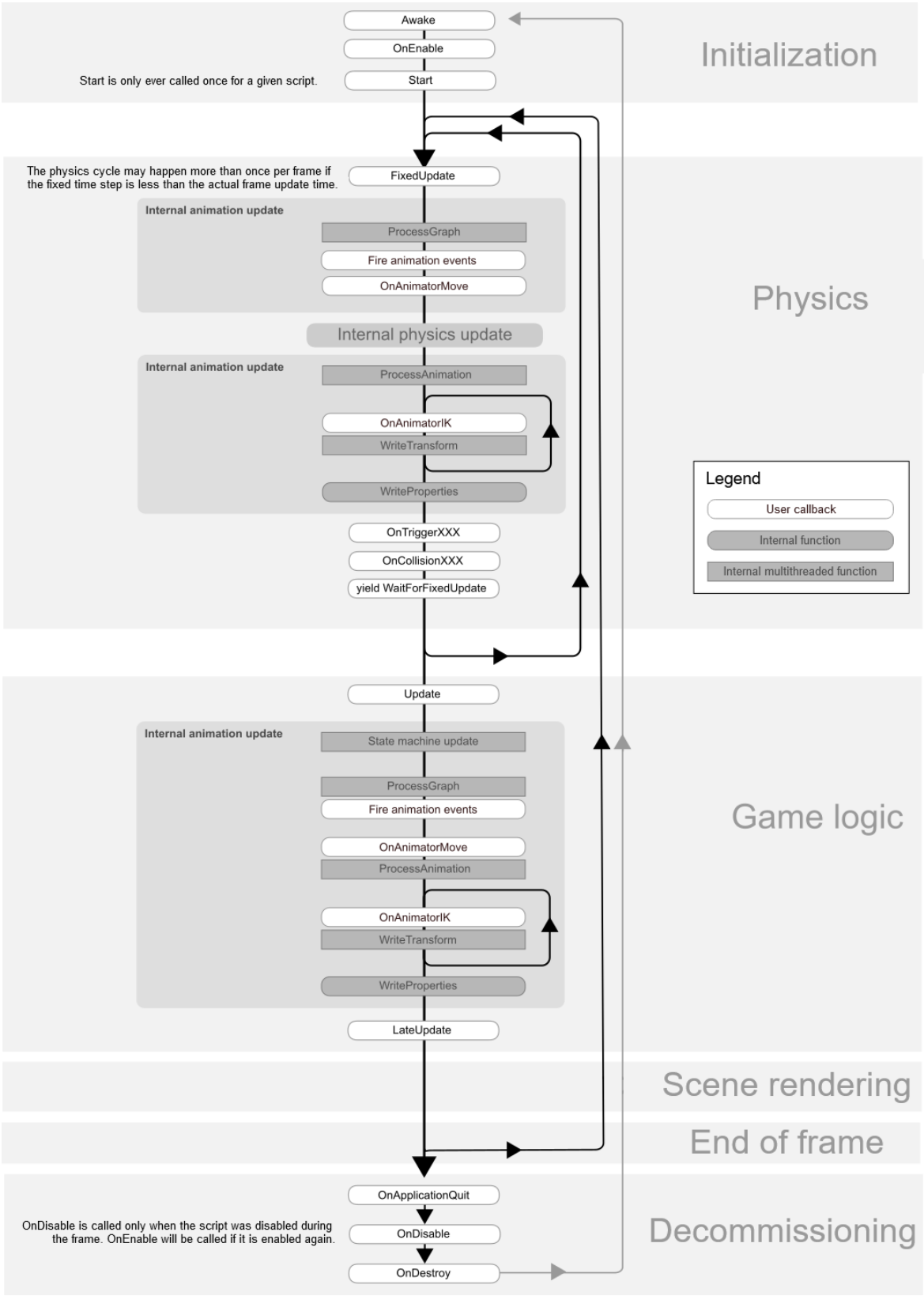


Figure 2.2: Flowchart and function calls in Unity.

Function	Description
awake()	Called when the object is created.
fixedUpdate()	Called in fixed time steps, 50 times per second.
lateUpdate()	Called in every frame after update().
onApplicationQuit()	Called when the application is terminated.
onAnimatorIK(layer)	Called when OnAnimatorIK(layer) is called by Unity.
onAnimatorMove()	Called when OnAnimatorMove() is called by Unity.
onCollisionEnter(col)	Called when OnCollisionEnter(col) is called by Unity.
onCollisionExit(col)	Called when OnCollisionExit(col) is called by Unity.
onCollisionStay(col)	Called when OnCollisionStay(col) is called by Unity.
onDestroy()	Called when the object is destroyed (e.g. when a new scenario is selected).
onDisable()	Called when the object is deactivated.
onEnable()	Called when the object is activated.
onTriggerEnter(col)	Called when OnTriggerEnter(col) is called by Unity.
onTriggerExit(col)	Called when OnTriggerExit(col) is called by Unity.
onTriggerStay(col)	Called when OnTriggerStay(col) is called by Unity.
start()	Called after awake().
update()	Called once per frame.

Table 2.2: Object domain functions.

domains have access to the C#-Lua environment. In practide, this means that the variable `CS` is predefined and ready to use. `CS` contains all the symbols (namespaces, classes, methods, properties etc.) defined in the application to be used in Lua scripts. The most important sumbols are the followsing.

- `CS.UnityEngine`: This contains the classes of Unity (documentation for these is contained online in the Unity documentation site).
- `CS.DG.Tweening`: This is the DemiGiant DOTween package namespace (documentation can be found here.
- `CS.LuaScripting`: This is the application namespace, which contains the classes of objects used in scripts as well as auxiliary methods to simplify the use of *Asset Bundles*.
- `CS.PPEffects`: The namespace of *PostProcessing Effects*.
- `CS.UDMS`: This namespace contains the class *Globals* (the scenario-independent objects of the application scene) and the *IKGameObjects* component which facilitates access to *inverse kinematics* of various humanoid models ("avatars").

In addition to the variable `CS`, domains also contain the variable `Room` (which gives access to the scenario's *LuaRoom* class instance – see Section A) as well as to the following variables.

- `self`: It exists in object domains and refers to the object.

Function	Description
awake()	Called when the group domain is created.
fixedUpdate()	Called in fixed time steps, 50 times per second.
lateUpdate()	Called in every frame after update().
onApplicationQuit()	Called when the application is terminated.
onDestroy()	Called when the domain is dstroyed (e.g., when a new scenario is selected).
onElementAnimatorIK(layer, id)	Called when OnAnimatorIK(layer) is called by Unity for any group member.
onElementAnimatorMove(id)	Called when OnAnimatorMove(layer) is called by Unity for any group member.
onElementCollisionEnter(col,id)	Called when OnCollisionEnter(col) is called by Unityfor any group member.
onElementCollisionExit(col,id)	Called when OnCollisionExit(col) is called by Unity for any group member.
onElementCollisionStay(col,id)	Called when OnCollisionStay(col) is called by Unity for any group member.
onElementTriggerEnter(col,id)	Called when OnTriggerEnter(col) is called by Unity for any group member.
onElementTriggerExit(col,id)	Called when OnTriggerExit(col) is called by Unity for any group member.
onElementTriggerStay(col,id)	Called when OnTriggerStay(col) is called by Unity for any group member.
start()	Called after awake().
update()	Called once per frame.

Table 2.3: Object group domain functions.

- `Group`: It exists in object group domains and refers to the group.
- `Members`: It exists (as an array) in object group domains and refers to all the group members.
- `Settings`: It exists in object domains and gives refers to global variables of the scenario settings.
- `setMusic(path)`: It exists in scenario settings domains and gives access to music files played in runtime.
- `scene`: It exists in scenario settings domains and a predefined scene can be specified through it. The scene must be included in the build and can be specified by passing its name to this property.

Chapter 3

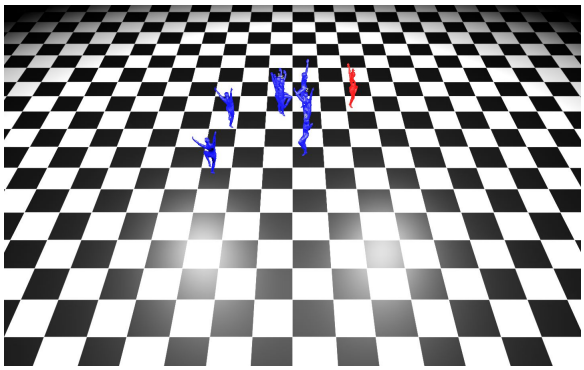
The Scenaria

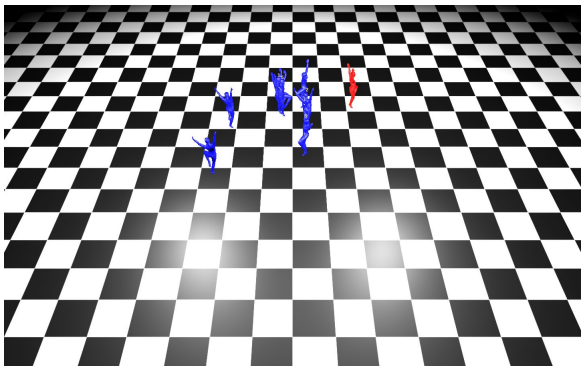
In this chapter we present some of the scenaria included in UDMSlua. In particular we examine in detail the Lua scripts of each scenario. For a better understanding of the scripts you should also refer to Appendices A (UDMSlua API) and B (UDMSlua Libraries).

3.1 A Simple Attraction-Repulsion Scenario

In this section we discuss in high detail a simple attraction - repulsion scenario. The main goal is to present a detailed commentary on the various components of a scenario. **We propose that you read this section quite carefully**; several points covered here will only be treated briefly in later sections.

The idea of this scenario is the following: some agents ("dancers") start in random position and orientation. Each agent's "target" is the "next" agent (i.e., the first agent's target is the second agent, ..., the last agent's target is the first one). Each agent can be in either (a) the "pursue" state, in which he moves towards his target; or (b) the "flee" state, in which he moves away from his target. State transitions depend on the distance between an agent and his target. The scenario starts in a "disordered" state, as seen in the left figure and *always* ends up in an "ordered" steady-state which can be a circular arrangement (as in the right figure), or an 8-shaped arrangement.





You are encouraged to run the scenario (in the actual application) several times and observe its outcome (you can also experiment with changing parameter values); it can be found in the folder

`UDMS_Data/StreamingAssets/Scripts/02_SteeringEtc/STR0002`

which contains three scripts: `settings.lua`, `group.lua` and `camera.lua`.

Before looking at each of these scripts in detail, let us stress that they (and most of the other scripts contained in the UDMSlua distribution) are written using mainly the UDMSlua libraries described in Appendix B. An alternative approach would be to code using the properties / methods approach described in Appendix A. We find the first approach more convenient; in fact the UDMSlua library functions have been written to avoid the use of the "lower-level" properties / methods API. However, all this is a matter of taste; if you prefer properties / methods, you can implement everything that the libraries can do and more. In other words, *everything that can be coded with library functions can also be written using the* UDMSlua API.

3.1.1 The `settings.lua` Script

This is the first script to be executed when the scenario is loaded. It contains the following lines.

```
1  -- aliases --
2  local UE = CS.UnityEngine
3  local VFX = require('effects')
4  local UT = require('utils')
5  local RM = require('functionsROOM');
6  -- vars --
7  local Nagn=8
8  local group
9  -- Everytime the script is reloaded the settings are applied.
10 applySettings();
11 function applySettings()
12       UE.Screen.SetResolution(1440,900, true, 0)
13       UE.QualitySettings.SetQualityLevel(5)
14       UE.Application.targetFrameRate = 30
15       UE.QualitySettings.vSyncCount = 0
16       UE.QualitySettings.shadows = UE.ShadowQuality.All
17 end
18 function setUp()
19       group = RM.addEmptyGroup(Room,'dancers','group.lua')
20       for i=0,Nagn-1 do
21         RM.addGroupMember(Room,'dancers','models/main','NeoMan')
22       end
23       RM.runGroupScript(Room,'dancers')
24       RM.addCamera(Room)
25       setMusic('MusicArchiveOrg/D_SMILEZ_-_07_-
            _Techno_Dance_Club.ogg')
26       VFX.clearAllGlobalEffects()
27 end
```

```
28 function update()
29       VFX.checkGlobalEffectInputs()
30       UT.listenToGenericShortcuts()
31 end
```

Let us consider the parts of the above code.

1. Line 2 declares an alias to the Unity engine namespace, through which we can use most of the Unity engine entities, such as `UE.Vector3`, `UE.Color` etc.
2. Lines 3-5 declare aliases of various UDMSlua libraries. These must either be lua standard libraries or be included in the `Libraries` subfolder of the application.
3. Line 7 declares the local variable `Nagn` (number of agents) and line 8 declares the local variable `group` (this will be the group of agents).
4. Lines 9-10 are self explanatory.
5. Lines 11-17 define the previously called function `applySettings()`. This function sets properties of various Unity engine entities and follows the standard Unity engine API (for details see the Unity online documentation).
6. Lines 18-27 define the function `setUp()`, which is called every time the scenario is loaded. In particular, the following things are done.
   (a) In line 19 the local variable group is created as an empty group. Note the call `RM.addEmptyGroup`; this calls the function `addEmptyGroup` of the `functionsROOM` library script. The function requires a name for the group (`'dancers'`) and a *group script* (`'group.lua'`). More details on this function (and all other UDMSlua library functions) are given in Chapter B.
   (b) In lines 20-22, `Nagn` members are added to the `'dancers'` group. Each member's avatar is `'NeoMan'`, which can be found in the *Asset Bundle* `'models/main'`.
   (c) In line 23 the group script (namely `'group.lua'`) is run. We have to explicitly run the group script as the application isn't aware of the point when the group is ready.
   (d) In line 24 a UDMSlua's main camera is added to the active room (namely `'Room'`) .
   (e) In line 25 we set the music to be played during scenario execution.
   (f) In line 26 we clear all visual effects (those effects are global so some might be activated in a previously run scenario). Note the call `VFX.clearAllGlobalEffects()`, since the function `clearAllGlobalEffects()` is defined in `effects.lua`, alias `VFX`.
7. In lines 28-31 we define the function `update()`, run once per every frame. It checks for visual effects inputs: from script (check the function `checkGlobalEffectInputs()` in `effects.lua` library) and from the keyboard, (`listenToGenericShortcuts()` in `utils.lua` library). These functions poll for the effects and the generic keyboard shortcuts.

3.1.2 The `group.lua` Script

We now discuss the code contained in the `group.lua` script. For ease of presentation, we present and discuss the code in several parts.

The first part of the code declares local variables (some of them are namespaces corresponding to UDMSlua libraries). Note that, in line 5, the `LFG` functions will be called with argument `Group`; this is the particular group to which the script `group.lua` applies. (in this case it is the *only* group of the scenario).

```
1  -- aliases --
2  local UE = CS.UnityEngine
3  local UT = require('utils')
4  local Clips = require('animations')
5  local LF1 = require('functionsGRP'); local LFG = LF1(Group)
6  local LFO = require('functionsOBJ')
7  local ROOM = require('functionsROOM');
8  local LOG = require('logic');
9  local debug = require('debug')
10 -- variables
11 local Nagn = Members.Count
12 local lights = {}
13 local Ground
14 local cube1
15 local sphere1
16 local empty1
17 local T0=100
18 local NormTransDur = 0.35
19 local TIME = 0
20 local d1=0.5
21 local d2=3
22 local dir
23 local dir2
24 local k=1.0
25 local wr=2
```

The second part of the code is the definition of the `start()` function. This is called when the group's creation is complete and the script is run for the first time.

```
26 function start()
27       -- GROUND
28       Ground=LFO.makeObject(Room,'Ground','Ground',"plane",
            UE.Vector3(0, -0.1, 0))
29       LFO.setPos(Ground,UE.Vector3(0,-0.2,0))
30       LFO.setScale(Ground,UE.Vector3(50,1,50))
31       LFO.textureObj(Ground,'textures/ground','checkerboard_2'
            ,50,50)
32       -- AGENTS
```

```
33        LFG.toggleIndices(true)
34        for i=0,Nagn - 1 do
35              LFG.setTurnToMoveDir(i,true)
36              LFG.setColorState(i,false)
37              LFG.setState(i,1)
38              LFG.setPos(i,UE.Vector3(3*math.random(-1,1),0,3*
                   math.random(-1,1)))
39              LFG.aniCrossFade(i,Clips[121],NormTransDur,true)
40              LFG.trailAttach(i,UE.Vector3(0,1,0),UE.Color.red,0,0.01
                   )
41              LFG.trailSetEndColor(i,UE.Color.blue)
42              LFG.trailSetEndWidth(i,0.05)
43        end
44        -- LIGHTS
45        for i=0,1 do
46              lights[i]=LFO.lgtMake(Room,tostring(i),"Light1","spot",
                   UE.Vector3(4*(-1)^i,4,0),UE.Vector3(90,0,0))
47              LFO.lgtSetRange(lights[i],60)
48              LFO.lgtSetIntensity(lights[i],2)
49              LFO.lgtSetSpotAngle(lights[i],155)
50              LFO.lgtSetColor(lights[i],UE.Color(1,1,1))
51        end
52  end
```

The function `start()` consists of three parts.

1. Lines 27-31 create a ground object, set its position and scale and texture it (using texture `checkerboard_2` from Asset Bundle `textures/ground`. Note the usage `LFO.*`, since all the functions used have been defined in `functionsOBJ.lua`, alias `LFO`.

2. Lines 32-43 set up the agents. Note the usage `LFG.*`, since all the functions used have been defined in `functionsGRP.lua`, alias `LFG`. These are *group object* functions, i.e., functions used for objects which are members of a group. The meaning of each function should be obvious from its name; more details on these functions can be found in Chapter B.

3. Similarly, lines 44-51 create two *lights* (more precisely: two object with light components) and define some of their properties.

Next comes the part of `group.lua` which defines the `update()` function. This is called once per frame (it is actually called from the Unity `Update()` function of the LuaRoom component).

```
53  function update()
54        TIME = TIME + 1
55        for i=0,Nagn-1 do
```

```
56              -- STATE
57              local d0=LFG.distToAgent(i,math.fmod(i+1,Nagn))
58              if d0<d1 then LFG.setState(i,2) end
59              if d0>d2 then LFG.setState(i,1) end
60              if LFG.getState(i)==1 then LFG.setColor(i,UE.Color.blue
                   ,0) end
61              if LFG.getState(i)==2 then LFG.setColor(i,UE.Color.red
                   ,0) end
62              -- LOCOMOTION
63              dir=LFG.dirToAgent(i,math.fmod(i+1,Nagn))
64              if LFG.getState(i)==1 then LFG.turnToDirSoft(i,dir,wr)
65              elseif LFG.getState(i)==2 then LFG.turnToDirSoft(i,-dir
                   ,wr)
66              end
67              LFG.moveFwd(i,0.05)
68              -- ANIM
69              if LFG.getStateOld(i)~=LFG.getState(i) then
70                    local s=LFG.getState(i)
71                    if LFG.getState(i)==1 then LFG.aniCrossFade(i,
                         Clips[177],NormTransDur,true) end
72                    if LFG.getState(i)==2 then LFG.aniCrossFade(i,
                         Clips[121],NormTransDur,true) end
73              end
74        end
75  end
```

The function `update()` can be broken down in the following parts.

1. In line 54 the variable `TIME` is incremented. *This is not a true time variable, rather it corresponds to the number of the current frame.*

2. The loop of lines 55-74 is executed once for every agent and can be broken down into three parts.

   (a) Lines 56-61 govern the agent's *state evolution*. The agent can be in two states: state 1 is "pursue" and state 2 is "flee". The agent enters: (a) "flee" state if his distance to the "next" agent is less than `d1`, (b) "pursuit" state if his distance to the "next" agent is greater than `d2` (lines 58-59). The agents are colored blue or red depending on their states (lines 60-61).

   (b) Lines 62-67 govern the agent's *locomotion*. In the "pursue" state the agent turns towards the "next" agent (line 64); in the "flee" state the agent turns away from the "next" agent (line 65). Then the agent moves forward (line 67), i.e., in his current direction.

   (c) Lines 68-73 govern the agent's *animation*, assigning a different animation for each of the two states. Note that the animation names are obtained from the `Clips` array, which was defined in line 4 (by the `animations` library).

The last part of the code implements the function `onElementAnimatorMove`, which corresponds to the Unity `onAnimatorMove()` function.

```
76 function onElementAnimatorMove(i)
77       LFG.aniSetRootMotion(i,true)
78 end
```

3.1.3 The `camera.lua` Script

This script controls the camera behavior. Let us look at its contents divided into parts. The first part contains the usual definitions of namespaces as local variables. Note that, in line 3, the `functionsCAM.lua` library is called with argument `self` which, in the context of the `camera.lua` script is the *Main Camera*.

```
1 local UE = CS.UnityEngine
2 local UT = require('utils')
3 local CAM = require('functionsCAM')(self)
4 local RM = require('functionsROOM')
5 local TIME=0
```

The next two parts of the code mainly use `CAM.*` functions, i.e., those defined in the library `functionsCAM.lua`. The second part contains the camera initialization.

```
6 function start()
7       CAM.setState(1)
8       CAM.setTargetGroup(Room, 'dancers')
9       CAM.setPos(UE.Vector3(4,3,-8))
10      groupDomain = RM.getGroup(Room, 'dancers')
11      CAM.lookAt(groupDomain.Members[0].transform.position +
           UE.Vector3(0, 1, 0))
12      CAM.stateInit()
13 end
```

The above lines can be explained as follows.

1. In line 7 the camera state is set to 1 (free camera).
2. In line 8 the group from which targets (agents) will be selected, is specified to be `dancers`.
3. In line 9 we set the camera (rig) position.
4. In line 10 we get the group domain (the `dancers` group).
5. In line 11 we set the camera to look at `group.Members[0]` (i.e., the first agent).
6. In line 12 we *initialize* the camera state. This step is additional to *setting* the camera state (line 7), uses the settings of lines 8-11 and is usually automatically called when the camera state changes.

```
14 function update()
15       TIME=TIME+1
16       CAM.updateStateFromKeyboard()
17       CAM.updateTargetFromKeyboard()
18       CAM.targetUpdate()
19       CAM.stateUpdate(TIME)
20       UT.printOnScreen(self.State)
21 end
```

The above lines can be explained as follows.

1. In line 15 we increment the frame counter `TIME`.
2. In lines 16-17 we update the camera state and target from keyboard (this has an effect only if the user has actually used the corresponding keyboard shortcuts).
3. In lines 18-19 the camera target and state are actually updated.
4. Line 20 uses a utility function to print on screen the current camera state.

3.2 Epidemic Dances

The scenaria presented in this section implements an *epidemic* dance model. We first give an *intuitive* description of the model, then a *mathematical* one and finally we analyze the scenaria (i.e., collections of `*.lua` scripts) which implement this model.

A word of caution: Section 3.2.2 includes quite a bit of mathematical notation. If you find this hard going, feel free to skip it and proceed directly to the following sections, which analyze the `*.lua` implementation.

3.2.1 Intuitive Description of the Model

Imagine a party in which a group of N agents is present; naturally the agents are expected to dance but they may initially be unwilling to do so (because they are shy, tired or for some other reason). However, if one agent starts dancing, his example may "inspire" additional agents. As more agents start dancing, the inspiration to the non-dancers increases; hence more agents may join in the dance. On the other hand, after a while, some agents have had enough dancing and revert to the non-dancing state.

This situation is similar to the spread of an *epidemic*. We can postulate the existence of a *dance virus* which has the following characteristics.

1. A healthy agent becomes infected at a round in which he has a particular number of infected neighbors.
2. When an agent is infected he dances for $M - 1$ time steps (rounds) and then recovers (becomes healthy).
3. A recovered agent may get reinfected at a later round.

These are the basic assumptions used to develop mathematical models of epidemics for which an extensive literature is available. Note that a key concept of the above model is that each agent has some *neighbors*.

3.2.2 Mathematical Description of the Model

The following describes a *general family* of epidemic dance models, which can be specialized to a particular model by appropriate selection of several components.

An *epidemic dance* is a triple $(G, M, \{x_1, ..., x_N\})$, where $G = (V, E)$ is a *graph* with vertex set $V = \{1, 2, ..., N\}$ and edge set E, M is a positive integer (the number of states) and $x_1, ..., x_N$ are integers between 0 and $M - 1$. The state of the n-th agent (equivalently, vertex of G) is s_n and at every time step it is updated to a new state s'_n as follows:

$$\forall s, n : s'_n = f_n(s_1, ..., s_n) = \begin{cases} 1 & \text{if } s_n = 0 \text{ and } \left(\sum_{m \in N_{(n)}} \mathbf{1}(s_m > 0)\right) \in \{x_1, ..., x_K\}, \\ 0 & \text{if } s_n = 0 \text{ and } \left(\sum_{m \in N_{(n)}} \mathbf{1}(s_m > 0)\right) \notin \{x_1, ..., x_K\}, \\ (m+1)_M & \text{if } s_n = m > 0. \end{cases} \tag{3.1}$$

The notation $(m + 1)_M$ means "$m + 1$ modulo M".

The "epidemic" interpretation is as follows.

1. Time evolves in discrete time steps (rounds).
2. $s_n(t)$ denotes the infection state of the n-th agent at the t-th round: $s_n(t) = 0$ means the agent is healthy and $s_n(t) > 0$ means he is infected.
3. At time $t = 0$, the system starts at some initial infection state $s = (s_1(0), ..., s_N(0))$.
4. At time $t = 1, 2, ...$ the system states are updated by

   $$\forall t, n : s_n(t) = f_n(s_1(t-1), ..., s_N(t-1))$$

   Looking at (3.1) we see that:

   (a) If $s_n(t-1) = 0$ (the n-th agent is healthy at time t) then: if the number of infected *neighboring* agents $\left(\sum_{m \in N_{(n)}} \mathbf{1}(s_m(t-1) > 0)\right)$ belongs to the set $\{x_1, ..., x_K\}$, then the agent gets infected at time t ($s_n(t) = 1$); otherwise he stays healthy ($s_n(t) = 0$)

   (b) If $s_n(t-1) = m > 0$. the agent has been infected for m rounds; then at time t we have $s_n(t) = m + 1$, i.e., the agent has been infected for $m + 1$ rounds, unless $s_n(t-1) = M - 1$, in which case we have $s_n(t) = 0$, i.e the agent is healed after $M - 1$ rounds of illness.

In short, the above framework can be used to model the transmission of the "dance virus" in an agent population. Specific M, $\{x_1, ..., x_K\}$ and G choices result in corresponding specific epidemic dances. We note that the above model is a special gaces of a *generations cellular automaton* (GCA). [7]

3.2.3 Epidemic Dance on a Circle

Let us now look at the `.lua` implementation of an epidemic dance model. It is located in the folder

`UDMS_Data/StreamingAssets/Scripts/11_Epidemic/EPI0001`

The `settings.lua` and `camera.lua` scripts are very similar to the ones discussed in Section 3.1, hence we move directly to `group.lua`, which contains the following code units.

***

The first part of the code includes initializations, aliases etc. The interesting lines are 11-14.

```
1  -- aliases --
2  local UE = CS.UnityEngine
3  local debug = require('debug')
4  local UT = require('utils')
5  local CLP = require('animations')
6  local LF1 = require('functionsGRP'); local LFG = LF1(Group)
7  local LFO = require('functionsOBJ');
8  local ROOM = require('functionsROOM');
9  local LOG = require('logic');
10 local Nagn = Members.Count
11 local gca
12 local nbrs1 = LFG.gcaMakeNbhd('rel1', Nagn, {-3, -2, -1, 0, 1},
     true)
13 local numStates = 5
14 local stateUpdateTime=50
15 local center0 = {}
16 local center1 = {}
17 local lights = {}
18 local R0=7
19 local R1=5
20 local transDur = 0.1
21 local TIME = 0
```

1. In line 11 we define a local variable `gca`, in which we will later store a GCA (generations cellular automaton); this will contain our epidemic model.
2. In line 12 we define the neighborhood of the GCA. The function `gcaMakeNbhd` is part of `functionsGRP.lua` (the group functions library). In this case it is called with the following arguments:
   (a) `'rel1'` specifies that the neighborhood will be specified by *1-d relative offset*.
   (b) `Nagn` is the number of agents.

(c) `{-3, -2, -1, 0, 1}` is the offset required by `'rel1'`. It means that the n-th agent will have as neighbors the agents $n-3, n-2, n-1, n$ (himself) and $n+1$. Note that this is a "relational" neighborhood structure, which does not depend on the actual geometrical locations of the agents.

(d) `true` indicates that the neighborhood will be toroidal, i.e., the neighbors of agent no. 1 are agents no. 2 and no. N.

3. Line 14 specifies that state updates will be performed every 50 frames.

***

The second part of the code defines the `start()` function, executed once, when the scenario is started. The following points are worth noting.

```
22 function start()
23       -- GROUND
24       local Ground=ROOM.getObject(Room, "Ground")
25       LFO.setPos(Ground,UE.Vector3(0,-0.5,0))
26       LFO.setScale(Ground,UE.Vector3(40,1,40))
27       LFO.textureObj(Ground,'textures/ground','grid_1',5,5)
28       -- LIGHTS
29       for i=0,1 do
30             lights[i]=LFO.lgtMake(Room,tostring(i),"Light1","spot",
                 UE.Vector3(4*(-1)^i,4,0),UE.Vector3(90,0,0))
31             LFO.lgtSetRange(lights[i],20)
32             LFO.lgtSetIntensity(lights[i],3)
33             LFO.lgtSetSpotAngle(lights[i],115)
34             LFO.lgtSetColor(lights[i],UE.Color(1,1,1))
35       end
36       -- AGENTS
37       LFG.grpSetNeighbors(nbrs1)
38       gca=LFG.gcaDefine({1, 2, 4}, {}, numStates)
39       for i=0,Nagn - 1 do
40             LFG.setColor(i,UE.Color.red)
41             LFG.trailAttach(i,UE.Vector3(0,0.1,0),UE.Color.red, 5,
                 0.05)
42             LFG.trailSetStartWidth(i,0.01)
43             LFG.trailSetEndWidth(i,0.04)
44             LFG.trailSetStartColor(i,UE.Color.yellow)
45             LFG.trailSetEndColor(i,UE.Color.red)
46             center0[i]=R0*UE.Vector3(math.cos(i*6.28/Nagn),0,
                 math.sin(i*6.28/Nagn))
47             center1[i]=R1*UE.Vector3(math.cos(i*6.28/Nagn),0,
                 math.sin(i*6.28/Nagn))
48             LFG.setState(i,0)
49             LFG.setPos(i,center0[i])
```

```
50              LFG.aniCrossFade(i,CLP[121],transDur,true)
51        end
52        LFG.setState(0,1)
53        LFG.setState(5,1)
54        LFG.toggleIndices(true)
55        onGcaStart()
56  end
```

1. In lines 23-36 we use functions from `functionsOBJ.lua` (the *object* functions library), because we are applying them to "isolated" objects; while in lines 36-55 we use functions from `functionsGRP.lua` (the *group* functions library), because we are applying them to members of a group.

2. In lines 41-45 we attach Trail Renderer to each agent and specify its properties.

3. In lines 46-47 we specify two "center" positions for each agaents. These are the positions towards which each agent will head, depending on his state.

4. In line 48 we set the i-th agent's state to 0, which is the "healthy" (i.e., non-dancing, idle state).

5. In line 49 we set the i-th agent's initial position to `center0[i]`; this is his "rest position".

6. In line 50 we set the i-th agent's initial animaiton to `CLP[121]`; this is an idle animation, which has been previously defined by the library `animations.lua`.

7. In lines 52-53 we set the 0-th agent's and 5-th agent's states to 1; so there exist two infected agents in the group.

8. Finally, in line 55 we invoke `onGcaStart()`, which will perform additional (peripheral) actions when the scenario is started.

***

The third part of the code defines the `update()` function, executed once per frame. The following points are worth noting.

```
57  function update()
58  TIME = TIME + 1
59        -- STATES
60        if TIME % stateUpdateTime == 0 then
61              LFG.gcaUpdate(Group,gca,"type1")
62              onGcaStep()
63        end
64        -- MOVES
65        for i = 0, Nagn-1 do
66              if LFG.getState(i)<=1 then
```

```
67                      if LFG.distAgentToPnt(i,center0[i])>0.1 then
68                            LFG.turnToDir(i,LFG.dirAgentToPnt(i,center0
                                 [i]),10)
69                            LFG.moveFwd(i,0.05)
70                      else
71                            LFG.turnToDir(i,LFG.dirAgentToPnt(i,center1
                                 [i]),5)
72                      end
73                elseif
74                      LFG.turnToDir(i,LFG.dirAgentToPnt(i,center1[i])
                           ,10)
75                      if LFG.distAgentToPnt(i,center1[i])>0.1 then
                           LFG.moveFwd(i,0.05) end
76                end
77          end
78          -- ANIMATION
79          for i = 0, Nagn-1 do
80                col=LFG.getColor(i)
81                s=LFG.getState(i)
82                LFG.setColor(i,UE.Color.Lerp(col,stateToColor(s,colmap)
                     ,0.05))
83                if LFG.getState(i)<=1 then
84                      if LFG.distAgentToPnt(i,center0[i])>0.1 then
85                            LFG.aniCrossFadeDiff(i,CLP[177],0.01,true)
86                      else
87                            LFG.aniCrossFadeDiff(i,CLP[182],0.01,true)
88                      end
89                else
90                      if LFG.distAgentToPnt(i,center1[i])>0.1 then
91                            LFG.aniCrossFadeDiff(i,CLP[121],0.01,true)
92                      else
93                            LFG.aniCrossFadeDiff(i,CLP[109],0.01,true)
94                      end
95                end
96          end
97    end
```

1. In line 58 we increment the frame counter.

2. In lines 60-63 we perform the state update, every `stateUpdateTime=50` frames. The actual state update happens in line 61 and implements the model of equation (3.1); line 62 invokes `onGcaStep()`, which will perform additional (peripheral) actions.

3. Lines 64-77 implement the agents' locomotion, depending on their state. Namely if the i-th agent is

(a) in a state $s \leq 1$, and further from point `center0[i]` than 0.1 units, then he will turn towards `center0[i]` and move forward; otherwise (i.e., when he is sufficiently close to `center0[i]`) he will turns towards `center1[i]` and not move;

(b) in a state $s > 1$, and further from point `center1[i]` than 0.1 units, then he will turn towards `center1[i]` and move forward; otherwise (i.e., when he is sufficiently close to `center1[i]`) he will not move.

4. Lines 78-96 implement the agents' animation. There are two different dancing animations (depending on whether the agent is (a) in dance frenzy and heading for the center or (b) in dance denial and heading for his rest position).

***

The final part of the code is long and tedious but mostly implements "utility operations". The following points are worth noting.

```
 98 function onGcaStart()
 99       local textToWrite = ""
100       for i=0,Nagn-1 do
101             textToWrite = textToWrite..LFG.getState(i)
102             LFG.setColor(i,stateToColor(LFG.getState(i),colmap))
103       end
104       textToWrite = textToWrite..'\n'
105       --UT.printOnScreen(textToWrite)
106       UT.writeText(textToWrite, "test.log", 'w')
107 end
108 function onGcaStep()
109       local textToWrite = ""
110       for i=0,Nagn-1 do
111             textToWrite = textToWrite..Members[i].State
112       end
113       textToWrite = textToWrite..'\n'
114       --UT.printOnScreen(textToWrite)
115       UT.writeText(textToWrite, "test.log", 'w')
116 end
117 function onDestroy()
118       UT.closeAllFiles()
119 end
120 function stateToColor(state)
121       if state==0 then return UE.Color.white end
122       if state==1 then return UE.Color.yellow end
123       if state==2 then return UE.Color.red end
124       if state==3 then return UE.Color.blue end
125       if state==4 then return UE.Color.cyan end
126       if state==5 then return UE.Color.green end
```

```
127         if state==6 then return UE.Color.gray end
128         if state==7 then return UE.Color.magenta end
129         if state==8 then return UE.Color.yellow end
130         if state==9 then return UE.Color.white end
131         return UE.Color.white
132 end
133 function onElementAnimatorMove(i)
134         LFG.aniSetRootMotion(i,true)
135 end
```

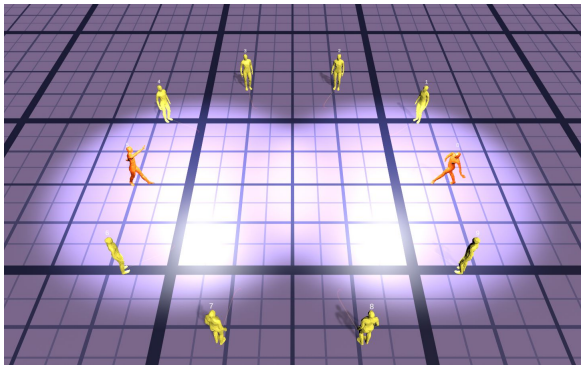
1. Lines 98-107 implement `onGcaStart()` and lines 108-116 implement `onGcaStep()`. In both cases, the main operations are
   (a) write the states of all agents to the file `test.log`;
   (b) color each agent (using the later defined function `stateToColor()`) according to his state;
   (c) an option (`UT.printOnScreen(textToWrite)`) to write the states on the screen has been commeneted out.
2. The function `onDestroy()` defined in lines 117-119 is invoked whenever the scenario is closed; it closes all open files (`UT.closeAllFiles()`) which in this case will be the previously used `test.log`.
3. The previously mentioned function `stateToColor(state)` function, defined in lines 120-132 is used to color each agent according to his state.
4. Finally, the main function of `onElementAnimatorMove(i)`, defined in lines 133-135 is to ensure that the i-th agent's animator controller is set to apply root motion.

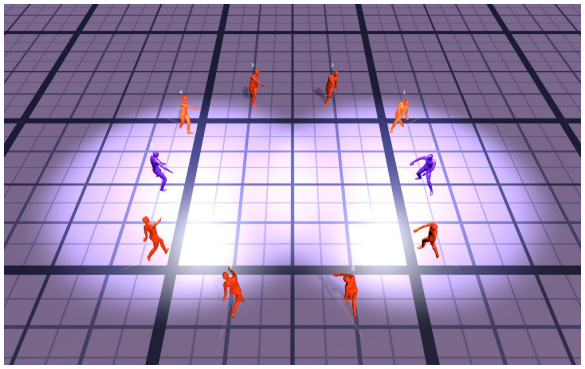
***

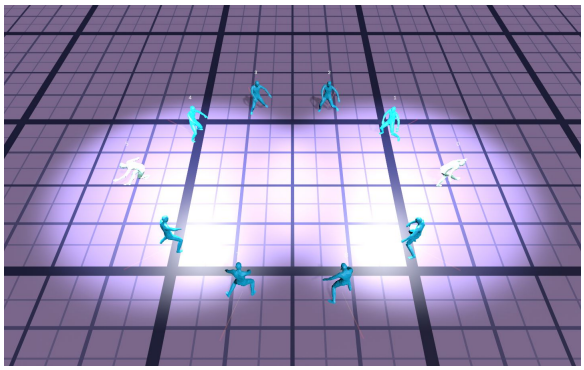
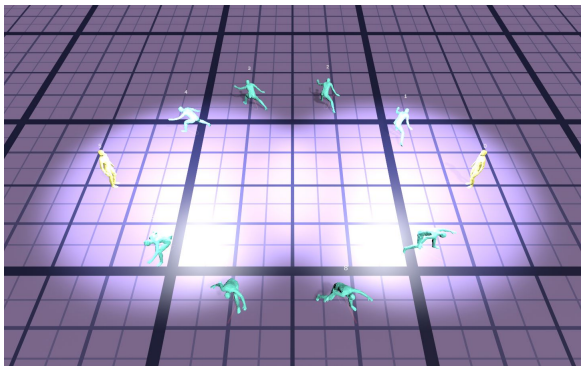
Now is a good time to actually run the scenario. As mentioned, it is located in the folder

`UDMS_Data/StreamingAssets/Scripts/11_Epidemic/EPI0001`

The following figures show some snapshots of the epidemic dance. Colors closer to red (blue) corresponded to a higher (lower) dance infection.





Now you can experiment with this scenario by changing parameters, neighborhood and formation.

3.3 Game Theoretic Dances

The scenario presented in this section implements a *game theoretic* dance model. We first give an intuitive description of the model, then a mathematical one and finally we analyze the scenario (i.e., collection of `*.lua` scripts) which implements this model.

Once again, Section 3.3.2 includes a bit of mathematical notation, which you can skip to proceed directly to the analysis of the `*.lua` implementation.

3.3.1 Intuitive Description of the Model

We now present a family of game theoretic dance models which attempt to describe the following situation. Imagine a party of N agents who can participate in dancing and consequently receive some payoff (for example the enjoyment of dancing); but dancing also has a cost (for instance the risk of being publicly exposed). Hence dancing or not dancing can be understood as a move in a *multi-player game*. What is required is to define this game and, in particular, to define an appropriate *payoff function*. We will do so in the next section.

3.3.2 Mathematical Description of the Model

A *dance game* is a quadruple (G, M, M_0, q), where $G = (V, E)$ is a graph with vertex set $V = \{1, 2, ..., N\}$ and edge set E, M is a positive integer (the number of states), M_0 is a positive integer (the dance threshold) and q is a local payoff function.

The state of the n-th agent (equivalently, vertex of G) is s_n. The n-th agent dances at round t iff $s_n(t) > M_0$. So his dancing behavior over the course of the game depends on the evolution of $s_n(t)$. This holds for any M, but is easier to explain for the case $M = 2$, i.e., we have two states. In this case : the number of dancing neigbors of the n-th agent is

$$x = \sum_{m \in \mathbf{N}(n)} s_m(t-1).$$

Now, at the t-th round the n-th agent assumes his neighbors will act as in the $(t-1)$-th round and compares the following two payoffs

$$Q_1 = q\left(\sum_{m\in\mathbf{N}(n)} s_m + 1\right) - 1 \text{ is the payoff he receives if he dances,}$$

$$Q_2 = q\left(\sum_{m\in\mathbf{N}(n)} s_m\right) \text{ is the payoff he receives if he does not dance.}$$

We use q to denote the "dancing pleasure" and we assume that it is an increasing function (more dancers in the floor means more pleasure). However, after subtracting his personal "dancing cost", it is conceivable that the n-th agent prefers to stay out and get all his pleasure by watching the other dancers. All this is reflected in the comparison of Q_1 and Q_2:

1. If $Q_1 \geq Q_2$ then the agent concludes that it is advantageous for him to dance and hence he increases his state number by 1 (moves into a "more dance-friendly" state).

2. Conversely, if $Q_1 < Q_2$ then the agent concludes that it is advantageous for him not to dance and hence he decreases his state number by 1 (moves into a "less dance-friendly" state).

Consequently, the local (i.e., for each agent) state transition function for $M = 2$ (the two-state agents: dancing and nondancing) is

$$\forall s, n : s_{new} = \begin{cases} 1 & \text{iff} \quad q\left(\sum_{m\in\mathbf{N}(n)} s_m + 1\right) - 1 \geq q\left(\sum_{m\in\mathbf{N}(n)} s_m\right), \\ 0 & \text{iff} \quad q\left(\sum_{m\in\mathbf{N}(n)} s_m + 1\right) - 1 < q\left(\sum_{m\in\mathbf{N}(n)} s_m\right). \end{cases} \tag{3.2}$$

For the more general case $M \geq 2$ we define

$$z_n = \text{"number of neighbors of } n\text{-th agent which have nonzero state"}$$

and then the state transition function is

$$\forall s, n : s_{new} = \begin{cases} \min(s_n + 1, M) & \text{iff} \quad q(z_n + 1) - 1 \geq q(z_n), \\ \max(s_n - 1, 0) & \text{iff} \quad q(z_n + 1) - 1 < q(z_n), \end{cases} \tag{3.3}$$

If we interpret 'dance" as "*cooperate*" and "not dance" as "*defect*", then we can see our dance game as a *multi-player spatial Prisoner's Dilemma* [2, 3, 5].

Furthermore, if you are familiar with *totalistic cellular automata* [4, 7] you probably have recognized that our game evolves by the rules of a totalistic CA. Specifically, consider the totalistic CA with identical birth conditions $\{x_1, x_2, ..., x_K\}$ and survival conditions $\{x_1, x_2, ..., x_K\}$ [1].

[1] We say that the TCA has birth conditions $\{x_1, x_2, ..., x_K\}$ and survival conditions $\{y_1, y_2, ..., y_L\}$ if

1. a cell is activated when it has x active neighbors and x is one of $x_1, x_2, ..., x_K$;

2. and cell survives when it has y active neighbors and y is one of $y_1, y_2, ..., y_L$.

This rule is usually abbreviated as $Bx_1x_2...x_K/Sy_1y_2...y_L$.

Then a cell / agent is active in the t-th round if and only if x of its neighbors were active in the $(t-1)$-th round and x is one of $x_1, x_2, ..., x_K$. This condition can be obtained by requiring that

$$\begin{aligned} &q(x_1+1)-1 \geq q(x_1) \\ &q(x_2+1)-1 \geq q(x_2) \\ &... \\ &q(x_K+1)-1 \geq q(x_K) \end{aligned}$$

(and the inequality is reversed for every x different from $x_1, x_2, ..., x_K$). With a little more effort we can prove the following *theorem*.

Theorem 3.3.1 *Every dance game $(G, 2, 1, q)$ is equivalent to a totalistic CA on the graph G. Conversely, for every totalistic CA on a graph G there exists an increasing function $q(\cdot)$ such that the dance game $(G, 2, 1, q)$ is equivalent to the totalistic CA.*

It follows from Theorem 3.3.1 that we can design dance games with interesting behaviors by looking for interesting behaviors in the well studied family of totalistic CA, and reproduce these as dance games.

3.3.3 Game Theoretic Dance on a Grid

Let us now look at the `.lua` implementation of a game theoretic dance model. It is located in the folder

`UDMS_Data/StreamingAssets/Scripts/12_GamTheoretic/GAM0001`

The `settings.lua` and `camera.lua` scripts are very similar to the ones discussed in Section 3.1, hence we move directly to `group.lua`. We only discuss the lines which introduce some previously unseen functions.

***

The first part of the code includes initializations, aliases etc. The interesting lines are 13-14 and 17-18.

```
 1 local debug = require('debug')
 2 local UE = CS.UnityEngine
 3 local UT = require('utils')
 4 local CLP = require('animations')
 5 local LF1 = require('functionsGRP'); local LFG = LF1(Group)
 6 local LFO = require('functionsOBJ');
 7 local ROOM = require('functionsROOM');
 8 local LOG = require('logic');
 9 local Nagn = Members.Count
10 local nc=math.sqrt(Nagn)
11 local dx=1.5
12 local topLftCor=dx*UE.Vector3(-math.sqrt(Nagn/4),0,math.sqrt(Nagn
     /4))
```

```
13 local form1 = LFG.frmMakeFormation("grid", Nagn, math.sqrt(Nagn),
     topLftCor,dx,dx)
14 local nbrs1 = LFG.gcaMakeNbhd('rel2', Nagn, math.sqrt(Nagn), {{-1,
     0}, {1, 0}, {0, -1}, {0, 1}}, false)
15 local ground
16 local transDur = 0.2
17 local cm=UT.colMakeColorMap("cool")
18 local numStates = 5
19 local gca
20 local threshold
21 local stateUpdateTime=70
22 local TIME = 0
```

1. In line 13 we define a *formation*, i.e., a spatial arrangement to which the agents will be placed. In this case it is characterized by the following parameters.
   (a) `"grid"` obviously means that the agents will be placed on a (two-dimensional) grid.
   (b) `Nagn` is the number of agents.
   (c) The next argument, `math.sqrt(Nagn)` is the number of columns. We are looking here for an orthogonal grid ($n_{\text{col}} = \sqrt{N_{agn}}$).
   (d) The previously defined `topLftCor` are the coordinates of the top left corner of the grid.
   (e) The previously defined `dx` is used as bot the horizontal and vertical spacing between adjacent points of the grid.
2. In line 13 we have defined the spatial formation to be assumed by the agents, but **not** the *neighborhood structure* (i.e., this does not follow automatically from the spatial arrangement) which will be defined next.
3. In line 14 we define a *neighborhood structure*, i.e., the interconnections between the agents (or: which agents are dancing with which). In this case it is characterized by the following parameters.
   (a) `'rel2'` specifies that the neighborhood will be specified by *2-d relative offset*. This matches the previously defined 2d grid,
   (b) `Nagn` is the number of agents.
   (c) The number of columns is, again, `math.sqrt(Nagn)`.
   (d) `{{-1,0},{1,0},{0,-1},{0,1}}` is the offset required by `'rel2'`. It means that the agent at position (i, j) will have as neighbors the agents with positions $(i-1, j)$, $(i+1, j)$, $(i, j-1)$, $(i, j+1)$.
   (e) `false` indicates that the neighborhood will not be toroidal (no wraparound at the borders of the grid).

4. Line 17 specifies that states will be colored according to "cool", one of predefined color schemes.

5. Line 18 specifies that model has $M = 5$ states.

***

The second part of the code defines the start() function, executed once, when the scenario is started. The following points are worth noting.

```
23 function start()
24       --GROUND
25       ground = ROOM.getObject(Room, 'Ground')
26       LFO.setScale(ground,UE.Vector3(40,1,40))
27       LFO.textureObj(ground,'textures/ground','grid_1',50,50)
28       -- AGENTS
29       LFG.grpSetFormation(form1)
30       LFG.grpSetNeighbors(nbrs1)
31       for i=0,Nagn - 1 do
32                   LFG.setRotY(i,180)
33                   LFG.setColor(i,UE.Color.red)
34                   LFG.attachTrail(i, UE.Color.red, 10, 0.05)
35       end
36       for i=nc/2-1,nc/2 do
37             for j=math.floor(nc/2)-1,math.floor(nc/2) do
38                   LFG.setState(i*nc+j,3)
39             end
40       end
41       LFG.setState(math.floor(Nagn/2),3)
42       threshold = 2
43       gca=LFG.gcaDefine({0, 1, 3}, {0, 1, 3}, numStates, threshold)
44       onGcaStart()
45 end
```

1. In lines 29-30 we actually apply the previously defined formation and neighborhood structure to the agents.

2. In lines 36-41 we initialize the states of some agents to 3 (all other, uninitialized states assyme the value 0).

3. In line 42 we set the threshold of the model to $M_0 = 2$.

4. In line 43 we define our dance game. It has "redance" conditions $\{0, 1, 3\}$ (i.e., the birth conditions are $\{0, 1, 3\}$ and the survival conditions are also $\{0, 1, 3\}$), it has numStates states and dance threshold threshold (and, of course, the previously specified neighborhood structure).

***

The third part of the code defines the `update()` function, executed once per frame. The following points are worth noting.

```
46 function update()
47 TIME = TIME + 1
48       -- STATES
49       if TIME % stateUpdateTime == 0 then
50             LFG.gcaUpdate(Group,gca,"type2")
51             onGcaStep()
52       end
53       -- ANIMATION+COLORING
54       for i = 0, Nagn-1 do
55             if LFG.getStateOld(i)~=LFG.getState(i) then
56                   if LFG.getState(i)==0 then LFG.aniCrossFade(i,CLP
                        [187],transDur,true) end
57                   if LFG.getState(i)==1 then LFG.aniCrossFade(i,CLP
                        [ 91],transDur,true) end
58                   if LFG.getState(i)==2 then LFG.aniCrossFade(i,CLP
                        [ 92],transDur,true) end
59                   if LFG.getState(i)==3 then LFG.aniCrossFade(i,CLP
                        [ 94],transDur,true) end
60                   if LFG.getState(i)==4 then LFG.aniCrossFade(i,CLP
                        [ 94],transDur,true) end
61                   if LFG.getState(i)==5 then LFG.aniCrossFade(i,CLP
                        [ 94],transDur,true) end
62             end
63             local col1=LFG.getColor(i)
64             local col2=UT.stateToColor(LFG.getState(i),numStates,cm
                  )
65             local col = UE.Color.Lerp(col1,col2,0.1)
66             LFG.setColor(i,col)
67       end
68 end
```

1. In lines 49-52 we specify that every `stateUpdateTime` frames we perfrom a state update.
2. In lines 63-66 we color the agents according to their state but using a linear interpolation of colors between states.

***

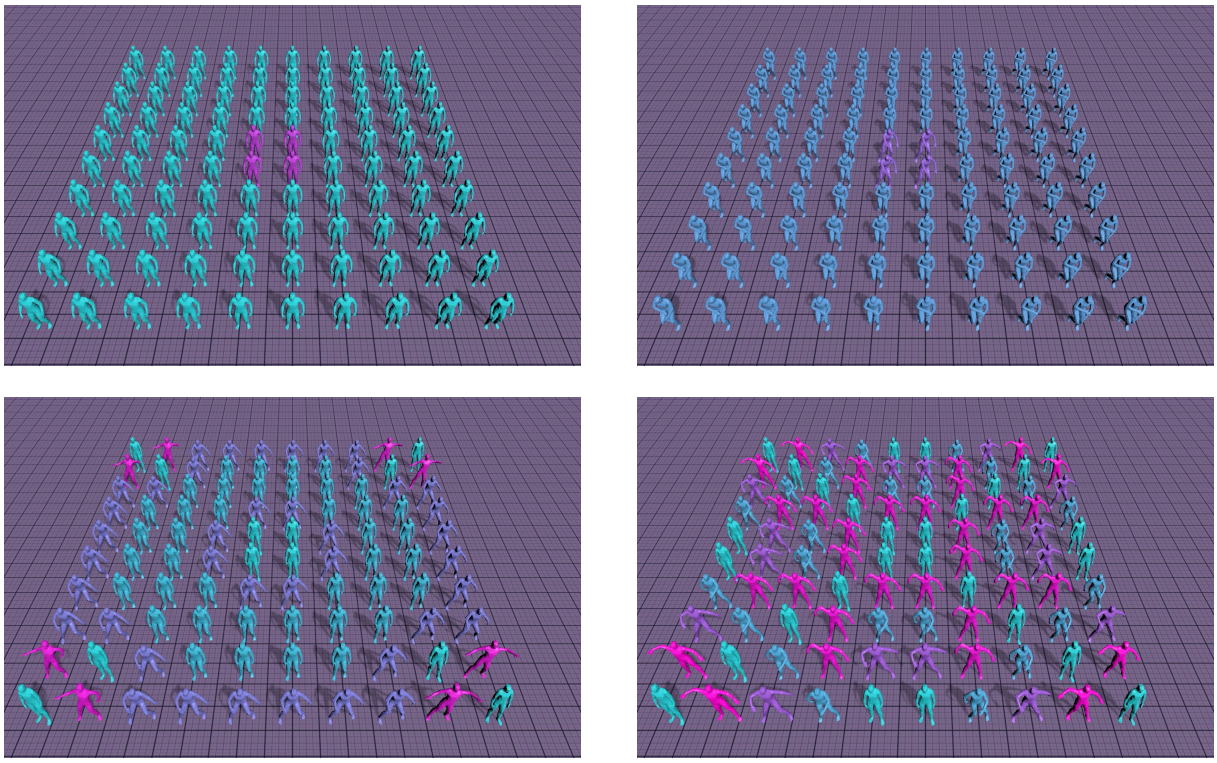
There is some additional code in `group.lua`, calling the functions `onGcaStart()`, `onGcaStep()`, `onDestroy()`, `onElementAnimatorMove(i)`. Since this is very similar to the code used for the epidemic model, we do not discuss it any further.

***

Now let us actually run the scenario. As mentioned, it is located in the folder

```
UDMS_Data/StreamingAssets/Scripts/12_GameTheoretic/GAM0001
```

The following figures show some snapshots of the game theoretic dance. Colors closer to red (blue) corresponded to a higher (lower) dance state.



Now you can experiment with this scenario by changing parameters, neighborhood and formation.

3.4 Logical Dances

3.4.1 Intuitive Description of the Model

Now we introduce a new class of dance models which are based on Boolean and multi-valued logics. The general idea of these models can be summarized as follows.

1. Time evolves in discrete steps (rounds).
2. We have, as usual, a collection of N agents.
3. At each round an agent dances if and only if a "dance condition" is satisfied.

4. Each agent has his own dance condition which is written as a logical function of the dancing states (of some or all agents) in the previous round.

To better understand the model, consider the following example. We have two agents and their dance conditions are as follows.

1. The first agent dances in the current round iff the second agent danced in the previous round.
2. The second agent dances in the current round iff the first agent did not dance in the previous round.

This situation can be described by the following model. For any sentence A, let $\mathbf{Tr}(A)$ denote its truth value which can be **TRUE** or **FALSE**. Also define the dancing states $s_n(t)$ by

$$\forall t \in \mathbb{N}_0, n \in \{1,2\} : s_n(t) = \mathbf{Tr}(\text{“Agent no. } n \text{ is dancing at round } t\text{”})$$

Then the agents' behavior is described by the following equations

$$\begin{aligned} s_1(t) &= s_2(t-1) \\ s_2(t) &= \mathbf{NOT}\ s_1(t-1). \end{aligned}$$

Alternatively, letting $\mathbf{Tr}(\cdot)$ take values in $\{0,1\}$ (with the usual interpretation of 1 as **TRUE** and 0 as **FALSE**) we can describe the situation by the following *numerical* equations

$$\begin{aligned} s_1(t) &= s_2(t-1) \\ s_2(t) &= 1 - s_1(t-1). \end{aligned}$$

So if, for example, the initial dance states are $s_1(0) = 0$, $s_2(0) = 0$, then we have the following state sequence.

t	1	2	3	4	5	6	7	8	9	10
$s_1(t)$	0	0	1	1	0	0	1	1	0	0
$s_2(t)$	0	1	1	0	0	1	1	0	0	1

In Section 3.4.3 we will present a general mathematical formulation of *Logical Dances* which contains the above example as a special case. However, we first need a brief discussion of *multivalued logics* and the corresponding *logical operators*.

3.4.2 Multivalued Logics and Logical Operators

A *Boolean logic* is a tuple $(\{0,1\}, \vee, \wedge, ')$ where $\{0,1\}$ is the set of *truth values* and $\vee, \wedge, '$ are *logical operators* on $\{0,1\}$, defined as follows.

a	b	$a \vee b$	$a \wedge b$	a'
0	0	0	0	1
0	1	1	0	1
1	0	1	0	0
1	1	1	1	0

Table 1

We can take $0, 1$ as undefined symbols and define $\vee, \wedge$ and $'$ by the above table; the semantics are that 0 is "**FALSE**", 1 is "**TRUE**", $\vee$ is "**OR**", $\wedge$ is "**AND**" and $'$ is "**NOT**". However, we can also take a numerical point of view, understanding $0, 1$ as real numbers and defining, for all $a, b \in \{0, 1\}$ the operations as follows

$$\begin{aligned} a \vee b &= \max(a, b), \\ a \wedge b &= \min(a, b), \\ a' &= 1 - a. \end{aligned}$$

It is easily checked that the above equations are equivalent to the definition of Table 1.

We can compute the truth values of logical sentences composed from disjunctions, conjunctions and negations of "primitive" sentences: given primitives $A_1, A_2, ...$ we assume that they have truth values $a_1, a_2, ...$ which belong to $\{0, 1\}$; then

$$\begin{aligned} \mathbf{Tr}(A_1 \ \mathbf{AND} \ A_2) &= a_1 \wedge a_2 = \min(a_1, a_2), \\ \mathbf{Tr}(\mathbf{NOT} \ A_1 \ ) &= a_1' = 1 - a_1, \\ &\text{etc.} \end{aligned}$$

Generalizations of the above are possible and lead to *multi-valued logic* [6]. The general idea is to expand the set of possible truth values and appropriately adjust the logical operators. Many multi-valued logics have been proposed and probably the most popular is Zadeh's "*Fuzzy logic*" [1] $([0, 1], \vee, \wedge, ')$. Here a sentence can take any trith values between 0 and 1 and, for all $a, b \in [0, 1]$ we define

$$\begin{aligned} a \vee b &= \max(a, b), \\ a \wedge b &= \min(a, b), \\ a' &= 1 - a. \end{aligned}$$

So now a sentence can have truth values between fully true (i.e., 1) and fully false (i.e., 0). For example we can have $\mathbf{Tr}(A_1) = 1/2$, indicating that the sentence is neither completely true nor completely false.
A great number of fuzzy logics appear in the literature, synthesized from two basic components.

1. The set of allowable truth values; we will always assume that it is either $[0, 1]$ or a subset of this.

2. The implementation of logical operators; since we assume a numerical truth value set, the logical operators will also be numerical functions.

In fact there exists an infinite number of "appropriate" logical operator functions[1]: the so-called *T-conorms* $S(\cdot)$ generalize **OR**, the *T-norms* $T(\cdot)$ generalize **AND** and the negations

$C(\cdot)$ generalize **NOT**. The following table summarizes some of the most popular t-conorms, t-norms and negations; many other functions satisfy the requirements of the above definitions.

Name	$S(a,b)$	$T(a,b)$	$C(a)$
Standard	$\max(a,b)$	$\min(a,b)$	$1-a$
Algebraic	$a+b-ab$	ab	
Bounded	$\min(1,a+b)$	$\max(0,a+b-1)$	
Drastic	a when $b=0$ b when $a=0$ 1 otherwise	a when $b=1$ b when $a=1$ 0 otherwise	

3.4.3 Mathematical Description of the Model

A *logical dance* is a quadruple $(G,\mathbb{T},M_0,\mathbf{f})$ where $G=(V,E)$ is a graph with vertex set $V=(1,2,...,N)$, $\mathbb{T}$ is a set of truth values, M_0 is a dance threshold and

$$\mathbf{f}=(f_1,...,f_N)$$

is the vector of transition functions such that every f_n is is (the numerical translation of) a logical sentence composed by t-conorm, t-norm and negation operators on $s_1,...,s_N$. The "logical dance" interpretation is as follows.

1. We have N agents whose dance states evolve in discrete time steps (rounds).
2. The dance state of the n-th agent at round t is $s_n(t)\in\mathbb{T}$.
3. The n-th agent performs dances at round t iff $s_n(t)>M_0$; otherwise he stays idel.
4. The state updates are given by

$$\forall t,n: s_n(t+1)=f_n(s_1(t),...,s_N(t))$$

where each $f_n(s_1(t),...,s_N(t))$ is a logical sentence expressing dance conditions.

Here is an example. Let $N=3$ and take the truth value set to be $\mathbb{T}=[0,1]$ (so we are dealing with the "classical" Zadeh fuzzy logic). We will choose the state evolution equations to correspond to the following dance conditions (here we symbolize the agents as P_1, P_2, P_3):

1. [P_1 dances at $t+1$] if and only if [P_1 danced **AND** P_2 did not dance] **OR** [P_2 danced **AND** P_3 did not dance] at t.
2. [P_2 dances at $t+1$] if and only if [P_1 danced **AND** P_2 did not dance] at t.
3. [P_3 dances at $t+1$] if and only if [P_1 danced **AND** P_2 danced] at t.

These are translated as follows.

$$
\begin{aligned}
s_1(t+1) &= S\left(T\left(s_1(t), C\left(s_2(t)\right)\right), T\left(s_2(t), C\left(s_3(t)\right)\right)\right), \\
s_2(t+1) &= T\left(s_1(t), C\left(s_2(t)\right)\right), \\
s_3(t+1) &= T\left(s_1(t), s_2(t)\right).
\end{aligned}
$$

And we will use the following logical operators:

$$
\begin{aligned}
S(a,b) &= a + b - ab, \\
T(a,b) &= ab, \\
C(a) &= 1 - a.
\end{aligned}
$$

Finally, the n-th agent dances at time t iff $s_n(t) > 0.25$ and otherwise stays idle.

3.4.4 Logic Dance on a Circle

Let us now look at the `.lua` implementation of a logic dance. It is located in the folder

```
UDMS_Data/StreamingAssets/Scripts/13_Logic/LOG0012
```

The `settings.lua` and `camera.lua` scripts are very similar to the ones discussed in Section 3.1, hence we move directly to `group.lua`. We only discuss the lines which introduce some previously unseen functions.

***

The first part of the code includes initializations, aliases etc. The interesting lines are 6 and 20-23.

```
 1 local UE = CS.UnityEngine
 2 local UT = require('utils')
 3 local CLP = require('animations')
 4 local LF1 = require('functionsGRP'); local LFG = LF1(Group)
 5 local LFO = require('functionsOBJ');
 6 local LOG = require('logic');
 7 local lights = {}
 8 local Nagn = Members.Count
 9 local transDur = 0.10
10 local cm
11 local d1=2
12 local d2=5
13 local txt1
14 local s = {}
15 local so = {}
16 local tt
17 local Sin
18 local Sout
```

```
19 local ss={}
20 for n=1,2^(Nagn-1) do
21       ss[n] =(n)%(2^(Nagn-1))
22       ss[n+2^(Nagn-1)]=(n)%(2^(Nagn-1)) + 2^(Nagn-1)
23 end
24 local TIME = 0
```

1. In line 6 we invoke the library `logic.lua` (alias `LOG`) which contains implementations of the logical operators, functions for the update of truth states etc.

2. In lines 20-23 we populate the array `ss` which, as will be seen, will be used in the definition of the state transition function.

***

The second part of the code defines the `start()` function, executed once, when the scenario is started. The only noteworthy line is 40, in which we use the previously defined `ss` to define the state transition function; this is specified in terms of two arrays `Sin` and `Sout`: when ever the current state is some element `Sin[k]` then the next state is `Sout[k]`.

```
25 function start()
26       -- LIGHTS
27       for i=0,3 do
28             lights[i]=LFO.lgtMake(Room,tostring(i),"Light1","spot",
                  UE.Vector3(4*math.cos(i*6.28/4),4,4*math.sin(i*6.28
                  /4)),UE.Vector3(90,0,0))
29             LFO.lgtSetRange(lights[i],10)
30             LFO.lgtSetIntensity(lights[i],2)
31             LFO.lgtSetSpotAngle(lights[i],115)
32             LFO.lgtSetColor(lights[i],UE.Color(1,1,1))
33       end
34       -- GROUND
35       Ground=LFO.makeObject(Room,'Ground','Ground',"plane",
            UE.Vector3(0, -0.1, 0))
36       LFO.setPos(Ground,UE.Vector3(0,-0.2,0))
37       LFO.setScale(Ground,UE.Vector3(40,1,40))
38       LFO.textureObj(Ground,'textures/ground','checkerboard_2'
            ,50,50)
39       -- STATE TRANSITION FUNCTION
40       Sin, Sout=UT.makeTransFun(ss,Nagn)
41       -- AGENTS
42       LFG.toggleIndices(true)
43       for i=0,Nagn - 1 do
44             LFG.setTurnToMoveDir(i,true)
45             LFG.setColorState(i,false)
46             s[i]=math.random(0,1)
```

```
47              LFG.setPos(i,UE.Vector3(2*math.cos(i*6.28/Nagn),0,2*
                   math.sin(i*6.28/Nagn)))
48              LFG.trailAttach(i,UE.Vector3(0,1,0),UE.Color.red,30,0
                   .01)
49              LFG.trailSetEndColor(i,UE.Color.blue)
50              LFG.trailSetEndWidth(i,0.05)
51              LFG.trailSetTime(i,0)
52        end
53        cm=UT.colMakeColorMap("cool")
54  end
```

***

The third part of the code defines the `update()` function, executed once per frame. The following points are worth noting.

```
55  function update()
56        TIME = TIME + 1
57        for j=0,Nagn-1 do
58              so[j]=s[j]
59        end
60        if TIME%50==1 then
61              s=LOG.updateStateByNum(so,Sin,Sout,Nagn)
62              for n=0,Nagn-1 do
63                    LFG.setState(n,math.floor(10*s[n]))
64              end
65        end
66        for i=0,Nagn-1 do
67              LFG.setColor(i,UT.stateToColor(math.floor(9*s[i]),10,cm
                   ),1)
68              -- MOVE
69              LFG.turnToDir(i,-LFG.getPos(i),1)
70              -- ANIM
71              if LFG.getStateOld(i)~=LFG.getState(i) then
72                    local s=LFG.getState(i)
73                    if LFG.getState(i)<2 then LFG.aniCrossFade(i,CLP
                         [181],transDur,true) end
74                    if LFG.getState(i)>2 then LFG.aniCrossFade(i,CLP
                         [103],transDur,true) end
75              end
76        end
77  end
```

1. In lines 57-59 we copy the previous state array `s` of the system to the old (previous frame) state array `so`.

2. In lines 60-65 we update the state array `s` of the system

   (a) This state update takes place every 50 frames (line 60).

   (b) First, in line 61 we use the old state `so` and the state transition function (defined by `Sin` and `Sout`) to compute the next state `s`. But this state contains float truth values in the interval $[0, 1]$.

   (c) Hence finally, in lines 62-64 we rescale and round the truth values so that they will be contained in the set $\{0, 1, ..., 9\}$.

***

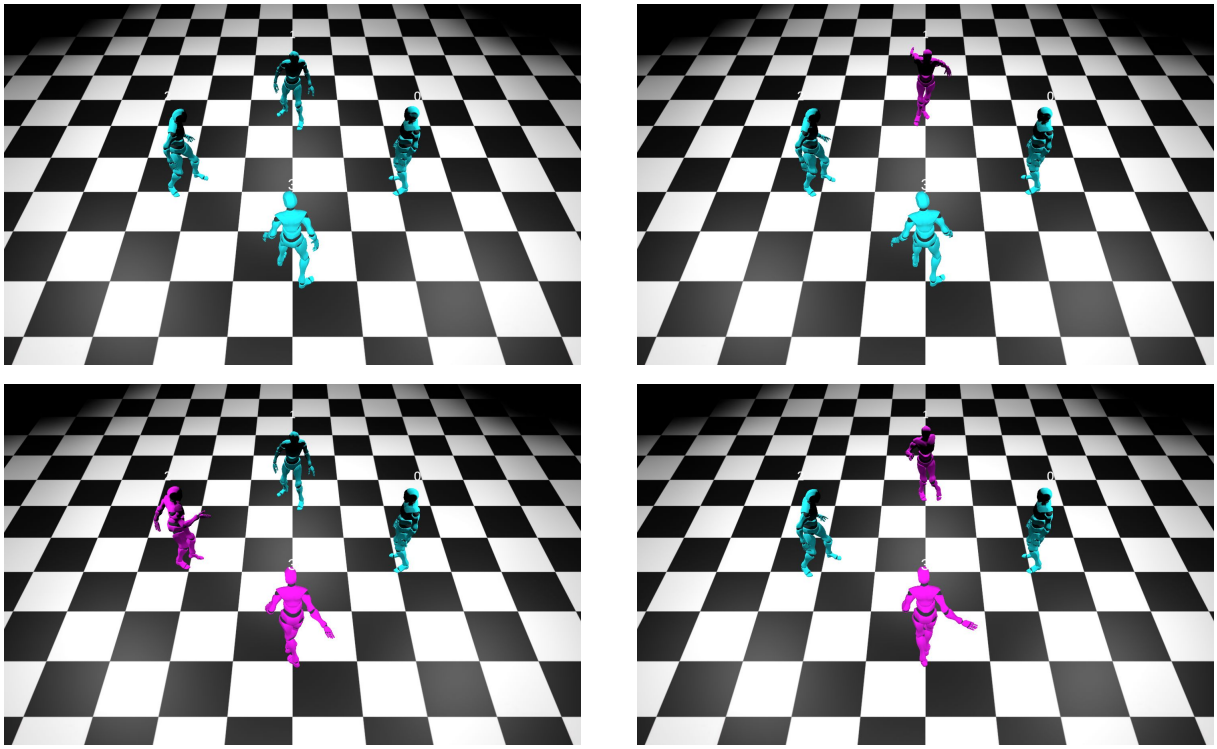
There is some additional code in `group.lua`, calling the functions `onGcaStart()`, `onGcaStep()`, `onDestroy()`, `onElementAnimatorMove(i)`. Since this is very similar to the code used for the epidemic model, we do not discuss it any further.

***

Now let us actually run the scenario. As mentioned, it is located in the folder

```
UDMS_Data/StreamingAssets/Scripts/13_Logic/LOG0012
```

The following figures show some snapshots of the game theoretic dance. Colors closer to red (blue) corresponded to a higher (lower) dance state.



Now you can experiment with this scenario by changing parameters, neighborhood and formation.

Appendix A

UDMSlua API

In this chapter we present the UDMSlua scripting API, and in particular the properties and methods of the basic UDMSlua objects. Some examples can be found in Section A.7; more examples and details can be found here.

A.1 Lua Camera Object

MetaData	Value
class	LuaCameraObject
file	LuaCameraObject.cs

In all the examples in this section it is assumed that `self` is a LuaCameraObject. This practically is the case only inside a `camera.lua` script when the line:

```
Room:InstantiateCameraRig()
```

has been executed for the scenario's room. To access the LuaCameraObject from other scripts you use the following code

```
-- Assuming that InstantiateCameraRig() has been called in the room
    before:
local camerasDomain = Room:GetIndividualDomain('CameraRig')
local luaCameraObject = camerasDomain.LuaIndividualObject
```

and then replace `self` with the luaCameraObject variable in the examples below.

A.1.1 Lua Camera Object Properties

`ActiveCamera: string`
Currently active virtual camera.
Available values: ["explorer", "dolly", "2D", "locked"]. Default value: "explorer"

```
-- Sets the locked camera as active in the scene
-- (modifies the virual cameras' priorities in the backend)
self.ActiveCamera = "locked"
```

AutoDolly: bool
Should the dolly camera be positioned on its path point nearest to the camera's follow target? Default value: true.

```
-- The camera's body will move along the dolly track to be
-- as near as possible to its follow target.
self.AutoDolly = true
```

Cameras: CinemachineVirtualCamera[]
List with all the available virtual cameras.

```
-- The following function activates a virtual camera based on
-- its index inside the Cameras array.
function activateCamera(index)
if index < self.Cameras.Length then
-- The virtual camera with the highest priority becomes
-- the active camera (check Unity's Cinemachine docs for more)
-- https://docs.unity3d.com/Packages/com.unity.cinemachine@2.3/
   manual/CinemachineVirtualCamera.html
for i=0, self.Cameras.Length - 1 do
self.Cameras[i].Priority = 10
end
self.Cameras[index].Priority = 100
end
end
```

DollyPath: string
Dolly camera's path. Available values: ["circular", "square", "custom1"]. Default value: "circular"

```
-- Sets the dolly virtual camera's path to a "circular" one.
self.DollyPath = "circular"
```

FOV: float
The camera's field of view. Default value: 40.

```
-- Sets the camera's field of view to 80.5
self.FOV = 80.5
```

PathPosition: float
Dolly camera's position on its path. Default value: 0. Overwritten if AutoDolly is set to true.

```
-- The camera's body moves to specified position on the track.
-- AutoDolly must be set to false for the change to apply.
self.PathPosition = 0.5
```

A.1.2 Lua Camera Object Methods

DollyPathKeyExists(pathKey: string): bool
Checks if there is a path registered with the provided key.

```
-- Check if path with key custom2 exists; if not create it.
if not self:DollyPathKeyExists('custom2') then
local yPos = 4
-- Circle (smoothed out square).
local points = {UE.Vector3(1, yPos, 1), UE.Vector3(-1, yPos, 1),
   UE.Vector3(-1, yPos, -1), UE.Vector3(1, yPos, -1)}
self:NewSmoothDollyPath('custom2', points)
end
```

FollowGroupAgent(groupName: string, agentId: int, offset: Vector3)
Makes the camera follow a specific group agent.

```
-- Sets target to a group member for the body of the camera.
-- Effect depends on the active camera.
local groupDomainName = 'group'
local agentId = 4
local offset = CS.UnityEngine.Vector3(0, 2, -3)
self:FollowGroupAgent(groupDomainName, agentId, offset)
```

GetDollyPath(pathKey: string): CinemachinePathBase
Returns a cinemachine path from its key or null if there isn't one.

```
-- Find closest point in path to (4, 5, 2) on circular path.
local path = self:GetDollyPath('circular')
local closestPointPathUnits = path:FindClosestPoint(UE.Vector3(4,
   5, 2))
```

LookAtGroupAgent(groupName: string, agentId: int, offset: Vector3)
Makes the camera target a specific group agent. Won't have an effect on a 2D camera.

```
-- Sets camera target to group domain 'group' member with id = 4
local groupDomainName = 'group'
local agentId = 4
local offset = CS.UnityEngine.Vector3(0, 1, 0)
self:LookAtGroupAgent(groupDomainName, agentId, offset)
```

NewDollyPath(pathKey: string, waypoints: Vector3[], looped: bool): Cinemach
Creates a new dolly path for the tracked camera. First and second order continuity of the path isn't guaranteed. Returns the created path component or the path registered to the provided key if the key already exists.

```
-- Check if path with key custom2 exists and if not create it.
if not self:DollyPathKeyExists('custom2') then
local yPos = 4
-- Square.
local points = {UE.Vector3(1, yPos, 1), UE.Vector3(-1, yPos, 1),
   UE.Vector3(-1, yPos, -1), UE.Vector3(1, yPos, -1)}
self:NewDollyPath('custom2', points)
end
```

NewSmoothDollyPath(pathKey: string, waypoints: Vector3[], looped: bool): CinemachinePathBase
Creates a new dolly path for the tracked camera. The path is smooth (first and second order continuity is guaranteed). Returns the created path component or the path registered to the provided key if the key already exists.

```
-- Check if path with key custom2 exists and if not create it.
if not self:DollyPathKeyExists('custom2') then
local yPos = 4
local points = {UE.Vector3(1, yPos, 1), UE.Vector3(-1, yPos, 1),
   UE.Vector3(-1, yPos, -1), UE.Vector3(1, yPos, -1)}
self:NewSmoothDollyPath('custom2', points)
end
```

SetDollyPathPosition(x: float, y: float, z: float)
Sets the position of the dolly camera's current path.

```
-- Moves dolly track's position (not camera's) to (0, 2, 0).
self:SetDollyPathPosition(0, 2, 0)
```

SetDollyPathScale(x: float, y: float, z: float)
Sets the dolly camera's current path's scale. Useful for expanding/compressing the path.

```
-- Moves the dolly track's scale (not the camera's) to (3, 1, 3).
-- Visible result is that dolly track has an expanded radius.
self:SetDollyPathScale(3, 1, 3)
```

SetFollowTarget(targetTransform: Transform, offset: Vector3)
Makes the camera follow the specified transform at an offset. If the transform is null the camera will follow the static point offset.

```
-- Sets the camera's follow target to be (0, 2, -2)
local transform = null
local offset = CS.UnityEngine.Vector3(0, 2, -2)
self:SetFollowTarget(transform, offset)
```

SetLookAtTarget(targetTransform: Transform, offset: Vector3)
Makes the camera look at the specified transform at an offset. If the transform is null the camera will look at the static point offset.

```
-- Sets the camera's look at target to be (0, 1, 0)
local transform = null
local offset = CS.UnityEngine.Vector3(0, 1, 0)
self:SetLookAtTarget(transform, offset)
```

A.2 Lua Game Object

MetaData	Value
class	LuaGameObject
file	LuaGameObjectCustom.cs

A.2.1 Lua Game Object Properties

EulerAnglesOld:Vector3
The rotation of the agent at the start of the frame.

```
-- Print the the rotation the agent 0 had
-- at the start of the frame.
print(Members[0].EulerAnglesOld)
```

PositionOld:Vector3
The position of the agent at the previous frame.

```
-- Print the distance traveled by the agent 0
-- from the previous frame.
print(CS.UnityEngine.Vector3.Distance(Members[0].PositionOld,
   Members[0].transform.position))
```

State:int
The state of the agent. 1 for active, 0 for inactive.

```
-- Set the agent 0's state to active
Members[0].State = 1
```

StateOld:int
The state of the agent at the start of the frame.

```
-- Check if the agent 0's state changed since the last frame.
if Members[0].StateOld ~= Members[0].State then
print('State changed!')
end
```

TurnToMoveDir:bool
If set to true, the agent will rotate to its moving direction.

```
-- Make the agent 0 rotate to its moving direction
Members[0].TurnToMoveDir = true
```

A.2.2 Lua Game Object Methods

DirAgentToPnt(point:Vector3):Vector3
Returns the normalized direction vector from calling Agent to point.

DirMine():Vector3
Returns the direction vector (normalized) of the calling agent
If calling Agent is inside disc of center (0,0,0) and radius d0, it returns calling Agent's direction; otherwise it returns direction from calling agent to (0,0,0).

DirStayInDisc(radius:float):Vector3
If calling Agent is inside disc of center (0,0,0) and radius d0, it returns the calling Agent's direction; otherwise it returns the direction from the calling agent to (0,0,0).

Displacement():Vector3
Returns the vector from calling Agent's position at previous update to the current one.

DistAgentToPnt(point:Vector3):float
Computes the distance of the calling Agent to point.

DistAgentToPntXZ(point:Vector2):float
Computes the distance of the calling Agent to a point on the XZ plane.

DistTravelled():float
Returns the distance travelled by calling Agent between previous and current update.

`GetAngles():Vector3` Returns the Euler angles of calling Agent.

`GetPos():Vector3`
Returns the agent's position.

`GoToPoint(point:Vector3, distance:float)`
Move distance units towards a point.

`GoToPointXZ(point:Vector2, distance:float)`
Move distance units towards a point on the XZ plane.

`IsActive():bool` Return true if the agent's state is 1.

`MoveFwd(distance:float)`
Move forward the specified distance.

`MoveInDir(direction:Vector3, distance:float, normalized:bool)`
Move towards direction distance units. If normalizes is true, the direction will be normalized before it is used.

`MoveInDirXZ(direction:Vector2, distance:float, normalized:bool)`
Move towards a direction on the XZ plane distance units. Should the direction be normalized?

`MoveRight(distance:float)` Move to the right the specified distance.

`MoveUp(distance:float)`
Move up the specified distance.

SetDir(direction:Vector3) Sets the direction of the calling Agent.

SetPos(position:Vector3) Sets the position of the calling Agent.

SetPosX(x:float)
Sets the x coordinate of the position of the calling Agent.

SetPosY(y:float)
Sets the y coordinate of the position of the calling Agent.

SetPosZ(z:float)
Sets the z coordinate of the position of the calling Agent.

SetRot(angles:Vector3) Sets the rotation of the calling Agent in euler angles.

SetRotX(angle:float)
Sets the x euler angle of the rotation of the calling Agent.

SetRotY(angle:float)
Sets the y euler angle of the rotation of the calling Agent.

SetRotZ(angle:float)
Sets the z euler angle of the rotation of the calling Agent.

SetScale(scale:Vector3)
Sets the scale of the calling Agent.

TurnToAngle(targetAngle:float, degrees:float)
Rotate the degrees agent towards the target angle.

TurnToDir(direction:Vector3, speed:float)
Turn towards the specified direction.

TurnToDirSoft(direction:Vector3, speed:float)
Turn towards the specified direction.

A.3 Lua Group Domain

MetaData	Value
class	LuaGroupDomain
file	LuaGroupDomain.cs

A.3.1 Lua Group Domain Properties

Dist: float[][]
A table containing the distance between any two group members at any point.

```
-- Print the distance of the 0 member to all the others
for i = 1, Members.Count - 1 do
print("Distance to member "..i.." is: "..Group.Dist:GetValue(0, i))
end
```

DomainName: string
A name that can be used as a group identifier.

```
-- Print the group's name (the name which was registered
-- for the group along with the script)
print(Group.DomainName)
```

Members: LuaGroupObject[]
The members of the group.

```
-- Print the state of each group member
for i = 0, Group.Members.Count - 1 do
print("Member "..i.."'s state: "..Group.Members[i].State)
end
```

A.3.2 Lua Group Domain Methods

AdaptiveStateUpdate(gca: GCA)
Updates the states of the group members according to the gca argument. Uses the current State and Neighbours of each member.

```
-- Set the gca rules.
local birthConditions = {1, 2, 4}
local surviveConditions = {1, 2, 4}
local numberOfStates = 3
local threshold = 0 -- Defaults to 0 if not provided.
local gca = CS.LuaScripting.GCA(birthConditions, surviveConditions,
    numberOfStates, threshold)
-- Run a gca step with the rules specified in the gca variable:
-- State will become 1 if an agent has 1, 2 or 4 neighbours
-- (birthConditions) with state > gca.threshold
-- Afterwards the state will increment or decrement inside the
-- [0, numberOfStates - 1] based on the surviveConditions.
-- Here, the surviveConditions are the same with the
-- birthConditions, so the state will update the same
-- when it is 0 and otherwise.
Group:AdaptiveStateUpdate(gca)
```

AddMember(): int
Instantiates a prefab from an AssetBundle within the group domain. Returns the new member's id or -1 if the prefab asset wasn't found.

```
function setUp()
-- Create a new empty group in the room
local myGroup = Room:AddGroupDomain('colorfoul', 'group.lua')
-- Add some models, the AddMember function returns
-- their ids inside the group
local xBotId = myGroup:AddMember('xbot', 'models/main')
myGroup:AddMember('ybot', 'models/main')
myGroup:AddMember('MvnPuppet', 'models/main')
-- Run the group script when all the members have been added.
myGroup:DoScript()
end
```

DirOfAgent(agentId: int): Vector3
Returns the direction vector (normalized) of agent Members[agentId].

```
-- Returns the direction of Members[0], equivalent to
-- Members[0]:DirMine()
local dir = Group:DirOfAgent(0)
```

DistOfAgents(agentId1: int, agentId2: int): float
Returns the distance from Members[agentId1] to Agents[agentId2]

```
-- Prints the distance between Members[0] and Members[2]
local dist = Group:DistOfAgents(0, 2)
print(dist)
```

DoScript(insideTheRoom: bool)
Executes lua code from a string inside the domain's environment.

```
function setUp()
-- Create a new empty group in the room
local myGroup = Room:AddGroupDomain('colorfoul', 'group.lua')
-- Add some models, the AddMember function returns
-- their ids inside the group
local xBotId = myGroup:AddMember('xbot', 'models/main')
myGroup:AddMember('ybot', 'models/main')
myGroup:AddMember('MvnPuppet', 'models/main')
-- Run the group script when all the members have been added.
myGroup:DoScript()
end
```

GCAUpdate(gca: GCA)
Updates the states of the group members according to the gca argument. Uses the current State and Neighbours of each member.

```
-- Set the gca rules.
local birthConditions = {1, 2, 4}
local surviveConditions = {}
local numberOfStates = 3
local threshold = 0 -- Defaults to 0 if not provided.
local gca = CS.LuaScripting.GCA(birthConditions, surviveConditions,
    numberOfStates, threshold)
-- Run a gca step with the rules specified in the gca variable:
-- State will become 1 if an agent has 1, 2 or 4 neighbours
-- (birthConditions) with state > gca.threshold
-- Afterwards, the state remains as it is if the
-- surviveConditions apply or circles back to 0.
-- Here, the surviveConditions are empty, so the
-- state of any active agent (state >= 1) will
-- circle back to 0.
Group:GCAUpdate(gca)
```

GetGroupCenter(): Vector3
Returns the mean center position of the group members.

```
-- Move all the agents towards the group center
local groupCenter = Group:GetGroupCenter()
local speed = 0.05
for i = 0, Members.Count - 1 do
Members[i]:GoToPoint(groupCenter, speed)
end
```

GetHoodCenter(center: Vector3, radius: float): Vector3
Finds the center of the members inside the specified circle.

```
-- Move all the agents inside of the circle with center (0, 0)
-- and radius 5.0 towards the center of their hood.
local center = CS.UnityEngine.Vector3(0, 0, 0)
local radius = 5.0
local hoodCenterInCircle = Group:GetHoodCenter(center, radius)
local agentIds = Group:GetMemberIdsInCircle(center, radius)
local speed = 0.05
for k, v in pairs(agentIds) do
Members[v]:GoToPoint(hoodCenterInCircle, speed)
end
```

GetMemberIdsInCircle(center: Vector3, radius: float): int[]
Returns the group members' ids that are inside the specified circle.

```
-- Move all the agents inside of the circle with center (0, 0)
-- and radius 5.0 towards the center of their hood.
local center = CS.UnityEngine.Vector3(0, 0, 0)
local radius = 5.0
local hoodCenterInCircle = Group:GetHoodCenter(center, radius)
local agentIds = Group:GetMemberIdsInCircle(center, radius)
local speed = 0.05
for k, v in pairs(agentIds) do
Members[v]:GoToPoint(hoodCenterInCircle, speed)
end
```

GetNearestAgentWithState(position: Vector3, state: int): int
Returns the id of the nearest to the position agent with state state. Returns -1 if none.

```
-- Prints the id of the nearest agent to the point
-- (0, 0, 0) with an active state (= 1)
local point = CS.UnityEngine.Vector3(0, 0, 0)
local desiredState = 1
local agentId = Group:GetNearestAgentWithState(point, desiredState)
print(agentId)
```

HighlightNeigbours(memberId: int)
Highlights the neighbours of a specific group member. The indices or the selection indicator must be visible to view.

```
-- Highlight the indices and the selection indicator
-- of the neigbours of the member with id = 0
Group:ToggleIndices(true)
Group:HighlightNeigbours(0)
```

InfectionUpdate(infectionInfo: InfectionInfo)
Updates the states of the group members according to the infectionInfo argument. Uses the current State and Neighbours of each member.

```
-- Set the gca rules.
local infectRate = 0.2
local healRate = 0.05
local infectionInfo = CS.LuaScripting.InfectionInfo(infectRate,
   healRate)
-- Run an infection step with the rules specified in the
   infectionInfo variable:
-- If state is 0, it will become 1 with a probability of:
   infectRate*(1 - 1/(alreadyInfectedAgentCount + 1))
-- If state is not 0, it will become 0 with a probability of:
   healRate*(1 - 1/(alreadyInfectedAgentCount + 1)),
-- otherwise it will become 1.
Group:InfectionUpdate(infectionInfo)
```

RegisterGridNeighbours(gridColumns: int)
Sets up all the members' neighbours for a square grid formation. Will reset the Neighbours property of each group member.

```
--[[
If there are 10 Members the resulting grid will be:
0 1 2 positions: (0, 0, 0) (2, 0, 0) (4, 0, 0)
3 4 5 (0, 0,-3) (2, 0,-3) (4, 0,-3)
6 7 8 (0, 0,-6) (2, 0,-6) (4, 0,-6)
9 (0, 0,-9)
--]]
local xDistance = 2.0
local zDistance = 3.0
local columns = 3
local topLeftPoint = CS.UnityEngine.Vector3.zero
Group:ToGridFormation(columns, topLeftPoint, zDistance, xDistance)
-- Display the index of each member
Group:ToggleIndices(true)
-- Register the neighbours of each member
-- (populates the Neighbours property of each member)
Group:RegisterGridNeighbours(columns)
-- Store the neighbours to a local lua table
-- to print them using the table.concat() function
local neighbours = {}
for i = 0, Members[5].Neighbours.Count - 1 do
neighbours[i+1] = Members[5].Neighbours[i]
end
-- Use the concat() from standard lua table library
-- it requires the lua array to start from 1
local table = require('table')
-- prints: 1, 2, 4, 7, 8
print(table.concat(neighbours, ', '))
```

ToGridFormation(columns: int, topLeftPoint: Vector3, rowDistance: float, colDistance: float)

Positions the group members to a grid. The member with id = 0 will be positioned at the top left corner, id 1 will be at the right, etc

```
--[[
If there are 10 Members the resulting grid will be:
ids: 0 1 2 at: (0, 0, 0) (2, 0, 0) (4, 0, 0)
3 4 5 (0, 0,-3) (2, 0,-3) (4, 0,-3)
6 7 8 (0, 0,-6) (2, 0,-6) (4, 0,-6)
9 (0, 0,-9)
--]]
local xDistance = 2.0
local zDistance = 3.0
```

```
local columns = 3
local topLeftPoint = CS.UnityEngine.Vector3.zero
Group:ToGridFormation(columns, topLeftPoint, zDistance, xDistance)
-- Display the index of each member
Group:ToggleIndices(true)
```

ToggleIndices(show: bool)
Shows/Hides the index of each member. Useful for debugging.

```
-- Shows the index of each group member directly
-- on top of the member.
-- The index always faces the camera.
Group:ToggleIndices(true)
```

UpdateStates(algorithm: string, algorithmData: string)
Updates the states of the group members according to a GOL algorithm. Uses the current State and Neighbours of each member.

```
-- Create a grid, register the grid neighbours, set the
-- initial state and run a step of a GOL algorithm.
local xDistance = 2.0
local zDistance = 3.0
local columns = 3
local topLeftPoint = CS.UnityEngine.Vector3.zero
Group:ToGridFormation(columns, topLeftPoint, zDistance, xDistance)
-- Register the neighbours of each member
-- (populates the Neighbours property of each member)
Group:RegisterGridNeighbours(columns)
-- Make the agents 0, 1 and 2 active.
Group:SetState({0, 1, 2})
-- Run a gol step with the rules:
-- Survive if there are 2 active agents in your neighbourhood
-- Be Born (become active) if there are 2 or 3 active agents in
   your neighbourhood
-- Afterwards, the active agents will be: 1, 5, 6, 7 assuming there
    were 10 agents in the group.
Group:UpdateStates('gameoflife', 'S2B23')
```

A.4 Lua Group Object

MetaData	Value
class	LuaGroupObject
file	LuaGroupObject.cs

A.4.1 Lua Group Object Properties

EulerAnglesOld: Vector3
The rotation of the agent at the start of the frame.

```
-- Print the the rotation the agent 0 had at the start of the
   frame.
print(Members[0].EulerAnglesOld)
```

Neighbours: List<int>
The registered neighbours' ids of the group member.

```
-- Print the neighbour ids of the group member with id = 3
for i = 0, Members[3].Neighbours.Count - 1 do
print(Members[3].Neighbours[i])
end
```

PositionOld: Vector3
The position of the agent at the previous frame.

```
-- Print the distance traveled by the agent 0 from the previous
   frame.
print(CS.UnityEngine.Vector3.Distance(Members[0].PositionOld,
   Members[0].transform.position))
```

State: int
The state of the agent. 1 for active, 0 for inactive.

```
-- Set the agent 0's state to active
Members[0].State = 1
```

StateOld: int
The state of the agent at the start of the frame.

```
-- Check if the agent 0's state changed since the last frame.
if Members[0].StateOld ~= Members[0].State then
print('State changed!')
end
```

TurnToMoveDir: bool
If set to true, the agent will rotate to its moving direction.

```
-- Make the agent 0 rotate to its moving direction
Members[0].TurnToMoveDir = true
```

A.4.2 Lua Group Object Methods

DirAgentToPnt(point: Vector3): Vector3
Returns the normalized direction vector from calling Agent to point.

DirAttractRepel(agentId: int, d0: float): Vector3
If the distance between calling Agent and Members[agentId] is less than(greater than) d0, then returns the normalized direction vector from Members[agentId] towards (away) the calling agent.

```
-- Make agent 0 try to keep constant distance 3.0 from agent 1
local dir = Members[0]:DirAttractRepel(1, 3.0)
local speed = 0.05
local normalizeDirection = false
Members[0]:MoveInDir(dir, speed, normalizeDirection)
```

DirAvoidAgent(agentId: int): Vector3
Returns the normalized direction vector from calling agent away Members[agentId]

```
-- Make agent 0 move away from agent 1
local dir = Members[0]:DirAvoidAgent(1)
local speed = 0.05
local normalizeDirection = false
Members[0]:MoveInDir(dir, speed, normalizeDirection)
```

DirAvoidNearestAgent(): Vector3
Returns the normalized direction vector from calling Agent to nearest Agent.

```
-- Make agent 0 move away from the nearest agent
local dir = Members[0]:DirAvoidNearestAgent()
local speed = 0.05
local normalizeDirection = false
Members[0]:MoveInDir(dir, speed, normalizeDirection)
```

DirMine(): Vector3
Returns the direction vector (normalized) of the calling agent

---

DirOfNearest(): Vector3
Returns the normalized direction vector of the Agent nearest to the calling agent.

```
-- Make agent 0 move at the same direction the nearest agent moves
local dir = Members[0]:DirOfNearest()
local speed = 0.05
local normalizeDirection = false
Members[0]:MoveInDir(dir, speed, normalizeDirection)
```

DirStayInDisc(radius: float): Vector3
If calling Agent is inside disc of center (0,0,0) and radius d0, it returns calling Agent's direction; otherwise it returns direction from calling agent to (0,0,0).

DirToAgent(agentId: int): Vector3
Returns the normalized direction vector from calling agent to Members[agentId].

```
-- Make agent 0 move towards agent 4
local dir = Members[0]:DirToAgent(4)
local speed = 0.05
local normalizeDirection = false
Members[0]:MoveInDir(dir, speed, normalizeDirection)
```

DirToHood(radius: float): Vector3
Returns the normalized direction vector from calling agent to the centroid of all agents in distance less than radius to calling Agent

```
-- Make agent 0 move towards the hood center of radius 3
local dir = Members[0]:DirToHood(3)
local speed = 0.05
local normalizeDirection = false
Members[0]:MoveInDir(dir, speed, normalizeDirection)
```

DirToNearest(): Vector3
Returns the normalized direction vector from calling agent to the nearest agent.

```
-- Make agent 0 move towards the nearest agent
local dir = Members[0]:DirToNearest()
local speed = 0.05
local normalizeDirection = false
Members[0]:MoveInDir(dir, speed, normalizeDirection)
```

DirToNearestActive(): Vector3
Returns the normalized direction vector from calling agent to the nearest active agent or Vector(0, 0, 0) if there is none.

```
-- Make agent 0 move towards the nearest active agent
local dir = Members[0]:DirToNearestActive()
local speed = 0.05
local normalizeDirection = false
Members[0]:MoveInDir(dir, speed, normalizeDirection)
```

Displacement(): Vector3
Returns the vector from calling Agent's position at previous update to the current one.

DistAgentToPnt(point: Vector3): float
Computes the distance of the calling Agent to point.

DistAgentToPntXZ(point: Vector2): float
Computes the distance of the calling Agent to a point on the XZ plane.

DistToAgent(agentId: int): float
Computes the distance of calling agent to Members[agentId] agent.

```
-- Prints the distance to the nearest active agent if there is one.
local activeId = Members[0]:GetNearestActive()
if activeId ~= -1 then
local dist = Members[0]:DistToAgent(activeId)
print(dist)
else
print('None of the agents is active.')
end
```

DistToHood(radius: float): float
Computes the distance of the calling Agent to the centroid with radius radius.

```
-- Moves the agent 0 at most 1 meter apart from the hood center.
local radius = 3.0
local distToHood = Members[0]:DistToHood(radius)
if distToHood > 1.0 then
local dir = Members[0]:DirToHood(radius)
local speed = 0.05
local normalizeDirection = false
Members[0]:MoveInDir(dir, speed, normalizeDirection)
end
```

DistToNearest(): float
Computes the distance of calling Agent to its nearest Agent

```
-- Try to move agent 0 at least 4 meters apart from
-- all other agents.
local distToNearest = Members[0]:DistToNearest()
if distToNearest < 4.0 then
local dir = Members[0]:DirAvoidNearestAgent()
local speed = 0.05
local normalizeDirection = false
Members[0]:MoveInDir(dir, speed, normalizeDirection)
end
```

DistToNearestActive(): float
Computes the distance of calling Agent to its nearest active Agent. Will return the max possible float value if there is none.

```
-- Moves the agent 0 at most 1 meter apart from the nearest
-- active agent or not at all if there is no active agent.
local distToActiveAgent = Members[0]:DistToNearestActive()
if distToActiveAgent > 1.0 then
-- dir will be Vector3.zero if there are no active agents,
-- i.e., the agent won't move at all
local dir = Members[0]:DirToNearestActive()
local speed = 0.05
local normalizeDirection = false
Members[0]:MoveInDir(dir, speed, normalizeDirection)
end
```

DistTravelled(): float
Returns the distance travelled by calling Agent between previous and current update.

GetAgentNearest(): int
Returns the index of the agent nearest to calling Agent. Will return the smallest id when there are more than one agents at the same min distance.

```
-- Make agent 0 look towards the nearest agent.
local nearestId = Members[0]:GetAgentNearest()
local dir = Members[0]:DirToAgent(nearestId)
Members[0]:SetDir(dir)
```

GetAngles(): Vector3
Returns the Euler angles of calling Agent.

GetNearestActive(): int
Returns the nearest active (State == 1) agent's id to the calling agent or -1 if there is none.

```
-- Print the id of the nearest active agent to agent 0
local id = Members[0]:GetNearestActive()
print(id)
```

GetHoodCenter(radius: float): Vector3
Returns a vector with the coordinates of the radius-neighborhood of the calling Agent.

```
-- Prints the hood center of the hood of radius 3 the agent 0
-- is part of while moving the agent towards it.
local radius = 3
print(Members[0]:GetHoodCenter(radius))
local dir = Members[0]:DirToHood(radius)
local speed = 0.05
local normalizeDirection = false
Members[0]:MoveInDir(dir, speed, normalizeDirection)
```

GetPos(): Vector3
Returns the agent's position.

GoToPoint(point: Vector3, distance: float)
Move distance units towards a point.

GoToPointXZ(point: Vector2, distance: float)
Move distance units towards a point on the XZ plane.

IsActive(): bool
Return true if the agent's state is 1.

MoveFwd(distance: float)
Move forward the specified distance.

MoveInDir(direction: Vector3, distance: float, normalized: bool)
Move towards direction distance units. Should the direction be normalized?

MoveInDirXZ(direction: Vector2, distance: float, normalized: bool)
Move towards a direction on the XZ plane distance units. Should the direction be normalized?

MoveRight(distance: float)
Move to the right the specified distance.

MoveUp(distance: float)
Move up the specified distance.

SetDir(direction: Vector3)
Sets the direction of the calling Agent.

SetPos(position: Vector3)
Sets the position of the calling Agent.

SetNeighbours(neighbours: int[]): void
Pass an array of indices to set them as neighbours of the group member.

```
-- Set 1 and 2 as neighbours of agent 0.
local neighbours = {1, 2}
Members[0]:SetNeighbours(neighbours)
```

SetPosX(x: float)
Sets the x coordinate of the position of the calling Agent.

SetPosY(y: float)
Sets the y coordinate of the position of the calling Agent.

SetPosZ(z: float)
Sets the z coordinate of the position of the calling Agent.

SetRot(angles: Vector3)
Sets the rotation of the calling Agent in euler angles.

SetRotX(angle: float)
Sets the x euler angle of the rotation of the calling Agent.

SetRotY(angle: float)
Sets the y euler angle of the rotation of the calling Agent.

SetRotZ(angle: float)
Sets the z euler angle of the rotation of the calling Agent.

SetScale(scale: Vector3)
Sets the scale of the calling Agent.

TurnToAngle(targetAngle: float, degrees: float)
Rotate the degrees agent towards the target angle.

TurnToDir(direction: Vector3, speed: float)
Turn towards the specified direction.

TurnToDirSoft(direction: Vector3, speed: float)
Turn towards the specified direction.

A.5 Lua Individual Domain

MetaData	Value
class	LuaIndividualDomain
file	LuaIndividualDomainCustom.cs

A.5.1 Lua Individual Domain Properties

DomainName: string
A name that can be used as a group identifier.

```
-- Print the group's name (the name which was registered for the
   group along with the script)
print(Group.DomainName)
```

A.6 Lua Room

MetaData	Value
class	LuaRoom
file	LuaRoom.cs

A.6.1 Lua Room Properties

Groups: Dictionary<string, LuaGroupDomain>
A map of the groups of the room with their names.

```
-- Print the state of all the members of all the groups
-- in the room.
for groupName, group in pairs(Room.Groups) do
for i = 0, group.Members.Count - 1 do
print('Group "'..groupName..'"'s member '..i..' has state: '
   ..group.Members[i].State)
end
end
```

Objects: Dictionary<string, GameObject>
A map of all the registered objects of the room with their keys.

```
-- Print all the registered simple (not associated with scripts)
-- objects in the room.
for key, gameObject in pairs(Room.Objects) do
print(key..' - '..gameObject.name)
end
```

RegisteredDomains: List<LuaDomain>
A list with all the group and individual domains inside the room. Also contains the room's settings domain.

```
-- Print all the registered domains' names and their types
for i = 0, Room.RegisteredDomains.Count - 1 do
print(Room.RegisteredDomains[i].DomainName..' - '
   ..Room.RegisteredDomains[i]:GetType().Name)
end
```

RoomName: string
The name of the room which is the path of the room's directory inside the Base Path.

```
-- Print the name of the active room.
print(Room.RoomName)
```

RoomScriptPath: string
The full path of the room's directory.

```
-- Print the path of the active room.
print(Room.RoomScriptPath)
```

SceneName: string
The name of the room's scene.

```
-- Print the name of the room's scene.
print(Room.SceneName)
```

A.6.2 Lua Methods

AddGroupDomain(groupName: string, scriptPath: string): LuaGroupDomain
Creates a new group and adds it to the group domain dictionary. Does not run the domain.

```
-- Instantiates 3 game objects from prefabs
-- which exists inside the asset bundle 'models/main' and
-- adds them as members of a new group domain named 'colorfoul'
-- inside the room.
-- The group domain's script path is './group.lua'
-- Create a new empty group in the room
local myGroup = Room:AddGroupDomain('colorfoul', 'group.lua')
-- Add some models, the AddMember function returns their ids inside
    the group
local xBotId = myGroup:AddMember('xbot', 'models/main')
myGroup:AddMember('ybot', 'models/main')
myGroup:AddMember('MvnPuppet', 'models/main')
-- Run the group script when all the members have been added.
myGroup:DoScript()
```

GetGroupDomain(groupName: string): LuaGroupDomain
Retrieves one of the rooms group domains from its name or null if it there is no group with that name.

```
-- Retrieves the group named 'gg' if it exists in the room.
local groupName = 'gg'
local group = Room:GetGroupDomain(groupName)
if group then
print(group.DomainName)
else
print('There is no group named '..groupName)
end
```

GetGroupDomainNames(): string[]
Return a list of the names of the registered group domains in the room.

```
-- Prints the names of all the group domains in the room,
-- then reprints the name of the 1st one if retrieved.
local gnames = Room:GetGroupDomainNames()
for i = 0, gnames.Count - 1 do
print(gnames[i])
end
-- Get the 1st of the group domains
if gnames.Count > 0 then
local domain = Room:GetGroupDomain(gnames[0])
print(domain.DomainName)
end
```

GetIndividualDomain(domainName: string): LuaIndividualDomain
The LuaIndividualDomain or null if no individual domain with that name exists in the room.

```
-- Retrieves the an individual domain named 'io'
-- if it exists in the room.
local domainName = 'io'
local domain = Room:GetIndividualDomain(domainName)
if domain then
print(domain.DomainName)
else
print('There is no individual domain named '..domainName)
end
```

GetIndividualDomainNames(): string[]
Return a list of the names of the registered individual domains in the room

```
-- Prints the names of all the individual domains in the room,
-- then reprints the name of the 1st one if retrieved.
local dnames = Room:GetIndividualDomainNames()
for i = 0, dnames.Count - 1 do
print(dnames[i])
end
-- Get the 1st of the individual domains
if dnames.Count > 0 then
local domain = Room:GetIndividualDomain(dnames[0])
print(domain.DomainName)
end
```

GetObject(objectKey: string): GameObject
Retrieves one of the room's registered objects from its key.

```
-- Retrieves the a registered object with the key 'Light'
-- if it's registered in the room.
local objectName = 'Light'
local gameObject = Room:GetObject(objectName)
if gameObject then
print(gameObject.name)
else
print('There is no registered object named '..objectName)
end
```

GetObjectKeys(): string[]
Return a list of the keys of the available registered objects in the room.

```
-- Prints the key of all the registered simple in the room,
-- then reprints the name of the gameObject of the
-- first one if retrieved.
local objectKeys = Room:GetObjectKeys()
for i = 0, objectKeys.Count - 1 do
print(objectKeys[i])
end
-- Get the name of the gameObject of the 1st of them.
if objectKeys.Count > 0 then
local gameObject = Room:GetObject(objectKeys[0])
print(gameObject.name)
end
```

InstantiateAndRegisterObject(objectKey: string, objectType: string, componen
Type[], activate: bool): GameObject
Instantiates a new registered object in the room. Use for simple objects that do not need their own scripts

```
-- Two ways to create a new object with a light component and
-- register it under the key "myLight"
-- Way 1:
local objectKey = 'myLight'
local components = {typeof(UE.Light)} -- default is null
local activateObject = true -- default is true so it can be omitted
local objectType = '' -- default is '' too, can't be omitted
   components value isn't the default
local myLight = Room:InstantiateAndRegisterObject(objectKey,
   objectType, components, activateObject)
-- Way 2 (as it is a light, the above is equivalent to):
local objectKey = 'myLight'
local objectType = 'light'
local myLight = Room:InstantiateAndRegisterObject(objectKey,
   objectType)
```

InstantiateCameraRig(): GameObject
Instantiates the cinemachine camera rig controlled by room's the camera.lua script.

```
-- usually inside the setUp() function in the room settings:
-- Notice that the room is lowercase inside the settings domain
-- (settings.lua) while uppercase inside the Individual
-- and Group domains.
local camera = Room:InstantiateCameraRig()
-- A Lua Camera Object will now be controllable via the camera.lua
   script of the scenario.
```

InstantiateGroup(objectName: string, bundleName: string, membersCount: int, groupDomainName: string, scriptPath: string): LuaGroupDomain
Instantiates a group of prefabs from an AssetBundle within a group domain inside this room.

```
-- Instantiates 10 game objects from the prefab 'NeoMan'
-- which exists inside the asset bundle 'models/main' and
-- adds them as members of a new group domain named 'group'
-- inside the room.
-- The group domain's script path is './group.lua'
local prefabName = 'NeoMan'
local assetBundle = 'models/main'
local amount = 10
local domainName = 'group'
local scriptRelativePath = './group.lua'
local group = Room:InstantiateGroup(prefabName, assetBundle, amount
   , domainName, scriptRelativePath)
```

InstantiateIndividualGameObject(objectName: string, bundleName: string, objectDomainName: string, scriptPath: string): GameObject
Instantiates a prefab from an AssetBundle with an individual domain inside this room.

```
-- Instantiates a GameObject from prefab 'NeoMan'
-- which exists inside the asset bundle 'models/main' and
-- adds it in a new IndividualDomain named 'neo'
-- inside the room.
-- The individual domain's script path is './ss/group.lua'
-- and is relative to the room's folder.
local prefabName = 'NeoMan'
local assetBundle = 'models/main'
local domainName = 'neo'
local scriptRelativePath = './ss/colorChange.lua'
local neoGameObject = Room:InstantiateIndividualGameObject(
   prefabName, assetBundle, domainName, scriptRelativePath)
```

InstantiateObject(objectType: string, components: Type[], activate: bool): GameObject
Instantiates a new object in the room. Use for simple objects that do not need their own scripts. The object won't be registered in the Room's object.

```
-- Two ways to create a new object with a light component
-- Keep the return value to access the object later.
-- If you need to access this object from other scripts,
-- consider registering the object using the
-- InstantiateAndRegisterObject() or use the
-- RegisterObject() later.
-- Way 1:
local components = {typeof(UE.Light)} -- default is null
local activateObject = true -- default is true so it can be omitted
local objectType = '' -- default is '' too, can't be omitted
   components value isn't the default
local myLight = Room:InstantiateObject(objectType, components,
   activateObject)
-- Way2 (as it is a light, the above is equivalent to):
local objectType = 'light'
local myLight = Room:InstantiateObject(objectType)
```

RegisterObject(objectKey: string, objectToRegister: GameObject)
Registers a GameObject in the room under a key.

```
-- Instantiate and register a light object. The object can
-- later be accessed easily via the room.
-- Equivalent to using the Room:InstantiateAndRegisterObject()
local objectType = 'light'
local myLight = Room:InstantiateObject(objectType)
local objectKey = 'myLight'
Room:RegisterObject(objectKey, myLight)
-- Later, maybe at another scope/script, change the light's color
local light = Room:GetObject('myLight'):GetComponent(typeof(
   CS.UnityEngine.Light))
light.color = CS.UnityEngine.Color(1, 0, 0)
```

Available object Types

The available object types for `InstantiateObject` and `InstantiateAndRegisterObject` functions are:

1. "camera": A game object with a UnityEngine.Camera and a UnityEngine.AudioListener component.
2. "cube": A cube primitive.
3. "cylinder": A cylinder primitive.
4. "light": A game object with a UnityEngine.Light component.
5. "plane": A plane primitive.
6. "quad": A quad primitive.
7. "sphere": A sphere primitive.
8. "vcamera": A game object with a Cinemachine.CinemachineVirtualCamera component.

Any other value will result in the creation of an empty object.

A.7 Scripting Examples

In this section we present some examples of UDMSlua scripting written in the " methods-and-properties" coding style. In Chapter B we will present examples which use the (more convenient) UDMSlua library functions.

A.7.1 Individual object controlling scripts

You can create game objects that are controlled by a lua script in a scenario using the `InstantiateIndividualGameObject()` function of the scenario's room.

For example to create a sphere that changes its color randomly everytime Space is pressed you could do the following:

settings.lua

```
local UE = CS.UnityEngine
-- Use the 'RoomA' scene that already contains a dimensional
-- light (to make colors visible)
scene = 'RoomA'
function setUp()
-- Create a gameObject that will act as the prefab
local spherePrefab = CS.UnityEngine.GameObject.CreatePrimitive(
   CS.UnityEngine.PrimitiveType.Sphere)
-- Instantiate it inside the room binding it to
      -- the 'colorChange.lua' script.
Room:InstantiateIndividualGameObject(spherePrefab, 'sphere', '
   colorChange.lua')
```

```
-- Remove the prefab as it served its purpose.
CS.UnityEngine.Object.Destroy(spherePrefab)
end
```

colorChange.lua

The relative path of this script (relative to the `settings.lua` file) is specified in the function `InstantiateIndividualGameObject()`.

```
-- Shortcut variable, to write 'UE' instead of 'CS.UnityEngine'
local UE = CS.UnityEngine
local renderer
-- awake() is equivalent to Unity's Awake()
-- used for initialization
function awake()
-- Store the renderer of the object in a local
      -- script variable for convenience
renderer = self:GetComponent(typeof(UE.Renderer))
-- Move the object to make it visible to the camera
self.transform.position = UE.Vector3(0, 0, 12)
end
-- update() is called every frame (equivalent to Unity's Update())
function update()
if UE.Input.GetKeyDown(UE.KeyCode.Space) then
renderer.material.color = UE.Color(UE.Random.value, UE.Random.value
   , UE.Random.value)
end
end
```

A.7.2 Group object controlling scripts

You can create a group of the same objects that are controlled by one lua script in a scenario using the `InstantiateGroup()` function of the scenario's room.

In this example we create 10 spheres and change the color of one of them randomly when Space is pressed:

settings.lua

```
local UE = CS.UnityEngine
-- Use the 'RoomA' scene that already contains a
-- directional light (to make colors visible)
scene = 'RoomA'
function setUp()
-- Create a gameObject that will act as the prefab
```

```
local spherePrefab = CS.UnityEngine.GameObject.CreatePrimitive(
   CS.UnityEngine.PrimitiveType.Sphere)
-- Instantiate 10 of them inside the room binding the group
      -- to the 'colorChangeGroup.lua' script.
Room:InstantiateGroup(spherePrefab, 10, 'sphereGroup', '
   colorChangeGroup.lua')
-- Remove the prefab as it served its purpose.
CS.UnityEngine.Object.Destroy(spherePrefab)
end
```

colorChangeGroup.lua

The relative path of this script (relative to the settings.lua file) is specified in the function `InstantiateGroup()`.

```
-- Shortcut variable, to write 'UE' instead of 'CS.UnityEngine'
local UE = CS.UnityEngine
local renderers = {}
function awake()
for i = 0, Members.Count - 1 do
-- Keep the renderers in a local array variable
renderers[i] = Members[i]:GetComponent(typeof(UE.Renderer))
-- Move the objects to make them visible to the camera
Members[i].transform.position = UE.Vector3(i - Members.Count / 2,
   0, 12)
end
end
function update()
if UE.Input.GetKeyDown(UE.KeyCode.Space) then
-- Change the color of a random renderer randomly
-- '//' is the floor division symbol
      -- (keeps the int part of the number)
local id = UE.Random.Range(0, Members.Count - 0.001) // 1
renderers[id].material.color = UE.Color(UE.Random.value,
   UE.Random.value, UE.Random.value)
end
end
```

A.7.3 Using custom prefabs

In the examples above, a GameObject (a sphere primitive) was manually created in-code and then instantiated into the room. However, this is only possible for primitive models as Unity doesn't support runtime model imports by default.

To workaround that, overloaded versions of `InstantiateIndividualGameObject()` and `InstantiateGroup()` are provided that accept asset bundle paths and can load any

prefabs stored in any asset bundle. The following code instantiates a group of 16 game objects from the "NeoMan" prefab of the "models/main" AssetBundle:

```
local group = Room:InstantiateGroup('NeoMan', 'models/main', 16, '
   group', 'group.lua')
```

A.7.4 Create a checker board

```
function setUp()
-- Load the desired texture from the Asset Bundles
local texture = CS.LuaScripting.AssetManager.LoadAsset(typeof(
   UE.Texture), 'checker', 'textures/ground')
-- Create a plane object
local myPlane = Room:InstantiateAndRegisterObject('myGround', '
   plane')
-- Set the plane's texture
myPlane:GetComponent(typeof(UE.Renderer)).material.mainTexture =
   texture
-- Scale the plane and the texture
myPlane.transform.localScale = UE.Vector3(10, 1, 10)
myPlane:GetComponent(typeof(UE.Renderer)).material.mainTextureScale
    = UE.Vector2(10, 10)
-- Add a directional light to make the ground visible
local myLight = Room:InstantiateAndRegisterObject('myLight', 'light
   ')
myLight:GetComponent(typeof(UE.Light)).type =
   UE.LightType.Directional
myLight.transform.eulerAngles = UE.Vector3(45, 0, 0)
end
```

A.7.5 Print text on screen

```
-- Import the "extras" library
local extras = require('extras')
-- Print two lines on screen
local firstLine = 'This is the first line.'
local secondLine = 'This is the second line.'
local posX = 200
local posY = 300
local color = UE.Color.red
extras.printOnScreen(firstLine..'
'..secondLine, posX, posY, color)
```

A.7.6 Play an animation from the available animations

```
-- Import the "animations" library which contains
-- all the available animation names
local animClips = require('animations')
-- Animate agent 0 with the "Armada" animation which is at index 0
local transitionDuration = 0.4
Members[0]:GetComponent(typeof(UE.Animator)):CrossFade(animClips
   [0], transitionDuration)
```

A.7.7 Access a IK limb game object

```
local iks = agent:GetComponent(typeof(CS.UDMS.IKGameObjects))
print(iks.Neck, iks.Head)
print(iks.Spine, iks.Spine1, iks.Spine2, iks.Hips)
print(iks.LeftShoulder, iks.LeftArm, iks.LeftForeArm, iks.LeftHand)
print(iks.RightShoulder, iks.RightArm, iks.RightForeArm,
   iks.RightHand)
print(iks.LeftUpLeg, iks.LeftLeg, iks.LeftFoot)
print(iks.RightUpLeg, iks.RightLeg, iks.RightFoot)
```

A.7.8 Create a group of different objects

```
function setUp()
-- Create a new empty group in the room
local myGroup = Room:AddGroupDomain('colorfoul', 'group.lua')
-- Add some models, the AddMember function returns
      -- their ids inside the group
local xBotId = myGroup:AddMember('xbot', 'models/main')
myGroup:AddMember('ybot', 'models/main')
myGroup:AddMember('MvnPuppet', 'models/main')
-- Run the group script when all the members have been added.
myGroup:DoScript()
end
```

A.7.9 Enable effects shortcuts

```
-- Inside settings.lua
local effects = require('effects')
function update()
effects.checkGlobalEffectInputs()
end
```

A.7.10 Enable a LUT effect with a specific texture

```
local effects = require('effects')
-- Get the global LUT effect
local lutEffect = effects.globalEffect('lut')
-- Set its texture
effects.setLUTEffectTexture(lutEffect, 'Action3D16')
-- Override its enable property to true.
lutEffect.enabled:Override(true)
```

Appendix B

UDMSlua Libraries

The UDMSlua libraries are files in the folder `StreamingAssets/Scripts/Libraries`. They are mainly collections of functions which facilitate many coding tasks which would require a complicated sequence of Methods and Properties calls. Each of the several libraries (i.e., files) contains a group of functions related to a particular task (e.g., Animations, Camera Functions etc.) To use functions from a particular library / file in a script you must include in this script a line such as the following.

```
local UT = require('utils')
local Clips = require('animations')
local LF1 = require('functionsGRP'); local LFG = LF1(Group)
```

Then you can invoke the included functions with the following syntax.

```
LFG.toggleIndices(true)
```

You can add your own functions in an existing library, or create your own library, stored in a file like `MyLibrary.luatart()` which must be placed in the libraries folder.

B.1 Animations

This library is contained in the file `animations.lua`. It defines and returns the `AniClips` array, which contains the names of the agent animation clips. These clips have been downloaded from `Mixamo.com`. The contents of `AniClips` are the following.

```
-- Moves\Capoeira
AniClips[  0]="Armada";
AniClips[  1]="Armada To Esquiva";
AniClips[  2]="Au To Role";
AniClips[  3]="Bencao";
AniClips[  4]="Capoeira00";
AniClips[  5]="Capoeira01";
AniClips[  6]="Capoeira02";
AniClips[  7]="Chapa 2";
AniClips[  8]="Chapa Giratoria 2";
```

```
AniClips[  9]="Chapa-Giratoria";
AniClips[ 10]="Esquiva 1";
AniClips[ 11]="Ginga Variation 1";
AniClips[ 12]="Ginga Variation 2";
AniClips[ 13]="Ginga Variation 3";
AniClips[ 14]="Macaco Side";
AniClips[ 15]="Martelo 2";
AniClips[ 16]="Martelo Do Chau";
AniClips[ 17]="Martelo Do Chau Sem Mao";
AniClips[ 18]="Meia Lua De Compasso";
AniClips[ 19]="Meia Lua De Frente";
AniClips[ 20]="Pontera";
AniClips[ 21]="Troca 1";
-- Moves\Dance\BreakDance
AniClips[ 30]="breakdance_1990_1";
AniClips[ 31]="breakdance_footwork_1";
AniClips[ 32]="breakdance_footwork_2";
AniClips[ 33]="breakdance_footwork_3";
AniClips[ 34]="breakdance_freeze_var_2";
AniClips[ 35]="breakdance_freezes";
AniClips[ 36]="breakdance_ready_2";
AniClips[ 37]="breakdance_uprock";
AniClips[ 38]="breakdance_uprock_var_1";
AniClips[ 39]="breakdance_uprock_var_2";
AniClips[ 40]="brooklyn_uprock";
AniClips[ 41]="flair_1";
AniClips[ 42]="head_spinning_inPlace";
-- Moves\Dance\Dance0
AniClips[ 50]="bellydancing1";
AniClips[ 51]="bellydancing2";
AniClips[ 52]="can_can";
AniClips[ 53]="chicken_dance";
AniClips[ 54]="dancing_0";
AniClips[ 55]="dancing_1";
AniClips[ 56]="dancing_2";
AniClips[ 57]="dancing_3";
AniClips[ 58]="dancing_running_man";
AniClips[ 59]="dancing_twerk";
AniClips[ 60]="flair_1";
AniClips[ 61]="macarena_dance";
AniClips[ 62]="moonwalk";
AniClips[ 63]="rhumba_dancing";
AniClips[ 64]="shuffling";
AniClips[ 65]="silly_dancing_0";
AniClips[ 66]="silly_dancing_1";
```

```
AniClips[ 67]="twist_dance";
AniClips[ 68]="dancing";
AniClips[ 69]="dancing_maraschino_step";
AniClips[ 70]="dancing_running_man";
AniClips[ 71]="macarena_dance";
AniClips[ 72]="ymca_dance";
-- Moves\Dance\DanceWalk
AniClips[ 80]="PeppyLoop";
AniClips[ 81]="PizzazLoop";
AniClips[ 82]="SkipLoop";
AniClips[ 83]="WalkSlideHipHop01";
AniClips[ 84]="WalkSlideHipHop02";
AniClips[ 85]="moonwalk";
-- Moves\Dance\HipHop
AniClips[ 90]="arms_hip_hop_dance";
AniClips[ 91]="bboy_hip_hop_move";
AniClips[ 92]="bboy_hip_hop_move_1";
AniClips[ 93]="booty_hip_hop_dance";
AniClips[ 94]="hip_hop_dancing";
AniClips[ 95]="hip_hop_dancing_1";
AniClips[ 96]="hip_hop_dancing_10";
AniClips[ 97]="hip_hop_dancing_11";
AniClips[ 98]="hip_hop_dancing_12";
AniClips[ 99]="hip_hop_dancing_13";
AniClips[100]="hip_hop_dancing_14";
AniClips[101]="hip_hop_dancing_15";
AniClips[102]="hip_hop_dancing_16";
AniClips[103]="hip_hop_dancing_2";
AniClips[104]="hip_hop_dancing_3";
AniClips[105]="hip_hop_dancing_4";
AniClips[106]="hip_hop_dancing_5";
AniClips[107]="hip_hop_dancing_6";
AniClips[108]="hip_hop_dancing_7";
AniClips[109]="hip_hop_dancing_8";
AniClips[110]="locking_hip_hop_dance";
AniClips[111]="robot_hip_hop_dance";
AniClips[112]="slide_hip_hop_dance";
AniClips[113]="slide_hip_hop_walk";
AniClips[114]="snake_hip_hop_dance";
AniClips[115]="step_hip_hop_dance";
AniClips[116]="tut_hip_hop_dance";
AniClips[117]="tut_hip_hop_dance_1";
AniClips[118]="wave_hip_hop_dance";
AniClips[119]="wave_hip_hop_dance_1";
-- Moves\Dance\Jazz
```

```
AniClips[120]="Jazz Dancing01";
AniClips[121]="Jazz Dancing02";
AniClips[122]="Jazz Dancing03";
AniClips[123]="Jazz Dancing04";
AniClips[124]="Jazz Dancing05";
-- Moves\Dance\NorthernDance
AniClips[130]="NorthernDance1";
AniClips[131]="NorthernDance2";
AniClips[132]="NorthernDance3";
AniClips[133]="NorthernDance4";
AniClips[134]="NorthernDance5";
-- Moves\Dance\Salsa
AniClips[140]="salsa_dancing_0";
AniClips[141]="salsa_dancing_1";
AniClips[142]="salsa_dancing_2";
AniClips[143]="salsa_dancing_3";
AniClips[144]="salsa_dancing_4";
AniClips[145]="salsa_dancing_5";
AniClips[146]="salsa_dancing_6";
AniClips[147]="salsa_dancing_7";
AniClips[148]="salsa_dancing_8";
AniClips[149]="salsa_dancing_9";
-- Moves\Dance\Samba
AniClips[150]="samba_dancing_0";
AniClips[151]="samba_dancing_1";
AniClips[152]="samba_dancing_2";
AniClips[153]="samba_dancing_3";
AniClips[154]="samba_dancing_4";
AniClips[155]="samba_dancing_5";
AniClips[156]="samba_dancing_6";
AniClips[157]="samba_dancing_7";
-- Moves\Dance\Swing
AniClips[160]="swing_dancing_0";
AniClips[161]="swing_dancing_1";
AniClips[162]="swing_dancing_2";
AniClips[163]="swing_dancing_3";
AniClips[164]="swing_dancing_4";
-- Moves\Ninja
AniClips[170]="Crouched Sneaking Left1";
AniClips[171]="Crouched Sneaking Right1";
AniClips[172]="Jump";
AniClips[173]="Ninja Idle0";
AniClips[174]="Ninja Idle1";
AniClips[175]="Ninja Idle2";
AniClips[176]="Ninja Idle3";
```

```
AniClips[177]="Sneak Walk0";
AniClips[178]="Sneak Walk1";
AniClips[179]="Walk Forward Arc";
-- Moves\Idle
AniClips[180]="Idle01";
AniClips[181]="Idle02";
AniClips[182]="Idle03";
AniClips[183]="Idle04";
AniClips[184]="Idle05";
AniClips[185]="Idle06";
AniClips[186]="Idle07";
AniClips[187]="Idle08";
AniClips[188]="Idle09";
AniClips[189]="Idle10";
AniClips[190]="Cover Idle";
AniClips[191]="Crouch Idle01";
AniClips[192]="Kneeling Idle";
AniClips[193]="Neutral Idle";
AniClips[194]="Offensive Idle";
AniClips[195]="Standing Idle01";
AniClips[196]="Standing Idle03";
AniClips[197]="Standing Idle04";
AniClips[198]="Zombie Idle01";
AniClips[199]="Zombie Idle02";
-- Moves\Walk\WalksInPlace
AniClips[200]="walk_inPlace04";
AniClips[201]="walk_inPlace05";
AniClips[202]="walking_inPlace";
AniClips[203]="walking_inPlace01";
AniClips[204]="walking_inPlace02";
AniClips[205]="walking_inPlace03";
AniClips[206]="walking_inPlace06";
AniClips[207]="walking_inPlace07";
AniClips[208]="walking_inPlace08";
AniClips[209]="walking_inPlace23";
AniClips[210]="walking_inPlace25";
AniClips[211]="walking_inPlace26";
AniClips[212]="walking_inPlace27";
AniClips[213]="walking_inPlace_1";
AniClips[214]="walking_inPlace_2";
AniClips[215]="walking_inPlace21";
AniClips[216]="walking_inPlace22";
AniClips[217]="walking_inPlace24";
-- Moves\Walk\Catwalk
AniClips[230]="Catwalk Sequence 01";
```

```
AniClips[231]="Catwalk Sequence 02";
AniClips[232]="Catwalk Sequence 03";
AniClips[233]="Catwalk Walk";
AniClips[234]="Catwalk Walk Forward Crossed";
AniClips[235]="Catwalk Walking";
AniClips[236]="Catwalk Walk Forward HighKnees";
AniClips[237]="Catwalk Walking01";
-- Moves\Walk\WalksCrouch
AniClips[240]="Crouch Walk01";
AniClips[241]="Crouch Walk02";
AniClips[242]="Crouch Walk03";
AniClips[243]="Crouch Walk04";
AniClips[244]="Crouched Walking01";
AniClips[245]="Happy Walk";
AniClips[246]="Cover Idle1";
AniClips[247]="Crouch Walk Back";
AniClips[248]="Crouch Walk Forward";
AniClips[249]="Crouch Walk Left";
AniClips[250]="Crouch Walk Right";
AniClips[251]="Crouched Sneaking Left0";
AniClips[252]="Crouched Sneaking Left1";
AniClips[253]="Crouched Sneaking Right0";
AniClips[254]="Crouched Sneaking Right1";
AniClips[255]="Side StepLeft";
AniClips[256]="Side StepRight";
AniClips[257]="Walk Crouching Left";
AniClips[258]="Walk Crouching Right";
-- Moves\Walk\_MscWalks
AniClips[260]="Strut Walking01";
AniClips[261]="Swagger Walk01";
AniClips[262]="WalkingDrunk01";
AniClips[263]="WalkingDrunk02";
AniClips[264]="WalkingFunny01";
-- Moves\Walk\WalksSneak
AniClips[270]="Sneak Walk01";
AniClips[271]="Sneak Walk02";
AniClips[272]="Sneak Walk03";
AniClips[273]="Sneak Walk04";
AniClips[274]="Sneak Walk05";
AniClips[275]="WalkingStealth01";
-- Moves\Walk\Walks
AniClips[280]="Standard Walk";
AniClips[281]="Walk01";
AniClips[282]="Walking01";
AniClips[283]="Walking02";
```

```
AniClips[284]="Walking03";
AniClips[285]="Walking04";
AniClips[286]="Walking05";
AniClips[287]="Walking06";
-- Moves\Walk\WalksBack
AniClips[290]="Walk Backward Arc";
AniClips[291]="Walk Backwards05";
AniClips[292]="Walking Backwards01";
AniClips[293]="Walking Backwards02";
AniClips[294]="Walking Backwards03";
AniClips[295]="Walking Backwards04";
AniClips[296]="Walking Backwards05";
-- Moves\Walk\WalksStrafe
AniClips[300]="Left Strafe Walk01";
AniClips[301]="Left Strafe Walk02";
AniClips[302]="Right Strafe Walk01";
AniClips[303]="Right Strafe Walk02";
```

B.2 Camera Functions

This library is contained in the file `functionsCAM.lua`. It defines functions which facilitate the manipulation of the Camera object. The defined functions are the following.

B.2.1 Basic Camera Functions

There exist three basic camera functions as described below. Actually the initializing functions are `init01()`, ..., `init06()`, and the updating functions are `update01()`, ..., `update06()`, as trhere exist six pre-defined camera states. Also note that if you define additional camera states you must introduce corresponding `initXY()`, `updateXY()` functions.

```
function init01()
-- initializes the camera to state 1
-- will be invoked every time the camera state changes to 1

setUpDefaultCamera(luaCamera)
-- Passes all the library's functionality to the specified LuaCamera. Using

function update01(TIME:int)
-- updates the camera (position, rotation etc.) when in state 1
-- TIME is not actual time but a counter of the current game frame
```

B.2.2 General Camera Functions

```
getFOV():float
-- returns the camera FOV

getOldState():int
-- gets the camera state of the previous frame

getPos():Vector3
-- returns camera position

getRot():Vector3
-- returns camera Euler angles

getState():int
-- returns the camera state

getTarget():int
-- returns target id

getTargetGroup():string
-- returnsthe target group name

getTargetOffset():Vector3
-- returns the look-at target offset

isCinemachineEnabled():bool
-- returns whether Cinemachine is enabled

lookAt(target:int)
-- makes the camera look at group:Members[target]
-- only effective when Cinemachine is turned off

mainCameraTargetUpdate()
--updates the non-Cinemachine camera target

moveFwd(dist:float)
-- moves the camera forward by dist units
-- only effective when Cinemachine is turned off

moveInDir(dir:Vector3, dist:float)
-- moves the camera dist units in direction dir
-- only effective when Cinemachine is turned off
```

```
moveRight(dist:float)
-- moves the camera to the right  by dist units
-- only effective when Cinemachine is turned off

moveUp(dist:float)
-- moves the camera up  by dist units
-- only effective when Cinemachine is turned off

setFOV(fov):float
-- returns the camera FOV

setPos(pos:Vector3)
-- sets the cmaera position to pos;
-- only effective when Cinemachine is turned off

setRot(eulerAng:Vector3)
-- sets the camera Euler angles
-- only effective when Cinemachine is turned off

setState(state:int)
-- sets the camera state

setTarget(newTargetId:int)
-- sets the camera target to group:Members[newTargetId])

setTargetGroup(room:Room, groupName:string)
-- sets the target group to be groupName
-- assuming groupName exists in Room

setTargetOffset(offset:Vector3)
-- sets the look-at offset for the current target

targetUpdate()
-- updates the camera target

turnToDir(dir:Vector3,speed:float)
-- turns the camera towards direction dir with angular speed speed
-- only effective when Cinemachine is turned off

updateStateFromKeyboard()
-- updates the camera state from the keyboard
-- (predefined keys: F1,...,F6, for states 1 to 6)

updateTargetFromKeyboard(targetsCount:int)
-- updates the camera target from the keyboard
```

```
-- (predefined keys LeftShift+1,...,LeftShift+9.
-- LeftShift+0 removes all camera targets.)

useCinemachine(use:bool)
-- use or not the Cinemachiune functionality
```

B.2.3 CineMachine Functions

These functions require some familiarity with the Unity Cinemachine package. Also, they will only work when Cinemachine is turned on.

```
dollyPathExists(pathKey:string):bool
-- returns whether dolly path with key int exists

getActiveCamera():string
-- returns the active camera name from the predefined virtual cameras

getDollyPath(pathKey:int):CinemachinePathBase
-- returns array of points of path with key int

getPosOnPath():Vector3
-- returns the current position of the camera on
-- the currently active path

newDollyPath(pathName, waypoints, looped)
-- Creates a new dolly path for the tracked camera.
-- First and second order continuity of the path isn't guaranteed.
-- Returns the created path component or the path registered to the provide

newSmoothDollyPath(pathName:string, waypoints:Vector3[], looped:bool)
-- Creates a new dolly path for the tracked camera.
-- The path is smooth (first and second order continuity is guaranteed).
-- Returns the created path component or the path registered to the provide

setActiveCamera(cameraName:string)
-- sets active camera name

setAutoDolly(auto:bool)
-- sets autodolly behavior on or off

setDollyPath(pathName:string)
-- sets the active dolly path to be the one with name

setPosOnPath(pos:Vector3)
```

```
--sets the current position on the path

setPathPos(x:int, y:int, z:int)
--sets the position of the entire path (as a game object)

setPathScale(x:int, y:int, z:int)
-- sets the scale of the current path
```

B.3 Group Functions

This library is contained in the file `functionsGRP.lua`. It contains functions which apply to either an entire group or the groups members.

B.3.1 Animation Functions

These functions are relevant to animations. When the first argument is `i`, the functin will be applied to the i-th group member (agent).

```
setAnim(i:int,anim:string,transDur:float)
-- sets the animation of  group.Members[i] to anim
-- with fade-in time (from previous animation) transDur

aniCrossFade(i:int,anim:string,transDur:float,rel:bool)
-- crossfades the animation of  group.Members[i] to anim
-- with fade-in time (from previous animation) is transDur
-- if rel is true,  transDur is relative time (% of anim duration)
-- if rel is false,  transDur is absolute time

groupFuns.aniGetAnimator(i:int):Animation Controller
-- returns the animation controller of  group.Members[i]

aniGetClipLength(i:int)
-- returns the .clip.length property of group.Members[i]
-- anim. controller

aniGetClipName(i:int):string
-- returns the .clip.Name property of group.Members[i]
-- anim. controller

aniGetClipSpeed(i:int):float
-- returns the .speed property of group.Members[i]
-- anim. controller

aniGetClipSpeedMultiplier(i:int):float
```

```
-- returns the .speedMultiplier property of group.Members[i]
-- anim. controller

aniGetClipTime(i:int,typ:string):float
-- if typ is "rel" returns  the current relative time
-- of group.Members[i] playing animation.
-- if typ is not "rel" returns  the current absolute time

aniGetSpeed(i:int):float
-- returns  the playback speed of group.Members[i]
-- playing animation

aniSetClipSpeed(i:int,v:float)
-- sets the .speed property of group.Members[i]
-- anim. controller to v

aniSetClipSpeedMultiplier(i:int,v:float)
-- sets the .speedMultiplier property of group.Members[i]
-- anim. controller to v

aniSetRootMotion(i:int,rootMotion:bool)
-- sets the .applyRootMotion property of group.Members[i]
-- anim. controller to rootMotion

aniSetStabFeet(i:int,stabFeet:bool)
-- sets the .stabilizeFeet property of group.Members[i]
-- anim. controller to stabFeet
```

B.3.2 Formations

These functions are relevant to formations, i.e., geometric placements of the group members.

```
frmCirclePoints(N:int,center:Vector3,radius:float, angleOffset:float)
-- returns formation as an array of N Vector3 positions
-- the positions are obtained from a circle with:
-- center center, radius radius

frmEllipsePoints(N:int,center:Vector3,a:float,b:float,angleOffset:float)
-- returns formation as an array of N Vector3 positions
-- the positions are obtained from an ellipse with:
-- center center, radii a and b

frmGridPoints(N:int,Nc:int,topLeftPoint:Vector3,rowDistance:float,colDistanc
-- returns formation as an array of Vector3 positions
```

```
-- the positions are obtained from a grid with:
-- N elements and Nc columns
-- with row distance rowDistance and column distance columnDistance

frmKrosePoints(N:int,center:Vector3,a:float,K:int,angleOffset:float):Vector3
-- returns formation as an array of N Vector3 positions
-- the positions are obtained from polar curve rho=a*cos(K*theta)

frmLinePoints(N:int,start:Vector3,step:Vector3)
-- returns formation as an array of N Vector3 positions
-- the positions are obtained from a line with:
-- elements starting at  start and advancing by step

frmLissajousPoints(N:int,ax:float,wx:float,az:float,wz:float,angleOffset:flo
-- returns formation as an array of N Vector3 positions
-- the positions are obtained from parametric curve
-- x(t)=ax * math.cos(wx*t)
-- z(t)=az * math.sin(wz*t)

frmMakeFormation(formation, N, ...)
-- returns a formation of N points
-- possible formation types
-- "circle": calls function  circlePoints
-- "ellipse":  calls function ellipsePoints
-- "line":  calls function linePoints(N,start,step)
-- "grid": :  calls function gridPoints
-- "nrose":  calls function nrosePoints
```

B.3.3 GCA Functions

These functions are relevant to *generalized Cellular Automata*. In other words, they define state updates, neighborhoods etc. For a better understanding of their use see the examples of Sections 3.2 and 3.3.

```
gcaDefine(BirthConds:int[],SurvConds:int[],NrStates:int)
-- defines a GCA (Generalized Cellular Automaton} in which
-- a site is activated if number of active neighbors belongs to BirthConds
-- a site survives if number of active neighbors belongs to SurvConds
-- a site advances to next state if current state is < NrStates
-- GCA behavior also depends on the previously defined neighborhood
-- (see function gcaMakeNbhd)

gcaMakeNbhd(nbhdType:string, N:int, ...)
-- returns a GCA neighborhood
```

```
-- this is a variable input arguments function
-- nbhdType can be
-- "rel1": calls gcaRelativeNeighbours
-- "rel2": calls gcaRelativeGridNeighbours
-- "filePath": calls gcaFilePathNeighbours

gcaFilePathNeighbours(N:int, fpath:string):int[][]
-- returns a neighborhood structure as edge list
-- the edge list is read from file in fpath

gcaRelativeGridNeighbours(N:int, NC:int, relArray2:int[][], wrap:bool)
-- returns a neighborhood structure as edge list
-- the edge list is read from file in fpath
-- returns a neighborhood structure  for a grid
-- with N elements and NC columns
-- the returned neighborhood structure is a corresponding int array
-- which is built by adding to the neighborhood of vertex k
-- the vertices offset by relArray2
-- if wrap is true then the neighborhood structure is toroidal

gcaRelativeNeighbours(N:int, relArray:int[],wrap:bool)
-- returns a neighborhood structure  for a line with N elements
-- the returned neighborhood structure is a corresponding int array
-- which is built by adding to the neighborhood of vertex k
-- the vertices offset by relArray
-- if wrap is true then the neighborhood structure is toroidal
```

B.3.4 Group Functions

This library provides functions to LuaGroup objects, when importing the library pass the group you want for those functions to act on.

Inside the library the 'group' is the parameter and represents a LuaGroup object.

It is essentially a reference of the actual group object that's passed on require()'s returned instance.

```
doScript()
-- runs the (previously assigned) group script

getDist():float[][]
-- returns an array of floats where the (i,j) element
-- contains the distance between the i-th and j-th agent of the group

getDomainName():string
```

```
--  returns the group domain name

getGroupCenter():Vector3
--  return the coordinates of the centroid of the group member positions

getMemberIdsInCircle(center:Vector3,radius:float):int[]
-- returns an array of the indices of all  group members within a circle
of center center and radius radius

getMembers():GameObject[]
-- returns an array of Game Objects, the group members

grpSetFormation(frm:Vector3[])
-- positions the members of the group in the positions contained in frm

grpSetNeighbors(nbr:int[])
-- sets the neighbors of group members to be the ones contained in nbr

grpSetStates(agents:GameObject[])
-- sets the state of all agents to 1

toGridFormation(N1:int,center:Vector3,rowDist:float,colDist:float)
-- sets the positions of group members to a grid with
-- N1 columns
-- centerpoint of the grid at center
-- distance between two row elements rowDist
-- distance between two column elements colDist

toggleIndices(toggle:bool)
-- if toggle is true shows a number above each group members
-- if toggle is false, hides the number
```

B.3.5 Group Member Functions

These functions apply to group *members*. To work correctly it is mandatory to have previously defined a `group` variable. Also, the argument `i` refers to the i-th agent, i.e., to

```
dirAgentToPnt(i:int,point:Vector3):Vector3
-- returns group.Members[i]:DirAgentToPnt(point)

dirAvoidAgent(i:int,j:int):Vector3
-- return the direction from group.Members[j] to group.Members[i]

dirAvoidNearestAgent(i:int):Vector3
```

```
-- returns the direction from group.Members[j] to group.Members[i]
-- where group.Members[j] is the agent closets to group.Members[i]

dirMine(i:int):Vector3
-- returns the Euler Angles (orientation) of group.Members[i]

dirOfAgent(i:int):Vector3
-- returns the Euler Angles (orientation) of group.Members[i]

dirOfNearest(i:int):Vector3
-- returns the Euler Angles (orientation) of group.Members[j]
-- where group.Members[j] is the agent closets to group.Members[i]

dirStayInDisc(i:int,radius:float):Vector3
-- if distance of group.Members[i] to Vector3(0,0,0) is greater than radius,
-- returns the direction from group.Members[i] to Vector3(0,0,0)
-- otherwise returns the direction of group.Members[i]

dirToAgent(i:int,j:int):Vector3
-- returns the direction from group.Members[i] to group.Members[j]

dirToHood(i:int,radius:float):Vector3
-- returns the direction from group.Members[i] to the centroid
-- of all members in a circle with radius radius and
-- center the current position of group.Members[i]

dirToNearest(i:int):Vector3
-- returns the direction from group.Members[i] of group.Members[j]
-- where group.Members[j] is the agent closest to group.Members[i]

distAgentToPnt(i:int,point:Vector3):float
-- returns the distanxcew from group.Members[i] to point point

distOfAgents(i:int,j:int):float
-- returns the distance between group.Members[i]
-- and group.Members[j]

distToAgent(i:int,j:int):float
-- returns the distance between group.Members[i]
-- and group.Members[j]

distToHood(i:int,radius):float
-- returns the distance from group.Members[i] to the centroid
-- of all members in a circle with radius radius and
-- center the current position of group.Members[i]
```

```
distToNearest(i:int):float
-- returns the distance from group.Members[i] to  group.Members[j]
-- where group.Members[j] is the agent closest to group.Members[i]

distTravelled(i:int):float
-- returns the distance travelled by group.Members[i]
-- between the last and current frame

getAllComponents(i:int):Component[]
-- returns an array of all components of  group.Members[i]

getAgentNearest(i:int):GameObject
-- returns the agent closest to group.Members[i]

getColor(i:int,j:int):Color
-- returns the .color of the j-th renderer of group.Members[i]

getDisplacement(i:int):Vector3
-- returns the displacemnt of group.Members[i]
-- between the previous and current frame

getEulerAngles(i:int):Vector3
-- returns the Euler Angles of group.Members[i]

getEulerAnglesOld(i:int):Vector3
-- returns the previous frame Euler Angles of group.Members[i]

getNearestActive(i:int):GameObject
-- returns the closest agent to group.Members[i]
-- which has state gretare than 0

getNeighbours(i:int):GameObject[]
-- returns an array of agents who are neighbors of group.Members[i]

getPosition(i:int):Vector3
-- returns the position of group.Members[i]

getPos(i:int):Vector3
-- returns the position of group.Members[i]

getPosOld(i:int):Vector3
-- returns the previous frame position of group.Members[i]

getRot(i:int):Vector3
```

```
-- returns the Euler Angles of group.Members[i]

getState(i:int):int
-- returns the state of group.Members[i]

getStateOld(i:int):int
-- returns the previous frame state of group.Members[i]

goToAgent(i:int,j:int,dist:float)
-- moves group.Members[i] a distance dst towards group.Members[j]

goToCenter(i:int,dist:float)
-- moves group.Members[i] a distance dist towards Vector3(0,0,0)

goToPoint(i:int,point:Vector3,dist:float)
-- moves group.Members[i] a distance dist towards point

highlightNeigbours(i:int)
-- highlights the neighbors of group.Members[i]

isActive(i:int):bool
-- returns true if group.Members[i] ha sstate greater than 0
-- and false otherwise

moveFwd(i:int,dist:float)
-- moves group.Members[i] a distance dist forward

moveInDir(i:int,dir:Vector3,dist:float,norm:bool)
-- if norm==true, moves group.Members[i] dist in direction dir
-- if norm==false, moves group.Members[i] dist*dir in direction dir

moveRight(i:int,dist:float)
-- moves group.Members[i] a distance dist to the right

moveUp(i:int,dist:float)
-- moves group.Members[i] a distance dist up

setColor(i:int,color,j:int)
-- sets to Color:
-- if j~=nil the color of the j-th renderer of group.Members[i]
-- if j==nil the color of all renderers of group.Members[i]

setColorState(i:int,colst:bool)
-- if colst is true, colors group.Members[i] by its state
```

```
setDir(i:int,dir:Vector3)
-- sets the Euler Angles of group.Members[i] to dir

setNeighbours(i:int,nbrs:int[])
-- sets the neighbors of group.Members[i] to be the elements of nbrs

setPos(i:int,pos:Vector3)
-- sets the position of group.Members[i] to dir

setPosX(i:int,xpos:float)
-- sets the x-position of group.Members[i] to xpos

setPosY(i:int,ypos:float)
-- sets the y-position of group.Members[i] to ypos

setPosZ(i:int,zpos:float)
-- sets the z-position of group.Members[i] to zpos

setRot(i:int,rot::Vector3)
-- sets the Euler Angles of group.Members[i] to rot

setRotX(i:int,xrot:float)
-- sets the Euler x-Angle of group.Members[i] to xrot

setRotY(i:int,yrot:float)
-- sets the Euler y-Angle of group.Members[i] to yrot

setRotZ(i:int,zrot:float)
-- sets the Euler z-Angle of group.Members[i] to zrot

setScale(i:int,scale:Vector3)
-- sets localScale of group.Members[i] to scale

setState(i:int,s:int)
-- sets the state of group.Members[i] to s

setTurnToMoveDir(i:int,turn:bool)
-- if turn==true, ensures that group.Members[i] is
-- oriented towards his move forward direction

turnFwd(i:int,ang:float)
-- Rotates group.Members[i]  ang degrees around z-axis.

turnToAngle(i:int,targetAngle:float,deg:float)
-- Rotates group.Members[i] by deg degrees around y-axis
```

```
-- so that his Euler y-angle approaches targetAngle

turnToDir(i:int,dir:Vector3,speed:float)
-- Rotates group.Members[i] around z-axis with angular speed
-- equal to speed so that his Euler Angles approach dir

turnToDirSoft(i:int,dir:Vector3,speed:float)
-- SLOWLY Rotates group.Members[i] around z-axis with angular speed
-- equal to speed so that his Euler Angles approach dir
```

B.3.6 Inverse Kinematics Functions

These functions are relevant to *inverse kinematics* of group members; hence the group members must be Unity humanoid *avatars*. The variable `i` refers to the i-th group member.

```
ikSetLookAtAgent(i:int,j:int)
-- sets  group.Members[i] to look at group:Members[j]

ikSetLookAtPnt(i:int,pnt:Vector3)
-- sets  group.Members[i] to look at point pnt

ikSetLookAtWeight(i:int,ikWeight:float)
-- sets the look-at weight of  group.Members[i] to ikWeight

ikSetPosObject(i:int,ikGoal:string,go:GameObject)
-- sets the  IK position goal of ikGoal of  group.Members[i] to go
-- possible values for ikGoal (according to Unity humanoid rig)
-- "LH": Left Hand
-- "RH": Right Hand
-- "LF": Left Foot
-- "RF": Right Foot

ikSetPosVec(i:int,ikGoal:string,pnt:Vector3)
-- sets the  IK position goal of ikGoal of  group.Members[i] to pnt
-- possible values for ikGoal (according to Unity humanoid rig)
-- "LH": Left Hand
-- "RH": Right Hand
-- "LF": Left Foot
-- "RF": Right Foot

ikSetPosWeight(i:int,ikGoal:string,ikWeight:float)
-- sets the  IK position goal weight of ikGoal
-- of  group.Members[i] to ikWeight
```

```
-- possible values for ikGoal (according to Unity humanoid rig)
-- "LH": Left Hand
-- "RH": Right Hand
-- "LF": Left Foot
-- "RF": Right Foot

ikSetRot(i:int,ikGoal:string,rot:Vector3)
-- sets the  IK rotation goal of ikGoal of  group.Members[i] to rot
-- possible values for ikGoal (according to Unity humanoid rig)
-- "LH": Left Hand
-- "RH": Right Hand
-- "LF": Left Foot
-- "RF": Right Foot

ikSetRotWeight(i:int,ikGoal:string,ikWeight:float)
-- sets the  IK rotation goal weight of ikGoal
-- of  group.Members[i] to ikWeight
-- possible values for ikGoal (according to Unity humanoid rig)
-- "LH": Left Hand
-- "RH": Right Hand
-- "LF": Left Foot
-- "RF": Right Foot
```

B.3.7 Navigation Functions

In the following functions `navs[i]` is the *Navigation Agent* component of the i-th agent, i.e., of `group.Members[i]`. Also, `ground` is a GameObject on which a NavMesh will be baked.

```
navAddSurface(ground):Surface
-- returns a .surface component of ground;
-- a NavMesh will be baked on this component

navBakeSurface(surface:Surface)
-- bakes a NavMesh on surface

navToAgent(i:int,j:int)
-- sets the destination of navs[i] to be the position of group.Members[j]

navToPoint(i:int,point:Vector3)
-- sets the destination of navs[i] to be the point  point

navAttachAgent(i:int):NavMeshAgent
-- attaches and returns a NavMeshAgent to group.Members[i]
```

```
navActive(i:int, status:bool)
-- activates the NavMeshAgent component of group.Members[i]
-- ONLY IF the component was inactive

navGetDestination(i:int)
-- returns the .destination of navs[i]

navGetVelocity(i:int):Vector3
-- returns the velocity of navs[i]

navSetDestination(i:int, point:Vector3)
-- sets the .destination of   navs[i] to  point

navSetSpeed(i:int, speed:float)
-- sets the .speed of   navs[i] to  speed
```

B.3.8 Trail Renderer Functions

These functions are relevant to *trail renderers* attached group members. The variable `i` refers to the i-th group member.

```
trailAttach(i:int,offset:Vector3,color:Color,time:float,width:float)
-- attaches a trail renderer to  group.Members[i] with properties
-- .color color, .time time, .startWidth and .endWidth width
-- the trail renderer is offset from  group.Members[i] by offset

trailGetEndColor(i:int):Color
-- returns the .endColor of the trail renderer attached
-- to  group.Members[i]

trailGetEndWidth(i:int):float
-- returns the .endWidth of the trail renderer attached
-- to  group.Members[i]

trailGetStartColor(i:int):Color
-- returns the .startColor of the trail renderer attached
-- to  group.Members[i]

trailGetStartWidth(i:int):float
-- returns the .startWidth of the trail renderer attached
-- to  group.Members[i]

trailGetTime(i:int):float
```

```
-- returns the .time of the trail renderer attached
-- to  group.Members[i]

trailGetTrail(i:int):TrailRenderer
-- returns the Trail Renderer attached to  group.Members[i]

trailSetEndColor(i:int,color:Color)
-- sets the .endColor of the trail renderer attached
-- to  group.Members[i]

trailSetEndWidth(i:int,width:float)
-- sets the .endWidth of the trail renderer attached
-- to  group.Members[i]

trailSetStartColor(i:int,color:Color)
-- sets the .startColor of the trail renderer attached
-- to  group.Members[i]

trailSetStartWidth(i:int,width:float)
-- sets the .startWidth of the trail renderer attached
-- to  group.Members[i]

trailSetTime(i:int,time:float)
-- sets the .time of the trail renderer attached
-- to  group.Members[i]
```

B.4 Logic Functions

This library is contained in the file `logic.lua`. It contains functions which implement various logical *disjunctions* (OR operations) and *conjunctions* (AND operations) appropriate for *multivalued* logics (i.e., with truth values in the set [0,1]). These functions are also known as *T-conorms* and *T-norms* [1].Their restriction to two-valued logic (i.e., with truth values in the set $\{0,1\}$) reduces to the classical OR and AND operations. These functions admit a variable number of input arguments

```
D1(...):float
-- with input x1:float, x2:float,..., xK:float
-- returns OR operator max(x1,x2,...,xK)

D2(...):float
-- with input x1:float, x2:float returns x1+x2-x1*x2
-- returns OR operator
-- with input x1:float, x2:float,..., xK:float
-- returns the iterative application of the two-variable case
```

```
C1(...):float
-- with input x1:float, x2:float,..., xK:float
-- returns AND operator min(x1,x2,...,xK)

C2(...):float
-- with input x1:float, x2:float,..., xK:float
-- returns AND operator x1*x2*...*xK
```

This is the multivalued logic generalization of negation.

```
N1(x:float):float
-- returns 1-x
end
```

The following two functions implement a state transition function $\mathbf{T} : \{0,1\}^N \to \{0,1\}^N$. Hence the states are N-long sequences of 0's and 1's. $\mathbf{T}$ is encoded by two $2^N \times N$ arrays, S_{in} and S_{out}; if s is the m-th row of S_{in} and s' is the m-th row of S_{out}, then $\mathbf{T}(s) = s'$.

```
updateStateByNum(so:int[],Sin:int[][],Sout:int[][],Nagn:int):int[]
-- Given the state-transition function encoded by
-- the 2^Nagn-by-Nagn arrays Sin and Sout, if so is a 0/1 array
-- equal to the  m-th row of Sin, the function
-- returns the m-th row of Sout

updateStateByNumFast(so,Sout,Nagn)
-- This is a fast implementation of updateStateByNum(...)
-- Sin is omitted from the input, and it is is assumed
-- that its m-th row equals the binary representation of the integer m
```

B.5 Luts

This library is contained in the file `luts.lua`. It defines an array `luts` which contains LUTs, i.e., *look up textures*, used for visual postprocessing. The contents of the `luts` array are the following.

```
textures[  1] = "Filter-Blue-Heavy"
textures[  2] = "Filter-Blue-Soft"
textures[  3] = "Filter-Green-Heavy"
textures[  4] = "Filter-Green-Soft"
textures[  5] = "Filter-Red-Heavy"
textures[  6] = "Filter-Red-Soft"
textures[  7] = "Filter-Yellow-Heavy"
textures[  8] = "Filter-Yellow-Soft"
textures[  9] = "Blue-Grass"
textures[ 10] = "Cold"
textures[ 11] = "Enhancer"
```

```
textures[ 12] = "Greenpink"
textures[ 13] = "Hard-Warmer"
textures[ 14] = "HeavyContrast"
textures[ 15] = "I-Hate-Yellow"
textures[ 16] = "Literally-BW"
textures[ 17] = "Pale-Green"
textures[ 18] = "Saturated-Green"
textures[ 19] = "Tropical-Colors"
textures[ 20] = "True-BW"
textures[ 21] = "Warm"
textures[ 22] = "Barren"
textures[ 23] = "Barren-Night"
textures[ 24] = "Bleak"
textures[ 25] = "Dark-Knight"
textures[ 26] = "Desert-Glare"
textures[ 27] = "Dream-World"
textures[ 28] = "Fallen-World"
textures[ 29] = "Foggy"
textures[ 30] = "Fury"
textures[ 31] = "Haze"
textures[ 32] = "Icy"
textures[ 33] = "Matrix"
textures[ 34] = "Mr.Robot"
textures[ 35] = "No-Blood"
textures[ 36] = "Pastel-Dream"
textures[ 37] = "Poison"
textures[ 38] = "Posterizer"
textures[ 39] = "Shut-Down"
textures[ 40] = "Sin-City"
textures[ 41] = "Tale-World"
textures[ 42] = "Under-Water"
```

B.6 Neighborhoods

This library is contained in the file `formations.lua`. It creates neighborhood structures, which are used in conjunction with GCA rules. For examples of their use see Sections 3.2 and 3.3.

```
fpathNeighbours(N:int, fpath:string):int[][]
-- returns a neighborhood structure  read from file found in fpath
-- the file is an edge list with one edge (two vertices) per line
-- the returned neighborhood structure is a corresponding int array

makeNbhd(nbhdType:string, N:int, ...)
-- creates a neighborhood structure
```

```
-- variable input arguments list
-- possible types of neighborhood:
-- "rel1": calls function relativeNeighbours
-- "rel2": calls function relativeGridNeighbours
-- "fPath": calls function fpathNeighbours

relativeGridNeighbours(N:int, NC:int, relArray2:int[][], wrap:bool)
-- returns a neighborhood structure  for a grid
-- with N elements and NC columns
-- the returned neighborhood structure is a corresponding int array
-- which is built by adding to the neighborhood of vertex k
-- the vertices offset by relArray2
-- if wrap is true then the neighborhood structure is toroidal

relativeNeighbours(N:int, relArray:int[],wrap:bool)
-- returns a neighborhood structure  for a line with n elements
-- the returned neighborhood structure is a corresponding int array
-- which is built by adding to the neighborhood of vertex k
-- the vertices offset by relArray
-- if wrap is true then the neighborhood structure is toroidal
```

B.7 Object Functions

This library is contained in the file `functionsOBJ.lua`. It contains functions similar to the group member functions of Section B.3, but now they apply to isolated objects. The input argument `go` refers to the object on which the function is applied.

B.7.1 Light Functions

```
function objfuns.lgtAttach(go:GameObject,typ:string)
-- attaches a light component to go, of type typ
-- (available types: "directional", "spot", "point")
-- returns the go to which the light has been attached

function objfuns.lgtMake(room:room,key:string,name:string,typ:string,pos:Ve
-- creates a new object in the room room with key key and name name
-- the object has a light component of type typ
-- (available types: "directional", "spot", "point")
-- the position of go is pos and its Euler angles rot
-- retuns go

function objfuns.lgtGetColor(go:GameObject):Color
```

```
-- returns the color of the light component of go

function objfuns.lgtGetIntensity(go:GameObject):float
-- returns the intensity of the light component of go

function objfuns.lgtGetRange(go:GameObject):float
-- returns the range of the light component of go

function objfuns.lgtGetSpotAngle(go:GameObject):float
-- returns the spota angle of the light component of go

function objfuns.lgtGetType(go:GameObject):string
-- returns the type of the light component of go

function objfuns.lgtSetColor(go:GameObject,color:Color)
-- sets the color of the light component of go

function objfuns.lgtSetIntensity(go:GameObject,a:float)
-- sets the intensity of the light component of go

function objfuns.lgtSetRange(go:GameObject,r:float)
-- sets the range of the light component of go

function objfuns.lgtSetSpotAngle(go:GameObject,a:float)
-- sets the spot angle of the light component of go

function objfuns.lgtSetType(go:GameObject,typ:string)
-- sets the type of the light component of go
-- (available types: "directional", "spot", "point")
```

B.7.2 Object Functions

```
function objfuns.getAllComponents(go:GameObject):components[]
-- returns an array of all components of go

function objfuns.getDirMine(go:GameObject):Vector3
-- returns the forward direction of go

function objfuns.getDirToPnt(go:GameObject,point:Vector3):Vector3
-- returns the dir from go towards point

function objfuns.getDistToPnt(go:GameObject,point:Vector3):float
-- returns the dist from go towards point

function objfuns.getPos(go:GameObject):Vector3
```

```
-- return the position of go

function objfuns.getPosX(go:GameObject):float
-- returns the x-component of the position of go

function objfuns.getPosY(go:GameObject):float
-- returns the y-component of the position of go

function objfuns.getPosZ(go:GameObject):float
-- returns the z-component of the position of go

function objfuns.getRot(go:GameObject):Vector3
-- returns the Euler Angle of go

function objfuns.getRotX(go:GameObject):float
-- returns the x-Euler Angle of go

function objfuns.getRotY(go:GameObject):float
-- returns the y-Euler Angle of go

function objfuns.getRotZ(go:GameObject):float
-- returns the z-Euler Angle of go

function objfuns.makeObject(room:room,key:string,name:string,typ:string,pos
-- creates and returns Game Object go of type typ, in position pos
-- (available types: "cube", "sphere", "capsule", "cylinder", "plane";
-- everything else creates an empty Game Object)
-- go is named name registered in the room room with key key

function objfuns.moveFwd(go:GameObject,d:float)
-- moves go forward by distance d

function objfuns.moveInDir(go:GameObject,dir,d:float)
-- moves go in direction dir by distance d

function objfuns.moveRight(go:GameObject,d:float)
-- moves go to the right  by distance d

function objfuns.moveUp(go:GameObject,d:float)
-- moves go up by distance d

function objfuns.setPos(go:GameObject,pos:Vector3)
-- sets the position of go to pos

function objfuns.setPosX(go:GameObject,xpos:float)
```

```
-- sets the x-coordinate of the position of go to xpos

function objfuns.setPosY(go:GameObject,ypos:float)
-- sets the y-coordinate of the position of go to ypos

function objfuns.setPosZ(go:GameObject,zpos:float)
-- sets the z-coordinate of the position of go to zpos

function objfuns.setRot(go:GameObject,rot:Vector3)
-- sets the Euler angles  of go to rot

function objfuns.setRotX(go:GameObject,xrot:float)
-- sets the x-Euler angle of the position of go to xrot

function objfuns.setRotY(go:GameObject,yrot:float)
-- sets the y-Euler angle of the position of go to yrot

function objfuns.setRotZ(go:GameObject,zrot:float)
-- sets the z-Euler angle of the position of go to zrot

function objfuns.setScale(go:GameObject,scale:Vector3)
-- sets the local scale of go to scale

function objfuns.setColor(go:GameObject,color:Color)
-- sets the color of all renderers of go to color

function objfuns.textureObj(go:GameObject,assetpath:string,texturename:stri
-- textures go with texture named texturename
-- which can be found in Asset Bundle assetpath

function objfuns.turnFwd(go:GameObject,a:float)
-- turns go by a around its forward axis

function objfuns.turnRght(go:GameObject,a:float)
-- turns go by a around its right axis

function objfuns.turnToAngle(go:GameObject,targAng:float,dAng:float)
-- turns go around its y-axis by dAng per frame
-- until its y-Euler angle is targAng

function objfuns.turnToDir(go:GameObject,dir:Vector3,dAng:float)
-- turns go around its y-axis by dAng per frame
-- until it is alighned to direction dir
-- (the  y-component of dir is ignored)
```

```
function objfuns.turnToDir2(go:GameObject,dir:Vector3,dang:float)
-- turns go around its y-axis by dAng per frame
-- until it is alighned to direction dir
-- (the  y-component of dir is ignored)

function objfuns.turnToPnt(go:GameObject,point:Vector3,wr:float)
-- turns go until it looks at point

function objfuns.turnUp(go:GameObject,a:float)
-- turns go by a around its upwards axis

function objfuns.setTurnToMoveDir(go:GameObject,turn:bool)
-- if turn is true, the go will be turned in every frame
-- so that it faces towards it movement direction
-- if turn is false, the above behavior is disabled
```

B.7.3 Trail Renderer Functions

```
function objfuns.trailAttach(go:GameObject,offset:Vector3,color:Color,tim:fl
-- attaches and returns a  trail renderer component to go with
-- color color, time time, startWidth and endWidth width
-- the trail renderer is offset from  go by offset

function objfuns.trailGetEndColor(go:GameObject):Color
-- returns the endColor of the trail renderer attached to go

function objfuns.trailGetEndWidth(go:GameObject):float
-- returns the endWidth of the trail renderer attached to go

function objfuns.trailGetStartColor(go:GameObject):Color
-- returns the startColor of the trail renderer attached to go

function objfuns.trailGetStartWidth(go:GameObject):float
-- returns the startWidth of the trail renderer attached to go

function objfuns.trailGetTime(go:GameObject):float
-- returns the time of the trail renderer attached to go

function objfuns.trailGetTrail(go:GameObject):TrailRenderer
-- returns the trail renderer component attached to go

function objfuns.trailSetEndColor(go:GameObject,color:Color)
-- sets to color the endColor of the trail renderer attached to go
```

```
function objfuns.trailSetEndWidth(go:GameObject,width:float)
-- sets to width the endWidth of the trail renderer attached to go

function objfuns.trailSetStartColor(go:GameObject,color:Color)
-- sets to color the startColor of the trail renderer attached to go

function objfuns.trailSetStartWidth(go:GameObject,width:float)
-- sets to width the startWidth of the trail renderer attached to go

function objfuns.trailSetTime(go:GameObject,tim:float)
-- sets to tim the time of the trail renderer attached to go
```

B.8 Room Functions

This library is contained in the file `functionsROOM.lua`. It provides the following functions:

```
getRoomName(room:LuaRoom):string
-- returns the name of room which is the name of the scenario's directory

getScriptPath(room:LuaRoom):string
-- Returns the full path of the scenario's base directory.

getSceneName(room:LuaRoom):string
-- Returns the name of the room's scene.

useCameraScript(room:LuaRoom)
-- Creates a LuaCameraDomain that uses by default the camera.lua script.
-- Access to the camera domain's properties is possible from that script usi

addCamera(room:LuaRoom)
-- Alias of the useCameraScript function

getGroups(room:LuaRoom): Dictionary<string, LuaGroupDomain>
-- Returns a dictionary of the registered LuaGroupDomains in the room.
-- key: groupName, value: LuaGroupDomain

getGroupNames(room:LuaRoom):List<string>
-- Return a list of the names of the registered group domains in the room.

getGroup(room:LuaRoom, groupName:string):LuaGroupDomain
-- Retrieves one of the room's group domains from its name.
-- A group in general can be accessed via the Group variable in its script o

addEmptyGroup(room:LuaRoom, groupName:string, scriptPath:string):LuaGroupDom
```

```
-- Creates a new group and adds it to the group domain dictionary. Does not
-- groupName: The name of the group. Must be unique in the room.
-- scriptPath: The path of the script that will control the group.

addGroupMember(room:LuaRoom, groupName:string, assetPath:string, prefab:stri
-- Adds and returns a new member to the group with groupName
-- appearing in LuaRoom room
-- the new member is a prefab with name prefab,
-- found in Asset Bundle with path+name assetPath

runGroupScript(room:LuaRoom, groupName:string)
-- runs the group script assigned to group groupName
-- When a group is constructed manually its script must also be manually run

addGroup(room:LuaRoom, assetBundle:string, prefabName:string, N:int, groupNa
-- will create a group named groupName, containing  N copies
-- of the prefab found in the AssetBundle/with name prefabName
-- it will also attach the script found in scriptPath and run it

getDomains(room:LuaRoom):Dictionary<string, LuaDomain>
-- Returns a dictionary of the groups and individual domains in the room
-- (basically all the script associated objects) key: domainName, value: dom

getIndividualObjectNames(room:LuaRoom):List<string>
-- Returns the registered names of the individual domains in the room.

getIndividualObject(room:room, domainName:string):LuaIndividualDomain
-- Retrieves one of the room's individual domains from its registered name.
-- An individual object can be accessed via the 'self' property inside its s

addIndividualObject(room:room, assetBundle:string, prefabName:string, object
-- will create an instance of the prefab found
-- in the assetBundle from the path assetBundle/prefabName
-- it will also attach the script found in scriptPath and run it

getObjects(room:room):Dictionary<string, GameObject>
 -- Returns a dictionary of the registered objects in the room
 -- key: objectKey, value: gameObject

getObject(room:room, objectKey:string):GameObject
-- It will return an object registered with key objectKey found in room room

getObjectKeys(room:room):List<string>
-- Returns a list of the keys of the available registered objects in the roo
```

```
addRegisteredObject(room:room, objectKey:string, objectType:string, componer
-- Instantiates a new registered object in the room.
-- Use for simple objects that do not need their own scripts.

addObject(room:room, objectType:string,components:component[],activateObject
-- Instantiates a new registered object in the room.
-- Use for simple objects that do not need their own scripts.
-- It doesn't register the object with a name.

registerObject(room:room, objectKey:string, object:go)
-- Registers an object with a name.
```

B.9 Utility Functions

This library is contained in the file `utils.lua`. It contains various utility functions used throughout the scenaria.

```
makeTransFun(ss,N)
-- Returns two tables
-- One containing the array binary notation of all possible N inputs of the
-- Another one containing the values of the same function in array binary no

byte2bin(n:int,N:int):array,array
-- Creates an array with N binary digits of the positive integer n.

byte2binString(n:int):string
-- Returns the number n as a string in binary notation.

colMakeColorMap(colmap:string):array
-- Returns an array of color vectors
-- accepted input values: 'cool', 'parula'

stateToColor(s:int,numStates:int,cm:array):Color
-- Maps the state s to the corresponding color from the cm array

-------------------------------------------------------------------------
navAddSurface(gameObject:GameObject)
-- Adds a NavMeshSurface to a game object to make it navigatable

navBuildSurface(surface:NavMeshSurface)
-- Builds the navmesh on the specified surface

-------------------------------------------------------------------------
```

```
writeText(text:string, filepath:string, mode:string)
-- Writes text to a file, if mode isn't specified
-- will default to 'a' for append

closeFile(filepath:string)
-- Closes an opened file.

closeAllFiles()
-- Closes all the opened files.

--------------------------------------------------------------------------------
DirPntToPnt(pnt1:Vector3,pnt2:Vector3):Vector3
-- Returns the normalized direction vector from point pnt1 to point pnt2.

DistOfPnts(pnt1:Vector3,pnt2:Vector3):float
-- Returns the distance from pnt1 to pnt2.

RotateVector(V1:Vector3,a:float):Vector3
-- Returns vector V1 rotated around the Y axis by an angle a

unpackVector3(vector3:Vector3):float, float, float
-- Returns the coordinates of a Vector3

unpackVector2(vector2:Vector2):float, float
-- Returns the coordinates of a Vector2

arrayToVector3(array:array):Vector3
-- Returns the vector 3 that corresponds to the provided array
-- eg {1, 2, 3} => Vector3(1, 2, 3)

arrayToVector2(array:array):Vector2
-- Returns the Vector2 that corresponds to the provided array
-- eg {1, 2} => Vector2(1, 2)

vector3ToArray(vector3:Vector3):array
-- Returns the lua array that corresponds to the provided Vector3
-- eg Vector3(1, 2, 3) => {1, 2, 3}

vector2ToArray(vector2:Vector2):array
-- Returns the lua array that corresponds to the provided Vector2

arrayMean(array:array, startId:int, endId:int):float
-- Returns the mean of the elements array[startId:endId]

arrayAbsMean(array:array, startId:int, endId:int):float
```

```
-- Returns the mean of the absolute value of the elements array[startId:endI

newCSFloatArray(numberOfElements:int):Array<float>
-- Returns a C# default float array with the specified number of elements

-------------------------------------------------------------------------

attachTrailRenderer(gameObject:GameObject):TrailRenderer
-- Adds a new trail renderer component to a game object and returns it.

printOnScreen(text:string, posX:int, posY:int, color:Color)
-- Prints text on the specified screen position

listenToGenericShortcuts()
-- Listens for the sound related shortcuts and adjusts the volume accordingl
```

B.10 Visual Effects

This library is contained in the file `effects.lua`. It contains methods that manage global and local post processing effects.

Note that this library contains some advanced functions for local post processing effects and some lower level functions that are not presented in this list.

```
function M.getEffect(effectName:string):PostProcessEffectSettings
-- Returns a global post processing effect from its name

function M.enableEffect(effect:PostProcessEffectSettings, value:bool)
-- If value isn't specified, enables the effect, if value is false it disabl

function M.setProperty(effectProperty:ParameterOverride, value)
-- Overrides an effect parameter with the specified value

function M.getProperty(effectProperty:ParameterOverride)
-- Returns the value of the specified effect parameter

function M.isEnabled(effect:PostProcessEffectSettings)
-- Returns true if the effect is enabled

function M.disableAllGlobalEffects()
-- Disables all the enabled global effects

function M.clearAllGlobalEffects()
-- Resets the values of all the global effect parameters
```

```
function M.setLUTEffectTexture(lutEffect:SimpleLUT, textureName:string)
-- Sets the lookup texture of a SimpleLUT effect to the specified file in tl

function M.checkGlobalEffectInputs()
-- Listens and responds to the effect keyboard shortcuts [RShift]+numbers
```

Appendix C

Asset Bundles

In UDMSlua we utilize asset bundles to provide access to resources inside the lua scripts. The available resources are described in this chapter by category.

C.1 Models

The following models are available inside the `models/main` asset bundle:

- xbot
- ybot
- NeoMan
- MvnPuppet

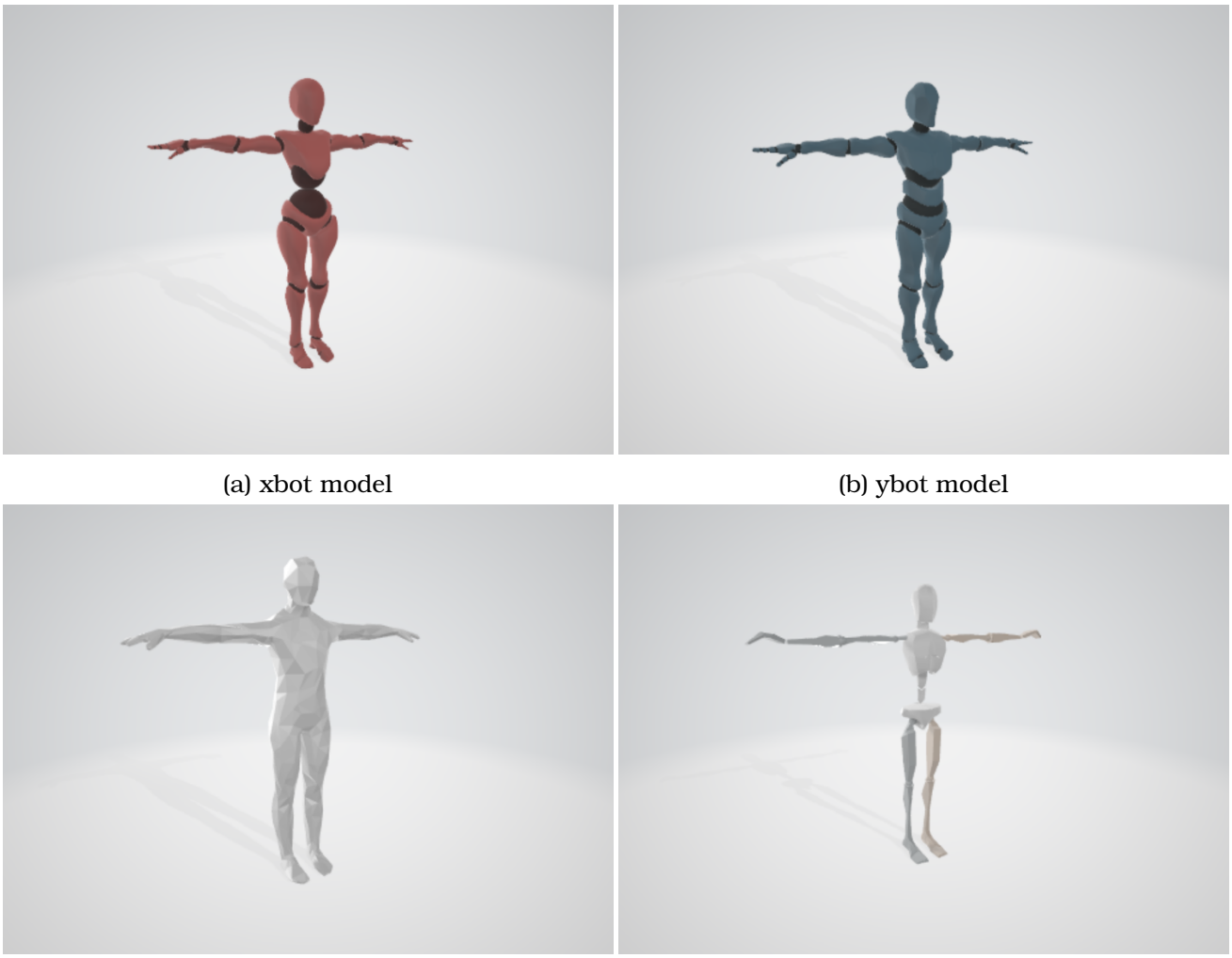
Check figure C.1 for a preview.

C.2 Textures

C.2.1 textures/dungeon

The following textures are available inside the `textures/dungeon` asset bundle:

- stone_tile.png
- stone_tile2.png
- stone_tile_normal.png
- stone_tile2_normal.png



(a) xbot model

(b) ybot model

(c) NeoMan model

(d) MvnPuppet model

Figure C.1: Available models

C.2.2 textures/ground

The following textures are available inside the `textures/ground` asset bundle:

- checkerboard_1.png
- checkerboard_2.jpg
- checkerboard_3.jpg
- checkerboard_4.jpg
- checkerboard_5.jpg
- design_01.jpg
- design_02.jpg

- design_03.jpg
- design_04.jpg
- design_05.jpg
- design_06.jpg
- design_07.jpg
- design_08.jpg
- design_09.jpg
- design_10.jpg
- design_11.jpg
- design_12.jpg
- design_21.jpg
- distortedPerspectiveGrid_01.jpg
- distortedPerspectiveGrid_02.jpg
- grid_1.png
- grid_2.jpg
- grid_black.jpg
- grid_blue.jpg
- grid_dots1.jpg
- grid_dots2.jpg
- grid_dots3.jpg
- grid_green.jpg
- grid_lines.jpg
- grid_sunny.jpg
- mosaic_01.jpg
- tessel_01.jpg
- tessel_02.jpg
- tessel_03.jpg

- tessel_04.jpg
- tessel_11.jpg
- tessel_12.jpg
- tessel_13.jpg
- tessel_14.jpg
- tessel_21.jpg

C.2.3 textures/scifi

The following textures are available inside the `textures/scifi` asset bundle:

- FloorTech01.jpg
- FloorTech01b.jpg
- FloorTech01c.jpg
- FloorTech02.jpg
- FloorTech02b.jpg
- FloorTech02c.jpg
- FloorTech03.jpg
- FloorTech04.jpg
- FloorTech04b.jpg
- FloorTech05.jpg
- FloorTech05b.jpg
- FloorTech05c.tga
- FloorTech05d.jpg
- FloorTech05e.jpg
- FloorTech06.jpg
- FloorTech07.jpg
- FloorTech08.jpg
- FloorTiles.jpg

- Wall01.jpg
- Wall01b.jpg
- Wall01c.jpg
- Wall02.jpg
- Wall02b.jpg
- Wall02c.jpg
- Wall03.jpg

C.2.4 textures/various

The following textures are available inside the `textures/various` asset bundle:

- Floor_01.png
- Sand_01.png
- Sand_02.png
- Water.png
- dirt_dry.png
- grass_01.png
- grass_02.png
- grass_dry_02.png
- roof_tile_01.png
- snow_01.png
- dirt_ice.png
- Roof_tile_02.png
- ice.png
- grass_dry_01.png
- Dirt_01.png

C.3 LUTs

The available look up textures inside the `textures/luts` asset bundle are the following [1]:

1. Filter-Blue-Heavy
2. Filter-Blue-Soft
3. Filter-Green-Heavy
4. Filter-Green-Soft
5. Filter-Red-Heavy
6. Filter-Red-Soft
7. Filter-Yellow-Heavy
8. Filter-Yellow-Soft
9. Blue-Grass
10. Cold
11. Enhancer
12. Greenpink
13. Hard-Warmer
14. HeavyContrast
15. I-Hate-Yellow
16. Literally-BW
17. Pale-Green
18. Saturated-Green
19. Tropical-Colors
20. True-BW
21. Warm
22. Barren
23. Barren-Night
24. Bleak

[1] Note that their index in this list is their corresponding index inside the `luts` library (see B.5).

25. Dark-Knight
26. Desert-Glare
27. Dream-World
28. Fallen-World
29. Foggy
30. Fury
31. Haze
32. Icy
33. Matrix
34. Mr.Robot
35. No-Blood
36. Pastel-Dream
37. Poison
38. Posterizer
39. Shut-Down
40. Sin-City
41. Tale-World
42. Under-Water

Appendix D

Extending UDMSlua

D.1 Adding Lua Scripts

To create a new scenario do the following.

1. Go to the basic scenario folder. The fastest way to do this is using the corresponding selection Open Base Directory in the application Main Menu, see Figure D.1:
2. Inside the basic scenario folder create a subfolder which will be the main folder of the new scenario; or you can have a subfolder for a *group* of related scenaria, which includes one subsubfolder for each scenario. See Figure D.2 for the folder hierarchy. In this example the subfolder `Examples` was created to group a set of scenaria.
3. In our main scenario folder we must have a file named `settings.lua` which is the basic scenario Lua file (see Figure D.3).
4. Write the `settings.lua` script and any other script files declared in it. When the scenario (file collection) is ready, or if we want to check our progress, we run it with the Main Menu selection Scenario Select.
5. When the scenario is running, we can open the console (with the tilde key ~ for debugging pruposes.
6. While developing a scenario, we can live update its resulting scene using the Main Menu's selection Reload Selected Scenario.

D.2 Adding Lua Libraries

To add a new function in a Lua library, or to create a new library, do the following.

1. Open or create the `.lua` file corresponding to the library. All library files must be located in the subfolder `UDMS_Data/StreamingAssets/Scripts/Libraries/`.

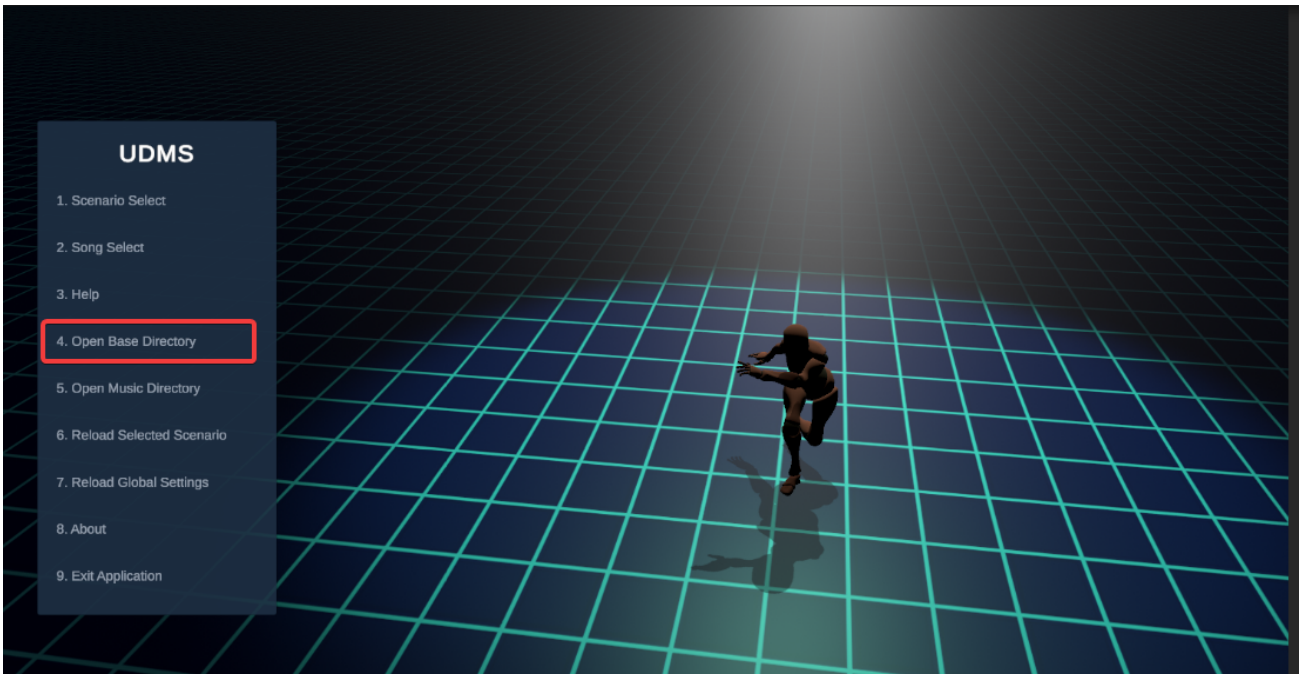


Figure D.1: Open basic scenario folder

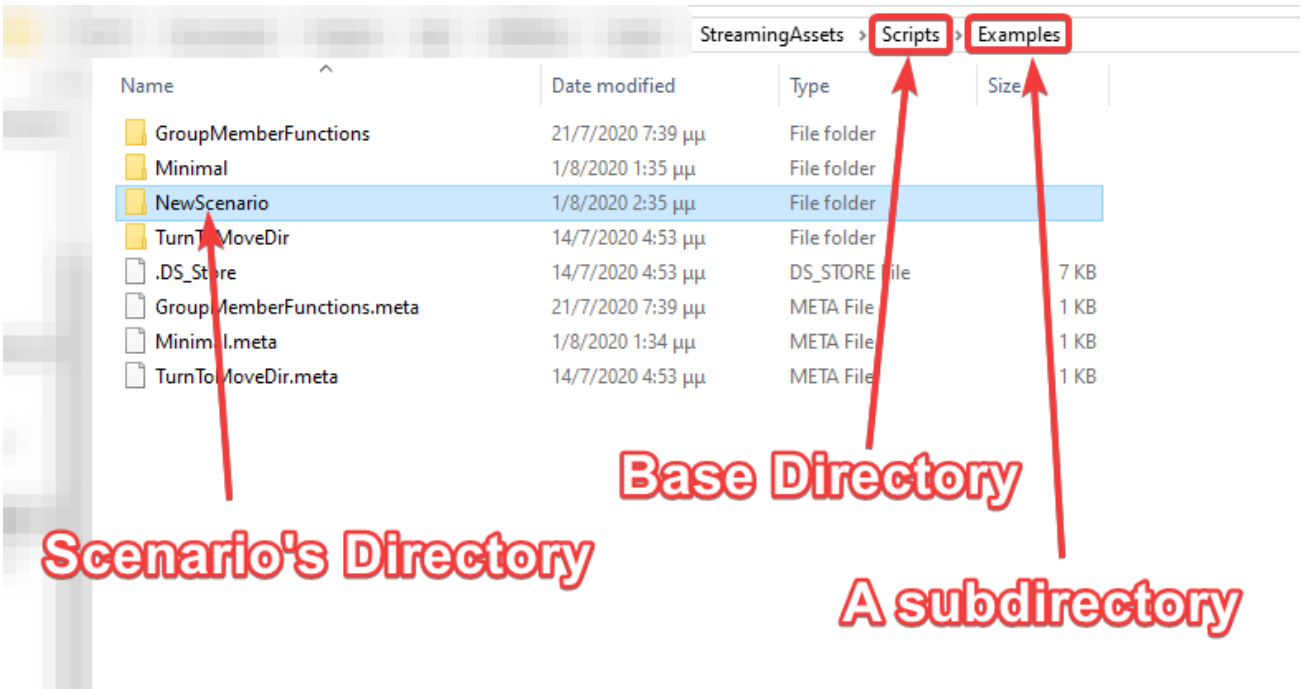


Figure D.2: Example for the path of main scenario folder

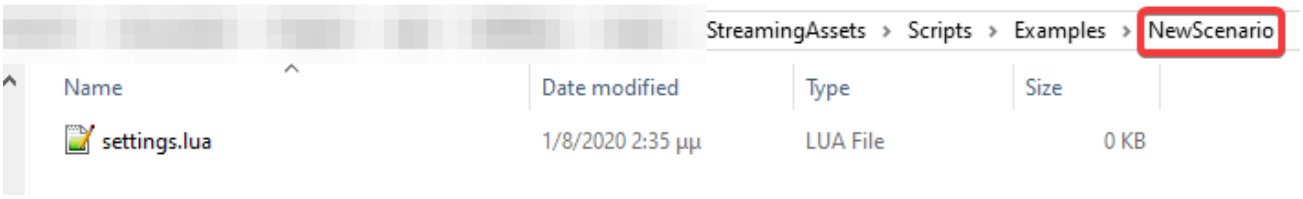


Figure D.3: Example of a valid scenario folder

2. Create the desired functions and make sure they are included in the table returned by the main library function (see the existing library files to better understand how this is achieved).

For example, a new library named `sayings` is created and deployed as follows.

1. The path `UDMS_Data/StreamingAssets/Scripts/Libraries/sayings.lua` corresponds to the newly created file `sayings.lua` which contains the following lines.

```
local M = {}
function M.greet(name)
print('Hello '..name)
end
return M
```

2. We can now call the `greet` function in any of our scripts as follows.

```
local M = require('sayings')
-- Will print 'Hello Fox' on the console.
M.greet('Fox')
```

Avoid library names which duplicate standard Lua library names. In case a name duplication exists, `require` will return the standard library.

D.3 Adding Streaming Assets (Music Files etc.)

UDMSlua can play any music files (but they must be in the `.ogg` format) which you add to the `Music` subfolder. Similarly, you may add any other streaming assets in the Streaming Assets folder.

D.4 Adding Resources

To add new Assets (models, textures etc.) to UDMSlua you must work with the UDMSlua *project*. This requires some experience with the Unity game engine. In this section we will show how to add a new texture.

Figure D.4: Adding a texture to an Asset Bundle.

1. We open the project in the Unity Editor, add the required elements and then incorporate them in an existing or newly created *Asset Bundle*. For example, in Figure D.4 we add the texture `stone_tile2` to the Asset Bundle with path: `textures/dungeon`

2. Then we recompile the Asset Bundles to contain the new Assets; to do this, we choose the Build Asset Bundles option from the Assets menu (see Figure D.5).

3. Finally, we move the recompiled Asset Bundles from the project to the application, by copying the files from the project's path `Assets/StreamingAssets/AssetBundles` to the application's path `UDMS_Data/StreamingAssets` (replacing, if necessary the old files with the same names). Now we can access the new Assets from the application's Lua scripts. In this example, we can use the new texture as follows: we create a new scenario with the file *settings.lua* including the lines;

```
local UE = CS.UnityEngine
scene = 'RoomA'
function setUp()
      local myCube = Room:InstantiateObject('cube')
      local textureName = 'stone_tile2'
      local assetPath = 'textures/dungeon'
      local texture = CS.LuaScripting.AssetManager.LoadAsset(typeof
         (UE.Texture),textureName,assetPath)
      myCube:GetComponent(typeof(UE.Renderer)).material.mainTexture
          = texture
end
```

Next we load the scenario and we see the cube is textured as in Figure D.6.

D.5 Adding C# Scripts

To add new functionalities to UDMSlua you must work with the UDMSlua *project*. This requires some experience with Unity game engine programming. In this section we will show how to

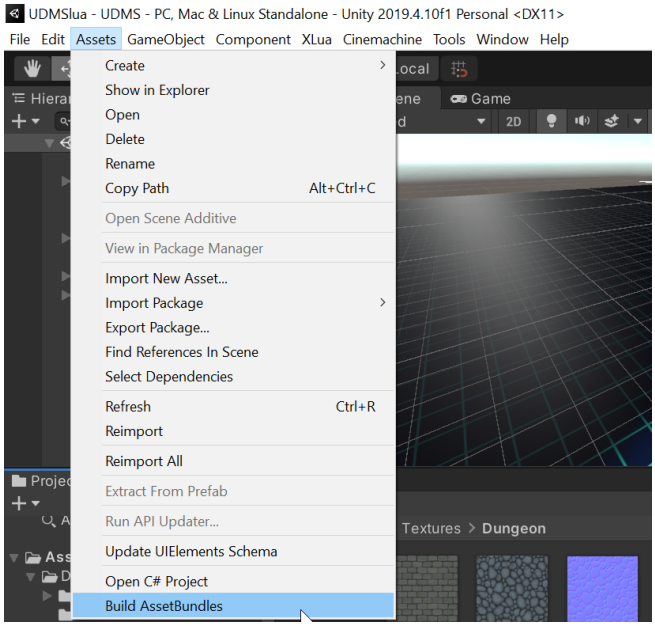


Figure D.5: Recompiling Asset Bundles.

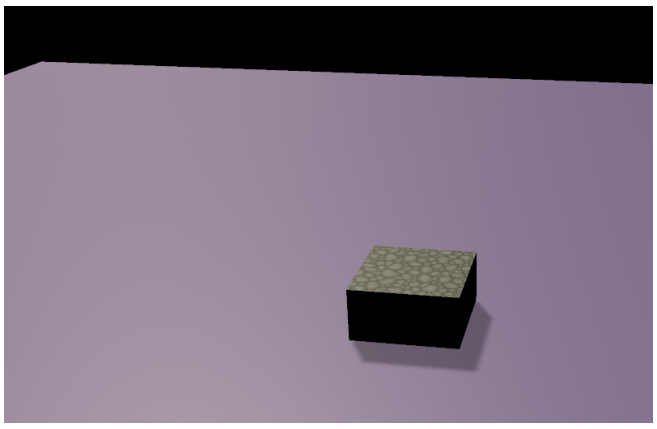


Figure D.6: The new texture is applied to a cube

add a new function to the class `LuaGroupDomain` and call it from a Lua script.

1. Open the `UDMSlua/Assets/Scripts/UDMS/LuaGroupDomain.cs` script.
2. Add the next *public* function inside the class:

```
public Vector3 GetGroupCenter()
{
      if (Members.Count == 0) return Vector3.zero;
      var center = Vector3.zero;
      foreach (var member in Members)
      {
            center += member.transform.position;
      }
      return center / Members.Count;
}
```

3. Connect the new function to the application's Lua environment. This is done with the Unity Editor's menu option **xLua->Generate Code**.
4. Call the new function from any object of the class `LuaGroupDomain` inside a Lua script:

```
-- Assuming Group is a LuaGroupDomain object
local groupCenter = Group:GetGroupCenter()
```

Appendix E

Imported Assets Used in UDMSlua

Here is a list of assets created by others and used by us in UDMSlua.

E.1 External Packages

1. *DoTween*. From Unity store.
2. *In-game Debug Console*. From Unity store.
3. *Standalone File Browser*. From GitHub.
4. *Cinematic Look LUT Library*. From Unity store
5. *xLua*. From GitHub.

E.2 Models

1. XBot and YBot from Mixamo.
2. MVNPuppet from Unity free package MVN Live Animation.
3. NeoMan from from Keijiro Takahashi's PuppetTest project in GitHub.

E.3 Animations

From Mixamo.

E.4 Music

1. Several compositions from the Free Music Archive.
2. Some more composed with PercussionStudio. These are not our compositions, I have downloaded them from Moosware Net.

E.5 Textures

1. *Dungeon stone textures*. From Unity store.
2. *SciFi textures*. From Unity store.

Bibliography

[1] Klir, George J., and Bo Yuan. *Fuzzy sets and fuzzy logic: theory and applications*. 1996.

[2] Masuda, Naoki, and Kazuyuki Aihara. "Spatial prisoner's dilemma optimally played in small-world networks." *Physics Letters A* 313.1-2 (2003): 55-61.

[3] Nowak, Martin A., and Robert M. May. "Evolutionary games and spatial chaos." *Nature* 359.6398 (1992): 826-829.

[4] Packard, Norman H., and Stephen Wolfram. "Two-dimensional cellular automata." *Journal of Statistical physics* 38.5-6 (1985): 901-946.

[5] Perc, Matjaž, and Attila Szolnoki. "Social diversity and promotion of cooperation in the spatial prisoner's dilemma game." *Physical Review E* 77.1 (2008): 011904.

[6] Smith, Kenneth C. "A multiple valued logic: a tutorial and appreciation." *Computer* 21.4 (1988): 17-27.

[7] Weisstein, Eric W. "Totalistic Cellular Automaton." `https://mathworld. wolfram. com/` (2002).