



Transport and Telecommunication, 2023, volume 24, no. 4, 349–360
Transport and Telecommunication Institute, Lomonosova 1, Riga, LV-1019, Latvia
DOI 10.2478/tjt-2023-0027

SCRIPTING SCENARIOS OF PEDESTRIAN BEHAVIOR IN A COMPUTER SIMULATOR OF SECURITY MONITORING SYSTEM: A PRACTITIONER'S PERSPECTIVE

Jarosław Sugier

Wrocław University of Science and Technology
Faculty of Information and Telecommunication Technology
Wrocław, Poland, 11/17 Janiszewskiego St., 50-372 Wrocław
jaroslaw.sugier@pwr.edu.pl

The subject of this paper is the AvatarTraffic simulator – a computer system capable of modelling in real time environments such as subway stations or airport halls populated with tens or hundreds of moving figures, which, in addition to pedestrian traffic typical for this type of objects, can perform predefined sequences of events and actions formulated as a simulation scenario. Thanks to the integration with a real monitoring system, the simulator, in addition to providing data streams (including video) generated by the virtual scene, is also able to dynamically respond to actions taken by the system's staff. Using the Unity simulation engine as the implementation platform, a number of practical problems had to be solved during the development, two of which are the subject of this article: a) supervising and correcting the work of AI algorithms used in Unity to simulate the pedestrian movement of avatars, and b) a textual description of the scenario of events taking place on the stage in a way editable for experts planning tests of the monitoring system. Some more challenging cases of people movement are discussed (including creating queues and passing through doors) and the paper presents original algorithms correcting the work of the Unity's built-in methods in the situations when the coordinated behaviour of people groups is required. Because of the specifics of the simulator environment the scenario needed to be expressed in a JSON text file, and the article presents the implemented mechanisms of its compilation directly to the C# runtime environment and discusses the original command language which was created to model sequences of events and actions making up the scenario.

Keywords: Scenario scripting, state machine, navigation, pathfinding, Unity 3D engine

1. Introduction

The paper discusses specific issues which arose during implementation of AvatarTraffic – a simulator developed on the Unity platform (Unity Technologies, 2023a) capable of visualizing a 3D scene in which people or groups of people move along designated paths and behave according to a specific scenario. The simulator can cover an area corresponding to the size of, for example, a subway station or an airport hall with tens or hundreds of people, and moreover can be integrated with a real security monitoring system, for which various data are generated imitating, for example, the operation of physical surveillance cameras (video streams) and signals from various virtual sensors (device statuses). Thanks to the possibility of reproducing various situations, the simulator makes possible: a) practical verification of the operation of the monitoring system at the design stage (location of cameras, sensor configuration, etc.), b) simulation of dangerous events and tests of critical procedures related to them (accidents and other unusual situations which would be difficult to stage in real life), and c) generating a large amount of reliable training data for machine learning algorithms, which can then be used to create AI systems supporting the surveillance (e.g. in automatic detection of malicious behaviour in video streams, instant signalling of emergency situations based on analysis of sensor data, etc.).

Being an extension of the two conference publications (Sugier, 2021 and Sugier, 2022), the subject of this article covers two aspects of the simulator: supervision of the Unity's AI algorithms that control the movement of virtual characters – the so-called *avatars* – which turned out to be insufficient in scenes requiring the interaction of a larger number of people, and an original mechanism for textual description and execution of scenarios of simulated events. The organization of the text is as follows. After presenting the review of the literature and the motivation of this work in Chapter 2, Chapter 3 discusses the configuration of the AvatarTraffic simulator and its internal resources that are related to the problem of controlling the people movement or are scenarios subjects. Then, in Chapter 4, the originally developed procedure for supervision and correction of the operation of navigation algorithms is presented, and in chapter 5 – the implemented method of describing the simulation scenario and the resulting command language which was used for its specification. The conclusions summarize the paper in Chapter 6.

2. Motivation and related work

Planning the route of object movement among the obstacles of a specific scene is a task that has been studied for years in robotics and game programming. The established and currently widely used approach is based on two concepts: the navigation mesh (*NavMesh*) describing the scene surfaces available for motion planning, and the navigation agent (*NavAgent*) which plans and controls the movement. When arranging a traffic path on a large *NavMesh*, the agent usually uses a hierarchical approach: its outline is developed at the top level of the hierarchy, while the details within individual areas are determined dynamically according to detected conditions. As shown, for example, by Rahmani and Pelechano (2017), various alternatives to the hierarchical organization of the network with appropriate additional pre-processing can significantly speed up the operation of agents.

Pathfinding methods in navigation networks are usually based on the standard A* algorithm (Hart *et al.*, 1968), which is a heuristic extension of Dijkstra's (1959) classic procedure for finding the shortest path between graph nodes. Adaptation of such an approach to the situation of simultaneous operation of many agents with specific kinematic constraints was described, for example, by Hönig *et al.* (2016). Recently much attention has been paid to the use of machine learning algorithms for multi-agent collision avoidance (see e.g. Majumder, 2021 and Qiu, 2020) but it seems, however, that such an approach still needs to be optimized before it reaches the efficiency enabling practical implementation in simulation engines.

On the other hand, the specification of the sequence of events and activities that make up the simulation scenario is an essential part of any complete simulator. An early classification of scripting languages supplementing the capabilities of popular game engines was presented by Anderson (2011) but this work does not focus on the creation of proper scenario scripts and deals more broadly with the problem of basic programming of the behaviour of scene objects. A dedicated solution for the description of simulation scenarios was described, for example, by Van Est *et al.* (2011), who proposed a standalone graphical editor defining a sequence of events in the form of a causality graph. The presented example application in the Unreal 3 environment included scenarios for training a safety inspector on an oil drilling site. The aim of the proposal developed by Sloomaker *et al.* (2014) was to support scenarios that enable the acquisition of complex cognitive skills, and the proposed non-standard language made it possible to express the conditions and actions that make up the course of events. In turn, Duval *et al.* (2015) applied pedagogical trees with the expected educational effects in the methodology of preparing serious games for professional training of employees. Finally, a scripting language giving full control over the behaviour of the script characters was described by Busetta *et al.* (2017), and thanks to the representation in the form of a scene chart, it enabled supervisor interventions during the course of the simulation.

It is worth underlining that the word “scripting” in this article is used in a specific sense. In the common language of game developing, it primarily means low-level programming of the behaviour of the game object (character, vehicle, but also a light source, camera, etc.) which is an alternative to graphically configuring the object's attributes via the GUI interface. Such programming is an essential component of any platform and uses either general-purpose languages such as C# in Unity (Unity Technologies, 2023c) or C++ in Unreal (Epic Games, 2023), or its own interpreter. In this work, however, the term “script” refers to a situation where the configurator of the simulator (who is usually a monitoring system expert, not a programmer) defines a simulation scenario as a sequence of events and actions along with the actors who will replay them. Such a definition is treated as part of the scene specification, is expressed in some notation and as a “scenario script” is part of the simulator's input, loaded with the scene definition.

3. The AvatarTraffic simulator

3.1. Operation mode and interfacing

AvatarTraffic is a simulator developed in the Unity 3D environment, which after starting the application loads the configuration file, builds the scene described in it, creates dynamic objects – including avatars – and simulates the described scenario in real time, communicating with the physical monitoring system via network connections.

The configuration file is described in the JSON format, which was adopted to enable full data editability, without launching of the application itself. Its individual sections describe the complete simulated environment, including: a) the geometry of the scene (rooms, connecting passages, platforms, doors, etc.), b) objects related to visualization (lighting, the main view camera, video streaming interface), c) people appearing on the scene, both controlled by the course of the scenario (actors) and implementing constant patterns of behaviour (background traffic independent of the main plot of events), d) objects

representing elements of the monitoring system (surveillance cameras, sensors, card readers at authorization points, etc.), and finally e) simulation scenario, expressed as a set of any number of so-called *scripts*.

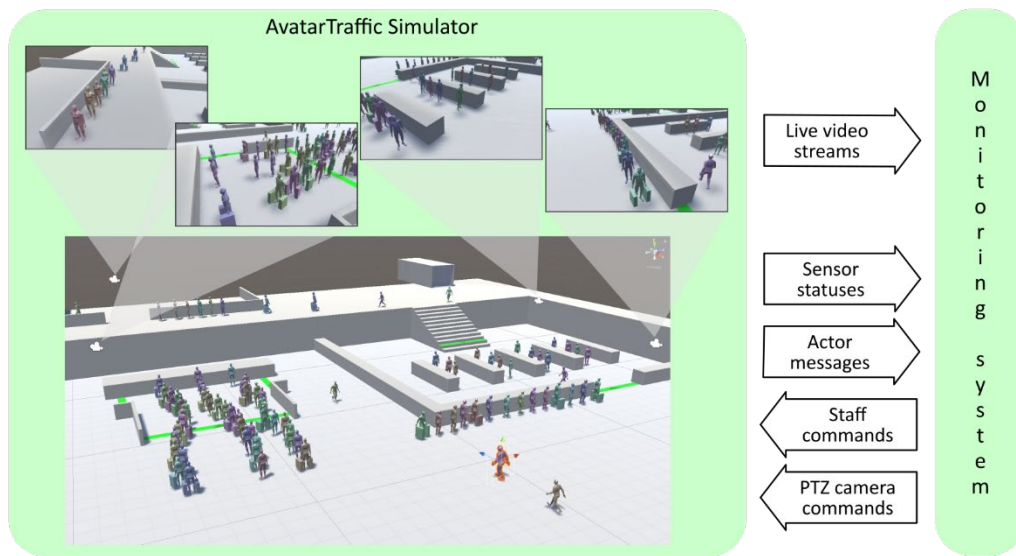


Figure 1. The AvatarTraffic simulator and its communication with the monitoring system

Although standalone operation is also possible e.g. for generation of data for ML algorithms, the full functionality of the simulator is exploited after its integration with a real monitoring system. Communication with it takes place by connecting the computer running the simulator to the network of the system (see Figure 1). In addition to sending video streams from virtual surveillance cameras placed on the stage, two-way communication between the simulator and the system includes the exchange of messages over the NATS protocol:

- sending events reported by virtual sensors placed on the stage, e.g. door opening or closing, motion detection, authorization of the card in the reader, etc.;
- sending messages from the actors defined in the scenario, e.g. reporting the presence of suspicious persons, standard reports on the course of the patrol, etc.;
- receiving commands from the system supervisor triggering various variants of the scenario, e.g. sending employees to check the reported defect, blocking certain passages, etc.;
- executing Pan-Tilt-Zoom commands directed to virtual surveillance cameras.

Using identical network interfaces and transmission protocols as those used by real components of the monitoring system (sensors, cameras, card readers, etc.), the information generated by AvatarTraffic remains technically indistinguishable from those coming from real devices. The only exception are the conversations of supervisors with simulated characters – actors realizing the scenarios described in the scripts. In these cases, the dialogue with the avatars (e.g. representing a security patrol), which in reality may be, for example, simply a communication via a walkie-talkie, must be transformed into an exchange of NATS messages predefined in the scenario.

In the prototype presented in this article, the main task was primarily to obtain reliable movement of people moving on a three-dimensional stage; their photorealistic visualization, with detailed texturing, lighting, etc., remained outside the scope of the work, but this aspect can be supplemented later in the development, using standard, but highly time- and labor-intensive techniques of computer graphics.

3.2. Resources of the scenes

In general, realistic planning and then visualization of the movement of a human figure – the most important aspect of the simulator's work – is a rather complicated task, but the Unity platform offers ready-made and practically effective solutions in this respect, which became the basis for the developed implementation. Unity implements the character's movement according to a typical approach, as it was described in Chapter 2: a NavMeshAgent component assigned to each avatar dynamically plans a route to a given point in the scene (Unity Techn., 2023b) and takes care of the full support of character animation controlled in accordance to its current type of movement (slow creeping, walking or running, turning or

turning back). The use of this functionality in the code comes down to passing the destination point, and then only to monitoring the progress of the movement, leaving the AI engine to create a path and move along it at a given speed. All motion details such as realistic decelerations, accelerations, turns and obstacle avoidance are handled by the animation engine and have proven to be sufficiently effective in the implementation.

The *NavMeshAgent* mechanism was used in the two basic resources – avatars and avatar generators; in addition, three additional objects were needed to supervise its work in more complex cases: Queues, Doors, and Waiting Areas. Figure 2 shows an example of these resources used in the simulation of a platform at a railway station. The security guards appearing on this stage are modelled by individual avatars controlled by simulation scenario scripts, avatar generators are used to create streams of passengers getting on or off the train, the entrance to and exit from the platform are realized as queues with three parallel threads, a separate door is intended for an authorized staff and the two waiting areas take control of the avatars lingering on the platform.

Avatars. The avatar's static path is described in the configuration file as a sequence of waypoints, given either as absolute coordinates or as a reference to a queue, door, or waiting area. In the case of a reference, the relevant object takes control of the *NavAgent*'s operation and manages behaviours such as waiting for the door to open or queuing. In the simplest case of an avatar – an automaton, its route is carried out without the intervention of scripts by its own controller which boils down to directing the *NavAgent* to successive transition points. In the case of a character dynamically reacting to events occurring during the simulation, his behaviour is described in the scenario, and the path consists only of the starting point where he is to appear on the scene. The subsequent transition commands are generated by the scripts during the simulation, but all it takes is to pass the destination to the *NavAgent* component.



Figure 2. A scene modelling a train platform with objects like avatar generators, doors, waiting areas and queues

Avatar generators. The stream of avatars moving along the same route is created automatically by an autonomous generator. It dynamically creates avatar-type objects either with a given intensity, or keeping a certain distance between them on the path, or implementing a distribution of generation density loaded from an external file. The generators are primarily used to represent the constant movement of people in the background (e.g. continuous streams of passengers at the subway station, etc.) and then their operating parameters remain unchanged throughout the simulation, but they can also act as actors controlled by the scenario scripts (e.g. a patrol consisting of three people sent to inspect an event that has happened on the scene).

Queues. In areas covered by monitoring systems the movement of people often creates queues, for example at ticket control points at the entrance to the platform, at baggage drop points, etc. In *AvatarTraffic* the queue can be optionally in the form of many parallel threads operated concurrently; e.g. in Figure 2 the queue representing the entrance to the platform consists of three threads. The behaviour of the avatar in the queue thread is specific and its movement is forced by the queue controller: it directs an approaching avatar to its assigned place in the selected thread and initiates transitions when the queue moves. The avatar at the head of the thread can be released with a certain average frequency or according to a distribution read from a file.

Door. The doors appearing on the stage can be of various mechanical construction (1- or 2-leaf, sliding or hinged), but most of all with a different principle of operation: automatic or locked. Automatic doors operate on the basis of a motion sensor and from the point of view of navigation they made a point of the route that does not require any additional service. In the case of lockable doors, a dedicated controller was needed which took control of the approaching avatars and executed a specific sequence of events: waiting a delay required to open the lock or use the card in the reader, opening the door leaves, the well-arranged passage of the waiting avatars, and closing the leaves.

Waiting areas. The waiting zone is a rectangular area with additional parameters that determine the average total time the avatar stays in it, as well as the speed of its movement and the time it stays in one place if the avatar is supposed to move randomly within the area. After the waiting time expires, the avatar resumes its navigation similarly to leaving the queue or door; an alternative is to use scenario commands that can instantly release one or all avatars residing in an area.

4. Supervising the AI navigation

4.1. Controlling simulation of queues

The queue controller takes control of the avatar's `NavMeshAgent` as soon as it becomes its next waypoint, and returns the control when the avatar is released from its thread. The sequence of events for enlisting an avatar in the queue had to be quite involved and consists in the following:

- (1) the queue controller assigns the avatar to the selected thread based on, e.g., thread occupancy or distance;
- (2) the thread controller passes to the avatar's `NavMeshAgent` its current endpoint as a target and stores it in the list of approaching avatars;
- (3) the list of approaching avatars is checked in each simulation cycle; if any of them comes closer to the endpoint than `NavDecisionDist` (queue's custom parameter), it is considered to be a new member of the thread and transferred to the standing list; (the task of the `NavMeshAgent` algorithms is to complete the movement to the destination point and stop)
- (4) the calculated new thread endpoint is passed to the rest of the approaching avatars so that they would not block the new standing one in its completion of the route.

After the list of avatars standing in the thread is established, the controller's task is to release the front standing person with requested intensity and send commands to move one position to the remaining ones.

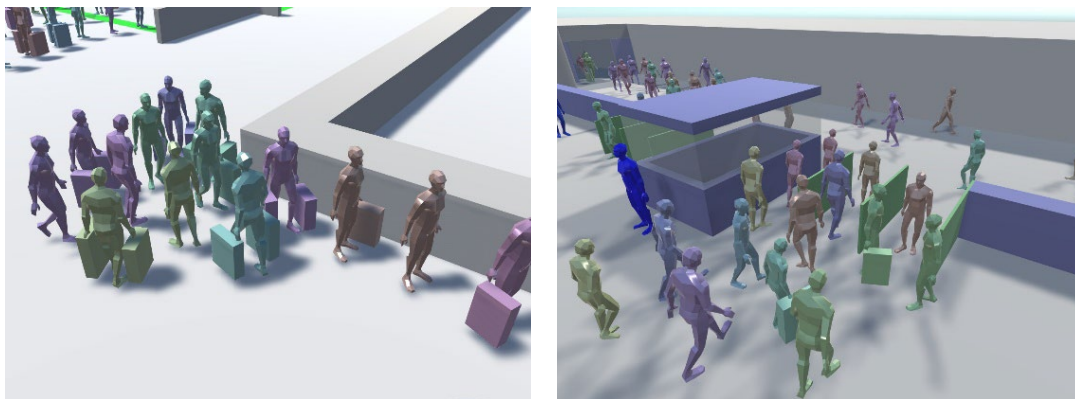


Figure 3. Incorrect operation of queues: a blockade at the end of the thread (left) and a queue jam with an avatar forced into the wrong thread (right)

The key to avoiding blockages was the introduction of `NavDecisionDist` parameter in step (3) and early selection of the closest upcoming avatar as the one who actually takes the position in the queue. Without this mechanism, when multiple avatars were directed to the same thread endpoint, their `NavMeshAgents` would block each other and no one could actually reach it (an example of which is shown in Figure 3). Selecting just one avatar and directing the others to a new endpoint was the only workable solution. Practically verified values of `NavDecisionDist` were at least 3 to 5 times larger than the agent's `stoppingDistance` parameter.

The second problem occurred with groups of avatars heading to adjacent threads of the same queue: their virtual “elbowing” led to avatars being pushed into the thread next to the one that they were assigned to (illustrated in the right part of Figure 3), or in front of the avatar that should come before it. The first problem affects the upcoming avatars and the second – the standing ones, but in both cases the queue thread was blocked and avatars were piling up around its end.

In order to remove the first type of problem, it was necessary to implement an additional procedure to check the approaching avatars: from among them, those that were considered blocked were first identified (based on their wrong orientation or too slow movement) and then assigned to a thread in which they have been blocked. This solution was more efficient than clearing the way for the blocked avatar to its destination while the resulting imbalance in the load of the threads was usually acceptable.

Interventions in the second case were performed each time the first avatar was released from the thread, as this problem manifested itself only when the queue moved (and thus its detection could be performed much less frequently than the previous one). The mechanism was based on reviewing the positions of all avatars standing in queue along its direction and – in the case of detecting a wrong succession – correcting the list in accordance with the actual order. As in the above case, it was easier to accept the actual position of the avatar than to carry out the procedure of exchanging its position with the neighbour.

4.2. Monitoring door passages

AvatarTraffic implements two types of doors: automatically opened and lockable. Automatic doors open to any approaching pedestrian, require no special supervision and can be treated as regular waypoints in operation of autonomous or generated avatars. In the case of doors with a lock, the opening procedure takes some time, during which a group of people can gather in front of them. Their orderly passage after opening required the development of an appropriate supervisory system, as the built-in AI algorithms often led to the blocking of avatars (an example is shown in Figure 4), obstructing each other's path.

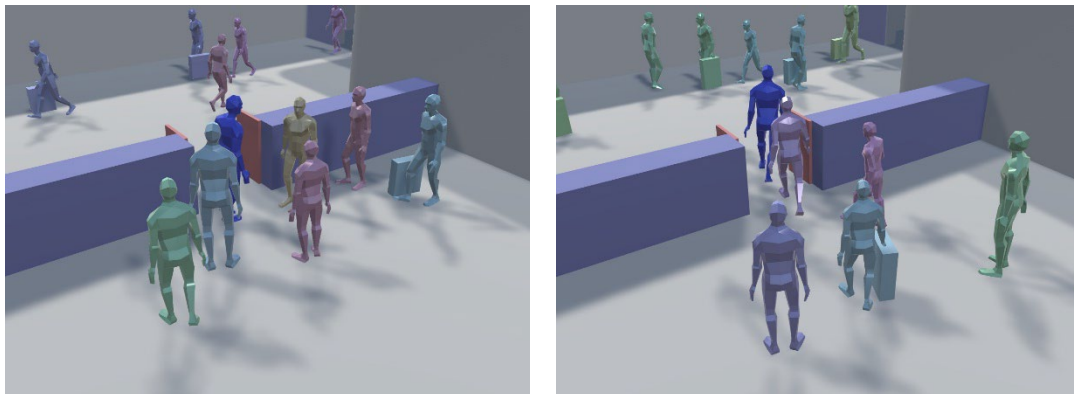


Figure 4. A deadlock of AI navigation agents in a group of avatars trying to pass an open door (left) and their correct operation with an additional guidance from a dedicated controller (right)

Three lists of avatars were needed in the implemented solution: approaching, waiting for opening (i.e. standing in front of), and released ones, i.e. currently passing through after opening. The supervising algorithm is even more complex than the one for queues and consist in the following steps:

- (1) any avatar that is directed to the door is added to the list of approaching ones and directed to a point in front of the door;
- (2) the controller detects the first avatar that reaches the door within the `openDistance` distance, recognizes it as the opener, and starts the `openTime` measurement;
- (3) while monitoring all avatars from the approaching list, the controller stops them when they cross the `waitingDistance` (another internal parameter of the controller, with a value greater than `openDistance`) or when it detects their immobility (meaning that the avatar has reached the waiting group at the door); in both cases, the avatar is moved from the upcoming list to the waiting list;
- (4) after the `openTime` expires, the animation of opening the door leaves is replayed and then the opening person is released, starting the procedure of liberating the waiting avatars;

- (5) the first avatar on the waiting list is released, i.e. directed to its next waypoint and transferred to the third list when its predecessor has covered a certain distance;
- (6) monitoring of released avatars must detect blockages: the agent of each immobilized avatar (except the one closest to the door) is stopped, its figure pulled back by about $\frac{1}{4}$ of its size to allow others to pass, and the avatar itself is moved back to the top of the waiting list;
- (7) when the released avatar successfully moves past the door, it is removed from the list and the control returns to its `NavMeshAgent`;
- (8) after emptying the list of released avatars, the animation of closing the door leaves is executed.

In step (3), detecting and stopping blocked avatars proved to be necessary to avoid avatars jostling in the group waiting for the door to open. For simplicity, each of them stands in the place they managed to reach when approaching the door, which makes their spatial distribution realistic and dependent on navigation history.

Successive activation of standing avatars in step (5) turned out to be crucial in the organization of the transition. Simultaneously releasing them all at once and relying on AI algorithms for autonomous navigation would be the simplest solution, but it did not work at all in practice. The most effective mechanism turned out to be waiting for the moment when the released predecessor managed to cover a certain distance – this solution took into account the cases of short-term blockades solved by AI, and it worked better than adopting strict time intervals. Equally important was also monitoring released avatars and reacting to (very frequent) instances of blocking each other's path in step (6).

5. Scripting the scenario events

Because the condition imposed by the runtime environment was to include a complete description of the simulated scene in a JSON file, it was necessary to specify also the scenario through a set of simple text fields, which had to express the whole variety of scenario states, conditions and possible actions. Additionally, the assumed goal was to execute the scenario directly by the Unity engine, without involving additional external tools, in a manner similar to its internal scripts written in C#.

The developed solution was based on the elementary concept of a deterministic Finite State Machine (FSM, e.g. Hopcroft *et al.*, 2000), whose description (states, transitions between them, transition conditions and generated responses) can be decomposed into components corresponding to the fields of the configuration file. The FSM concept is intuitive, does not require knowledge of programming principles from the script author and naturally expresses the parallel flow of various scenario threads. Particularly important was the need to enable concurrent descriptions: it is fundamental in specification of the simulation process but obtaining it in sequential programming languages is problematic and often based on mechanisms non-intuitive for users who are not programmers.

5.1. Specification of state machines

The operation of a state machine boils down to testing the transition conditions, switching between the states and generating output signals, i.e. commands sent to game objects. The input JSON configuration file describes each machine as a separate `Script` object; there can be any number of them in the file and their order is irrelevant because they all work concurrently and as peers. Due to the fact that the machines can test each other's state (although they cannot force transitions), it is left to the author's decision to appoint one of them as the main manager whose state will determine the behaviour of the others.

The `Script` object is a list of FSM states (graph nodes), each of which is defined by four fields:

- `name` (unique within a given FSM);
- `onEntry`: operations to perform when entering the state;
- `transitions`: a list defining all transitions coming out of a given state;
- `postExit`: operations to perform when leaving the state.

The order of the states in the list is not important except for the first one, which is taken as the initial (reset) state. Each element of the `transitions` list, in turn, defines one transition between states (graph arc) and is described by three fields:

- `condition`: a logical condition causing transition;
- `switchToState`: the name of the target state;
- `onTransition`: operations to be performed at the time of transition (i.e. before the current state changes its value).

The above simple set of 7 fields is a complete specification of the description grammar; apart from name and switchToState, which are plain identifiers, the fields onEntry, postExit and onTransition are text strings interpreted as commands addressed to game objects, and condition – a Boolean condition that tests their state. This is the way how both the FSM structure (the graph) and interactions with objects appearing on the simulated scene, i.e. testing their state and sending control commands, are mapped in the hierarchy of JSON fields.

From the point of view of automata theory, the onEntry field corresponds to generating Moore type output, and onTransition – Mealy type. The postExit field is formally redundant and was introduced as a simplification for the frequent case when certain Mealy operations are common to all transitions leaving a certain state.

5.2. Processing the description

While translating the description of the machine states into an internal representation in C# (native to the Unity environment) could be solved using typical dynamic data structures representing the graph, the task requiring an original solution was writing the FSM commands and conditions so that they could be interpreted by the simulation engine. The overall scheme of the implemented idea is shown in Figure 5.

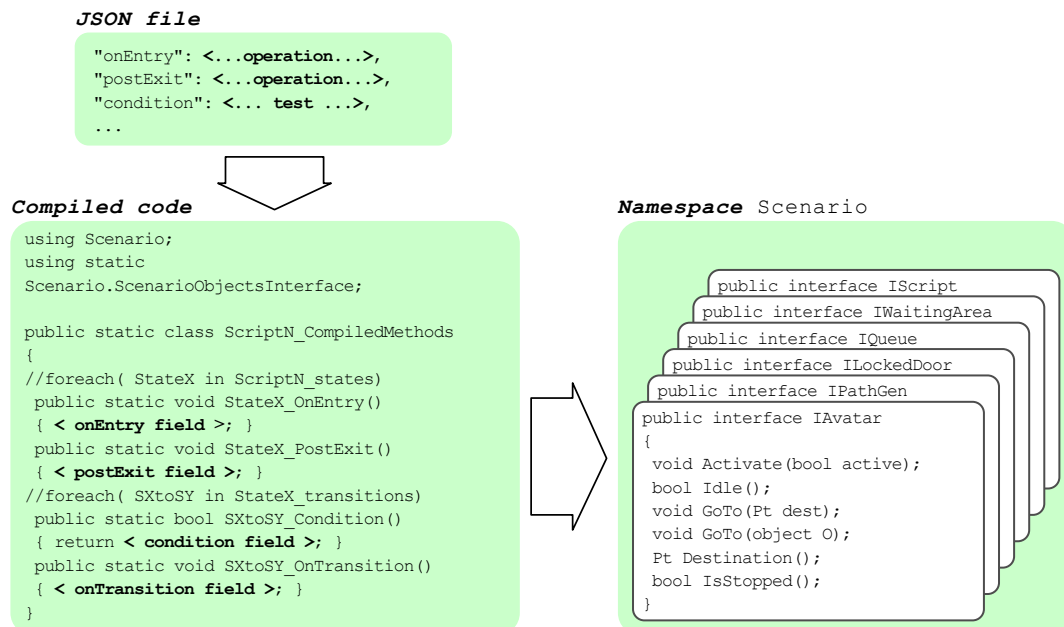


Figure 5. Access to game objects granted to external FSM code describing the scenario

The preparation of the machine for work and then its runtime simulation is managed by instances of the ScriptExecutor class. Using an appropriate C# syntax-compliant template, the fields from the JSON file are placed as bodies of static methods. For each FSM machine, the source code of the ScriptN_CompiledMethods class is created (see Figure 5), which for each state defines two void methods (onEntry and postExit) and for each transition void + bool methods (condition and onTransition). During scene initialization this code is compiled into memory, and ScriptExecutor remembers pointers (*delegates* in C# terms) to the methods, thus creating a complete dictionary of functions needed to simulate machine operation. The compilation takes place in the Mono environment, which for this purpose had to be added to the simulator distribution as one of its resources, because the C# environment created by Unity for the final application does not include the functionality of runtime compilation.

After preparing the executable code, the operation of the FSM consists of the following steps:

- (1) initialization: take the first state described in the file as the current one and perform its onEntry operation (i.e. call the respective delegate);
- (2) in each simulation frame: browse the items of the transitions list for the current state
- (3) if the transition condition is met:

- (4) call the associated `onTransition` operation
- (5) assign a new current state
- (6) perform the `postExit` operation from the previous state
- (7) execute `onEntry` from the new current state
- (8) return to (2) waiting for a new simulation frame
- (9) if none of the conditions were met, go back to (2) and wait for the next simulation frame.

The transitions are tested in step (2) in the order they were included in the JSON file. Since their conditions do not have to be disjoint, this order can be significant, as the first transition will be performed whose condition evaluates to `true`.

In the Unity environment, the order of activating objects in a given simulation frame is not specified, hence – in the case of concurrent work of many FSMs operating in one scenario – the `ScriptExecutor` objects of different scripts will be launched in an unpredictable order. This fact should be considered if the machines should synchronize their work by testing each other's states; in such cases their switching should be organized accordingly, e.g. by introducing "almost instantaneous" sequences of transitions with states lasting one frame, i.e. with the transition condition always met.

As Figure 5 shows, the FSM code is compiled with only one namespace attached, `Scenario`, which alone determines its access to the simulation environment. In particular, without linking to the standard `System` or `UnityEngine` libraries, the code cannot access the computer file system or I/O devices and communicate with the Unity engine. Allowed access is limited to calling methods of the static `ScenarioObjectsInterface` class (which do not need to be preceded by the class name thanks to the `using static` directive). This class keeps dictionaries of all objects that are acceptable in scenarios (avatars, generators, doors, queues and waiting areas) and makes them available by returning predefined interfaces: e.g. a static method `Avatar(string name)` returns an `IAvatar` interface to the avatar found with the given name, and so on.

5.3. The language

In the JSON text, the code specifying FSM operations and conditions begins with a reference to the object's interface through its name and then calls one of the component methods. Although the typical for object-oriented programming notation `object.method()` is used here, the author of the script does not need to understand the particular mechanism behind it. For example, the string:

```
Avatar("Av1").GoTo(2.5, 3.0)
```

in the `onEntry` field means that on entering the respective state the avatar "Av1" will be directed to the point with the given coordinates, while a similar command:

```
Avatar("Av1").GoTo(Door("Exit7"))
```

– to the door "Exit7" (where `GoTo()` is the member method of the `IAvatar` interface, overloaded for different arguments). In this way, the commands stored in the JSON file can be transferred directly to the C# code, which is then natively executed by the simulation engine at runtime, while the mechanism of browsing the dictionaries and calling the appropriate delegates in C# remains hidden from the script author.

The direct use of the C# language syntax allows to construct JSON fields using its additional capabilities, e.g. one field can contain an arbitrarily long list of instructions separated by semicolons and complex logical conditions can use logical conjunctions (`&&` as AND, `||` – OR, etc.) or comparison operators. For example, a condition starting some particular thread of scenario can look like:

```
WaitArea("Platform").Count() > 5 && Avatar("Burglar").IsIdle().
```

Static methods of the `ScenarioObjectsInterface` class and the set of interfaces it uses define the basis of the scenario language. The basic operations begin with accessing the corresponding object identified by its name which is done with the following methods:

```
public static IAvatar Avatar(string Name);
public static IPathGen PathGen(string Name);
public static ILockedDoor Door(string Name);
public static IQueue Queue(string Name);
public static IWaitingArea WaitArea(string Name);
public static IScript Script(string Name);
```

These methods return an appropriate interface whose components determine the set of operations that the scenario can perform with an object of a given type. Without tedious listing the complete member lists of these interfaces, their capabilities can be summarized as follows:

- With the `IAvatar` interface one can check the avatar status (navigating or idle), read its target coordinates or assign new ones, and direct it to a specific queue, door or waiting area;
- `IPathGen` allows to read the current number of avatars moving along the route, access an individual avatar (based on its sequential number), stop or resume generation, or trigger a single generation of a given number of avatars;
- `ILockedDoor` and `IQueue` interfaces allow only to add an avatar to their list;
- Using `IWaitingArea` one can add an avatar to the group, release all or read their current number;
- The `IScript` interface allows only to check if a machine is in particular state, while any other affecting its operation is not possible.

In addition to the mechanism “access a game object – perform an operation on it”, some auxiliary resources are available in the scripts to facilitate the construction of scenarios:

- reading the console of the computer running the simulation (i.e. detection of keystrokes),
- setting timers (e.g. for controlling the temporal switching of states),
- pseudorandom number generation,
- sending NATS messages and notifying of received ones.

With such a mechanism for the translation of commands stored in the configuration file, extending the scripting language with additional constructs is very simple and consists only in modifying the C# methods of the appropriate interfaces. Although the “access interface – call its method” mechanism seems limited at first, the flexibility of C# syntax allows to describe quite complicated situations. For example, a command that directs an avatar to the same destination as another one can use the returned position as an argument:

```
"Avatar("Supervisor").GoTo( Avatar("Patrol").Destination() )"
```

Or as another example, a complete JSON definition of a state that sends an avatar to some door after a random $5 \div 30$ second delay or immediately after the operator presses the ‘S’ key, can be expressed as:

```
name: "DelayedInit",
onEntry: "Timer(0).Start(Rand(5,30))",
transitions: [
  {
    condition: "Timer(0) || Key('S')",
    switchToState: "AvatarStarted",
    onTransition: "Avatar("Intruder").Activate();
                  Avatar("Intruder").GoTo(Door("MainEntrance"))"
  }
],
postExit: ""
```

6. Conclusions

The AI algorithms provided by the Unity environment proved to be very convenient in simulating the simple movement of people, undisturbed by dynamic objects, but they turned out to be insufficient in scenes involving a larger number of avatars grouped in an area where their harmonized action was required. The cooperation of many navigation agents agreeing with each other, for example, the order of the passage through the door, turned out to be too weak to avoid traffic blockages. Such impasses (when a group of avatars froze around a point where only one of them could go) were sometimes resolved after a few seconds by the AI algorithms, but usually lasted much longer, interfering with the movement of other avatars and completely stopping the flow of the scenario. For this reason, it was necessary to implement additional controllers that, in addition to handling the general operation of queues and doors, had to constantly monitor the behaviour of the involved avatars. The paper presented detailed procedures used for guiding the avatars through consecutive steps required for successful passing queue or door objects. The applied methods for detecting and removing blockages could be more effective (and also simpler) if Unity's `NavMeshAgent` had provided diagnostic data about the present state of navigation, in particular about objects that are currently blocking it.

Another task that required an original solution was expressing the simulation scenario in the form of textual commands and conditions which could be included in the configuration file and then interpreted by the simulation engine. In order to avoid creating a new description language from scratch or using an external interpreter, the adopted solution considered the fields of the configuration file directly as C# code fragments which, incorporated into a complete definition of a static class, were compiled into memory during scene initialization, and executed when the state machines representing scenario behaviours were simulated. The key aspect here was to ensure automatic translation of code fragments, which for the author of the script should remain commands and conditions describing flow of the actions, into the C# code accessing classes representing all scene objects through appropriate interfaces. Typing the C# code directly into JSON fields facilitated experimentation during the development of the scenario description language and this was an important advantage when working on the prototype of the application. On the other hand, the need to use the C# syntax forced the use of specific notation and imposed many restrictions – seemingly artificial for the author of the script who is not a professional programmer. However, this way many other inconveniences that would be caused by integrating and calling an external interpreter have been avoided.

The implemented method of scenario execution allows the code from the JSON file to directly access the game objects at the same level as the native C# code of the simulator. Although it resulted in a very high efficiency of execution, unattainable for external interpreters, such strong authorization created potential security hazards. For this reason, adequate attention has been paid to the use of C# namespaces and interfaces in a controlled and secure organization of this access.

Even in the natural language the exact specification of a simulation flow is a complex task that leads to lengthy and involved descriptions. The implemented concept of state machines worked very well in modelling the general course of action with its multiple parallel threads, but the need to describe a large number of details with them meant that the final specification in the JSON format was not only long, but also broken into many small fragments scattered in various fields. Navigating in such a description is not easy for a person without programming experience, and therefore the direction of further development should be the development of an external editor that would visually support the design of FSM machines, helping with graphic interpretation of the JSON hierarchy.

Acknowledgements

The paper presents results of a project supported by grant No. POIR.01.01.01-00-0235/17 awarded by the Polish National Centre for Research and Development (NCBR) within the framework of the Smart Growth Operational Programme 2014-2020 of the European Regional Development Fund (ERDF).

References

1. Anderson, E. F. (2011) A Classification of Scripting Systems for Entertainment and Serious Computer Games. In: *Third International Conference on Games and Virtual Worlds for Serious Applications*, 47-54. <https://doi.org/10.1109/VIS-GAMES.2011.13>.
2. Busetta, P., Robol, M., Calanca, P., Giorgini, P. (2017) PRESTO Script: scripting for serious games. In: *AI & Games Symposium at AISB*, 18-22.
3. Dijkstra, E.W. (1959) A note on two problems in connexion with graphs. *Numer. Math.* 1, 269–271. DOI:10.1007/BF01386390.
4. Duval, Y., Panzoli, D., Reymonet, A., Plantec, J.-Y., Thomas, J., Jessel, J.-P. (2015) Serious Games Scenario Modeling for Non-experts. In: *Proceedings of the 7th International Conference on Computer Supported Education Volume 1*. SCITEPRESS, 474–479. DOI:10.5220/0005489904740479.
5. Epic Games, Inc. (2023) Unreal Engine Documentation: Programming and Scripting. <https://docs.unrealengine.com/5.1/en-US/unreal-engine-programming-and-scripting>, last accessed 2023/01/13.
6. Hart, P. E., Nilsson, N. J., Raphael, B. (1968) A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2), 100–107.
7. Hönig, W., Kumar, T. K., Cohen, L., Ma, H., Xu, H., Ayanian, N., Koenig, S. (2016) Multi-agent path finding with kinematic constraints. In: *Proceedings of the International Conference on Automated Planning and Scheduling*, 26(1).
8. Hopcroft, J.E., Motwani, R., Ullman, J.D. (2000) *Introduction to Automata Theory, Languages, and Computation*, 2nd edn. Harlow: Pearson Education.
9. Majumder, A. (2021) Pathfinding and Navigation. In: *Deep Reinforcement Learning in Unity*. Apress, Berkeley, CA.

10. Qiu, H. (2020) Multi-agent navigation based on deep reinforcement learning and traditional pathfinding algorithm. arXiv preprint arXiv:2012.09134 ([https://arxiv.org/pdf/2012.09134 .pdf](https://arxiv.org/pdf/2012.09134.pdf)).
11. Rahmani, V., Pelechano, N. (2017) Improvements to hierarchical pathfinding for navigation meshes. In: *Proceedings of the Tenth International Conference on Motion in Games*, November 2017, Barcelona, Spain.
12. Slootmaker, A., Kurvers, H., Hummel, H., Koper, R. (2014) Developing scenario-based serious games for complex cognitive skills acquisition: Design, development and evaluation of the EMERGO platform. *Journal of Universal Computer Science*, 20(4), 561-582. DOI:10.3217/jucs-020-04-0561
13. Sugier, J. (2021) Modelling Pedestrian Behavior in a Simulator for a Security Monitoring System. In: *Theory and Engineering of Dependable Computer Systems and Networks. Proceedings of the Sixteenth International Conference on Dependability of Computer Systems DepCoS-RELCOMEX, June 28 – July 2, 2021, Wroclaw, Poland*. Advances in Intelligent Systems and Computing, 1389, 425-436. DOI:10.1007/978-3-030-76773-0_41.
14. Sugier, J. (2022) Scripting Complex Events and Behaviors in Computer Simulation of a Security Monitored Area. In: *Reliability and Statistics in Transportation and Communication. Selected Papers from the 22nd International Multidisciplinary Conference on Reliability and Statistics in Transportation and Communication, RelStat'22, 19–22 October 2022, Riga, Latvia*. Lecture Notes on Networks and Systems (in publication).
15. Unity Technologies (2023a) Real-Time Development Platform, <http://unity.com>, last accessed 2023/01/13.
16. Unity Technologies (2023b) Unity User Manual: Navigation and Pathfinding, <https://docs.unity3d.com/Manual/Navigation.html>, last accessed 2023/01/13.
17. Unity Technologies (2023c) Unity User Manual: Scripting. <https://docs.unity3d.com/Manual/ScriptingSection.html>, last accessed 2023/01/13.
18. Van Est, C., Poelman, R., Bidarra, R. (2011) High-level Scenario Editing for Serious Games. In: *Proceedings of the International Conference on Computer Graphics Theory and Applications (GRAPP-2011)*, 339-346. DOI:10.5220/0003374503390346.