# Automatic Generation of Interactive NPC Scripts for a Mixed-Reality Integrated Learning Environment

Andrew M. Yuan
Lawton Chiles High School
7200 Lawton Chiles lane
Tallahassee , FL32306
andrewyuan6@gmail.com

Fengfeng Ke
College of Education
Florida State University
3205C Stone Building
Tallahassee, FL32306
fke@admin.fsu.edu

Raymond Naglieri
College of Education
Florida State University
3205C Stone Building
Tallahassee, FL32306
rjn15b@my.fsu.edu

Zhaihuan Dai
College of Education
Florida State University
3205C Stone Building
Tallahassee, FL32306
zd12@my.fsu.edu

Mariya Pachman
College of Education
Florida State University
3205C Stone Building
Tallahassee, FL32306
mpachman@fsu.edu

## ABSTRACT
The Mixed-Reality Integrated Learning Environment (MILE) developed at Florida State University is a virtual reality based, inclusive and immersive e-learning environment that promotes engaging and effective learning interactions for a diversified learner population. MILE uses a large number of interactive Non-Player Characters (NPCs) to represent diverse research-based learner archetypes and groups, and to prompt and provide feedback for in situ teaching practice. The NPC scripts in MILE are written in Linden Scripting Language (LSL), and can be quite complex, creating a significant challenge in the development and maintenance of the system. To address this challenge, we develop NPC_GEN, an automatic NPC script generation tool that takes high-level NPC descriptions as input and automatically produces LSL scripts for NPCs. In this work, we introduce NPCDL, a language that we design for NPC_GEN to give high-level descriptions of NPCs, describe how NPC_GEN translates an NPCDL description into an LSL script, and report a user study of NPC_GEN. The results of our user study indicate that with minimal training, non-technical people are able to write and modify NPCDL descriptions, which can then be used to generate LSL scripts for the NPCs: the development and maintenance of NPCs is greatly simplified with NPC_GEN.

## CCS Concepts
**• Human-centered computing →Human computer interaction (HCI) →Interactive systems and tools • Software and its engineering →Software notations and tools →Context specific languages →Domain specific languages.**


*ICETC 2018*, October 26–28, 2018, Tokyo, Japan

ACM ISBN 978-1-4503-6517-8/18/10…$15.00
https://doi.org/10.1145/3290511.3290554

## Keywords
Mixed reality; virtual reality; OpenSimulator; Linden scripting language; interactive scripts; automatic script generation.

## 1. INTRODUCTION
Virtual Reality (VR) is a computer-generated 3D representation of a real-life environment. A user can autonomously navigate a VR in the form of a graphical representation, known as an *avatar*, and interact with simulated objects and other avatars via both verbal and nonverbal communications in real time and at the same pace the user would experience events in the real world [1]. VR has been implemented as a promising learning platform to support a variety of education activities [2]. The Mixed-reality Integrated Learning Environment (MILE) developed at Florida State University is a VR-based, body sensory technology integrated learning system that supports simulated and immersive teaching and mentoring practice [3].

Non-Player Characters (NPCs) play an important role in MILE, which is based on *OpenSimlator* (OpenSim), an open source, VR server software [4]. NPC scripts are written in Linden Scripting Language (LSL) [5]. To support a realistic learning environment, the functionalities of NPCs can be quite complex. Many NPCs in MILE are implemented with more than 1000 lines of LSL code. This posts a significant challenge for the MILE project since (1) the number of NPCs in MILE is large, and (2) the NPC functionality often changes due to design changes, user-test feedbacks, expert feedbacks, and scenario customization. As MILE modules are often customized after deployment, this also results in maintenance issues. These problems motivate the development of NPC_GEN, an automatic NPC script generation tool that takes high-level NPC descriptions as input and automatically produces LSL scripts for NPCs.

NPC_GEN uses NPCDL, a high-level NPC description language that we designed to describe NPCs. NPCDL specifies high-level NPC functionalities and omits implementation details. NPC_GEN takes an NPCDL description and automatically generates the LSL script that supports the specified functionality. NPC_GEN is publicly available[1]; and the NPC scripts that it

[1] http://hydra.cs.fsu.edu:9012/

generated have been deployed in MILE. In this paper, we present NPCDL, describe key techniques used in NPC_GEN to translate an NPCDL description into an LSL script, and report a user study for using NPC_GEN. The results of the user study indicate that with minimal training, non-technical people are able to write and modify NPCDL programs, which can then be used to generate LSL scripts for the NPCs: the development and maintenance of NPCs is greatly simplified.

The rest of the paper is organized as follows. Section 2 discusses the background and related work. Section 3 presents NPCDL. Section 4 describes key techniques in NPC_GEN to translate an NPCDL description to an LSL script. Section 5 reports our user study. Finally, Section 6 concludes the paper.

# 2. BACKGROUND AND RELATED WORK

## 2.1 MILE and a MILE Scenario

With a 3D VR environment and the body sensory technology, MILE promotes sensory and action *immersion* that enhances education by allowing multiple perspectives, situated learning, and transfer [6]. Another salient feature of MILE is *open-endedness*. MILE allows for adaptive construction and customization of simulated universes by both designers and end users, enabling a swift and adaptive design based on specific teaching needs. MILE also induces *symbolic* immersion by invoking situated, archetypical experiences of teaching: coaching, facilitating, and managing learners of diverse archetypes [6].

MILE supports both one-on-one and group training. A typical MILE training session has a human *facilitator* who leads the training session, and one or more *trainees*. The facilitator, trainees, and other human players navigate MILE as *avatars*. Both NPCs and avatars (played by other human players such as peers) can be agents to prompt and provide meaningful feedback for in situ teaching practice. They can personify or represent diverse research-based learner archetypes and groups. Human-controlled avatars will be able to perform reciprocal, authentic interactions with trainee during domain-specific instruction or coaching tasks (e.g., facilitate inquiry and problem solving without giving away answers). Interactive NPCs, on the other hand, may exemplify pre-timed or pre-conditioned behaviors to prompt the practice of targeted skills. NPCs participate in learning activities and play critical roles in emulating a realistic learning environment.

Figure 1 shows a screen capture of a MILE session for training classroom management skills. In the figure, the trainee is in the front of the class, giving the lecture; and the facilitator is in the background (not shown in the figure). The students in the class are NPCs. To emulate a realistic classroom setting, this session must have different types of NPC students such as disrespectful students, inattentive students, quiet students, and eager students. These NPC students give the trainee visual feedbacks to prompt the trainee to practice techniques to deal with different classroom situations and different types of students. For example, disrespectful NPC students may constantly chat in the classroom until the trainee intervenes with some classroom management technique such as calling the student to answer a question. Besides having different types of NPCs to simulate the realistic classroom dynamic, the training activities are also embedded in the interactions between NPCs and the trainee: the trainee learns and practices the necessary skills by interacting with the NPCs. For example, when an eager NPC raises hands, the trainee may call the NPC name; after that the eager NPC may ask a question and expect an answer; and depending on the answer from the trainee, the NPC may satisfy with the answer, ask more questions, or repeat the question again. By interacting with the NPCs, the trainee will learn the proper teacher behavior during lecture as well as classroom management skills. Such interactions are a part of the scenario design and must be supported by NPC scripts. Note that the session may be fully automatic with pre-timed and pre-scripted NPC students. The session may also be guided by a human facilitator who decides the proper training activities by issuing commands to the NPCs to start the activities.


**Figure 1. Screen capture of a MILE training session.**

## 2.2 Linden Scripting Language

Linden Scripting Language (LSL) is the language that all scripts in Second Life and OpenSim are written [5]. Unique features of LSL that are used in NPC scripts include the concepts of *state* and *event.* An event can be thought of as a trigger that leads the script to perform certain operations. LSL has a set of pre-defined events. Examples include a touch event that is triggered when an avatar touches an NPC, and a listen event when a message is received. Through these events, an NPC can react and interact with human avatars as well as other objects. Different *states* allow an NPC to react to an event in different ways.

Figure 2 shows an example LSL script. This script has two global variables (*npc* and *lis*) and two states, default and S1. In the default state (Lines (3) to (10)), when the object is touched, the *touch_start* function in Lines (5) to (9) is executed, which creates an NPC, makes the NPC sit on the object, and changes the state to S1. In state S1, when the NPC is touched, the *touch_start* function in Lines (13) to (15) is executed, which removes the NPC and changes the state to default. In state S1, the NPC also listens to the public channel (channel 0) (Line (12)). When it receives a "sound" message, the *listen* routine (Lines (16) to (19)) is triggered; and the NPC plays the sound file named "hello_sound". Here, the "sound" message can be viewed as a command to the NPC.

As illustrated in the example in Figure 2, an NPC script must specify many details in addition to the actions that the NPC performs. The details include giving NPC a name, creating channels to listen, maintaining the variables, controlling sound volume, etc. As a result, a complex interaction can take a significant amount of logic to implement in LSL. To support the designed learning activities in MILE modules, many of our hand-coded NPC scripts are more than 1000 lines of code, causing significant problems in the design, development, and maintenance of MILE modules. NPC_GEN and NPCDL are developed to address this challenge.

```
(1)  key npc;
(2)  integer lis;
(3)  default {
(4)      state_entry( ) { }
(5)      touch_start(integer num) {
(6)          npc=osNpcCreate("Zac", "Brown",
                                  llGetPos(), "app");
(7)          osNpcSit(npc, llGetKey(), ...);
(8)          state S1;
(9)      }
(10) }
(11) S1 {
(12)     state_entry () {lis = llListen(0, "", "", "");}
(13)     touch_start(integer num) {
(14)         osNpcRemove(npc); state default;
(15)     }
(16)     listen (integer c, string n, key ID, string msg) {
(17)         if (msg == "sound")
(18)             llTriggerSound("hello_sound", 1.0);
(19)     }
(20) }
```

**Figure 2. An example NPC LSL script.**

## 2.3 Related Work

It is well known that LSL scripts are tedious to develop. The LSL community has used multiple approaches to ease the LSL script development. One approach is to collect existing LSL scripts into libraries that are made publicly available. An example of such libraries is in Outworldz[2]. Another approach is to develop LSL script generators like what we do in this work. Existing LSL script generators mainly fall in two types. The first type contains LSL script generators to help novice LSL programmers start scripting. This type of generators in general has an easy-to-access interface and can generate simple LSL scripts. Examples include Script Me![3], Con Wylie's Script Generator[4], and Scratch for Second Life[5]. More advanced LSL script generators are domain specific. An example is the Particles LSL Generator[6] that generates a specific type of LSL scripts. Our NPC_GEN belongs to the last category, generating domain specific scripts, that is, interactive NPC scripts for MILE. There exist NPC scripts and NPC script generators such as those in Outworldz[7]. However, these existing NPC scripts and generators are not suitable for MILE. To the best of our knowledge, there is no existing LSL script generator that can support the flexible NPC interactions required by NPCs in MILE and supported by NPC_GEN.

# 3. NPCDL, A HIGH-LEVEL NPC DESCRIPTION LANGUAGE

As discussed in Section 2, from the perspective of the MILE design and development, only the high-level functionality of NPCs is important. The high-level functionality of an NPC includes what actions it can perform, what responses it can recognize and react to, what will trigger the NPC to perform an interaction (an interaction consists of a sequence of actions and responses), and what commands the NPC supports. On the other hand, the LSL script for an NPC must deal with many implementation details that are mostly of no concern to MILE designers. NPCDL, a high-level description language for MILE NPCs, bridges this gap. NPCDL omits implementation details and focuses on the high-level functionality of NPCs. The implementation details are automatically generated when an NPCDL description is translated into an LSL script. NPCDL specifies the following:

- **NPC states:** NPC states decide how the NPC reacts to triggers. NPC states in NPCDL are different from LSL states. NPC_GEN realizes each NPC state (described in NPCDL) using four LSL states as will be discussed in Section 4.2.
- **Interactions:** Interactions that an NPC can perform determine its functionality. An interaction consists of a sequence of **actions** and expected **responses**. NPCDL supports flexible actions that can be performed by the NPCs.
- **Triggers:** Each NPC interaction has a trigger, which is essentially a command in a communication channel that prompts the NPC to perform an action or to start an interaction. Each NPC in MILE has three pre-defined communication channels that can be used to receive commands (triggers), the state_control channel, the action_control channel, and the parameter_control channel. An NPC may also use a custom channel and the public channel (channel 0 in Opensim) to receive commands.

An NPC described by NPCDL may have one or more NPC states. Each NPC state may have one or more interactions with triggers. Figure 3 shows an example NPCDL program that will be used throughout this section as examples.

```
(1)  NPC twomoods begin
(2)    state default begin
(3)      channel -1 "-a1": action (random say ["Bor", "Yawn"]);
(4)      randomtime 20 30 : action (animation bored_ani);
(5)    end
(6)    state goodmood begin
(7)      channel -1 "-a1":  askaction (say "Can you check it?");
(8)        rbegin channel 0 1 ["yes"] :
(9)          action (say "Is it correct?",  sound is_it_correct);
(10)           rbegin channel 0 1 ["Yes"]:
(11)             action (say "Thank you.", sound thank_you);
(12)           rend
(13)           rbegin allothers:
(14)             action (say  "Ok, we will work on this.");
(15)           rend
(16)       rend
(17)       rbeginallothers:
(18)         action (say "We have a problem.")
(19)       rend
(20)     time 30 : action (random animation ["ani1", "ani2"]);
(21)   end
(22) end
```

**Figure 3. An example NPCDL program.**

The overall structure of an NPCDL description is as follows. Enclosed in the **begin** and **end** block are a list of NPC states. The bolded words in the description are reserved words in NPCDL. In the example in Figure 3, the *npc_name* is twomoods; and there are two NPC states.

[2] https://www.outworldz.com/cgi/freescripts.plx

[3] http://www.3greeneggs.com/autoscript/

[4] http://wiki.secondlife.com/wiki/Con-Wylies-Script-Generator

[5] http://web.mit.edu/~eric_r/Public/S4SL/

[6] http://particles-lsl-generator.bashora.com/

[7] https://www.outworldz.com/opensim/posts/npc/

**NPC** npc_name **begin**
NPC state *1*
……
NPC state *m*
**end**

## 3.1 NPC States

NPCDL describes an NPC state in the following form. The block enclosed by **begin** and **end** contains a list of triggers and interactions supported in this NPC state; and each trigger and interaction are separated by a colon (':').

**state** npc_state_name **begin**
Trigger *1* : interaction *1*
……
Trigger *m* : interaction *m*
**end**

There are two NPC states in Figure 3, Lines (2) to (5) for the NPC state named *default*, and Lines (6) to (21) for the NPC state named S1. In OpenSim, the NPC changes states by receiving commands in the form of '-gotostate:*state_name'* in its pre-defined state_control channel. For example, when the NPC described in Figure 3 receives a command '-gotostate:*default*', it changes its state to default.

## 3.2 Action

Each NPC state supports a list of interactions with triggers. An interaction consists of a sequence of actions and responses. NPCDL can specify flexible NPC actions. NPCDL allows a variety of **basic actions** that the NPC can perform, which include animation, sound, text message to different channels (including issuing commands to other NPCs), random animation (perform one random animation selected from a set of choices), random sound, and random text message, and the most generic user defined basic actions that is represented by a user-defined LSL routine. In addition, an **action** consists of a list of basic actions.

For example, *animation bored_ani* (Line (4) in Figure 3) is a basic action that performs animation file named "bored_ani"; *sound is_it_correct* is a basic action that plays the sound file named "is_it_correct"; *random say ["Bor", "Yawn"])* is a basic action that would say randomly either "Bor" or "Yawn". These basic actions can be combined to form an action with multiple operations. For example, Line (11) in Figure 3,

*action (say "Thank you.", sound thank_you);*

has two operations, saying a text message "Thank you" and playing a sound file named "thank_you*"*.

One common student action in the classroom or lab setting is that a student would raise a hand and wait to be called by the teacher. If the teacher does not call the student, the student may say "Excuse me" and continue raising the hand. If the teacher calls the student, the student will then ask a question (or perform an action). Since this is very common for MILE NPCs, NPCDL uses a special action called *askaction* to specify such an interaction. For example, in Line (7) of Figure 3, we have

*askaction (say "Can you check it?");*

This action starts by the NPC raising a hand and waiting to be called. After the NPC is called, it will perform the say operation to ask the question.

## 3.3 Response

In order for an NPC to carry on an interaction, it must be able to detect the responses to its actions. MILE is built over OpenSim, whose NPCs can only detect a small number of events such as receiving a text message, sensing an avatar touching, etc. Hence, the type of potential responses that NPCDL can use is restricted. To design the response in NPCDL, we considered all events that an NPC can detect in OpenSim, and decided to only support responses with text messages, mainly for the following reasons. First, most MILE modules require a facilitator. With a facilitator, all of the NPC detectable events other than text message can be emulated by a text message from the facilitator. For example, to emulate an avatar touching an object event, the facilitator can send a text message to the NPC to trigger the appropriate actions from the NPC. Second, the generic event sensed by NPCs are often too generic for the intended event. For example, when a response is for a particular avatar to touch an object. The generic touching event detected by the NPC cannot limit the touching event to the particular avatar (a text message from the facilitator does not have this issue). Hence, the responses in NPCDL are various types in text messages in different channels.

In NPCDL, when sending a text message without specifying a channel, it is defaulted to the public channel (*channel 0*). Each NPC in MILE has three pre-defined channels: state_control channel (*channel -1* in NPCDL), action_control channel (*channel -2*) and parameter_control channel (*channel -3*). Other channels can be directly specified. A response specifies the channel number (channel_num) for the text message, and the number of keywords expected from the list of keywords, the list of keywords. It has the following forms (See Lines (8), (10) in Figure 3 for examples):

**rbegin channel** channel_num num_keyword [keyword list]

… interaction triggered by the response (reactions to the response)

**rend**

or

**rbegin channel** channel_num string

... interaction triggered by the response (reactions to the response)

**rend**

The latter is a concise form for the commonly used case when the number of keywords in the response is 1. Line (8) in Figure 3

*rbegin channel 0 1 ["yes"]*

specifies a response from the public channel with one keyword "yes". An equivalent way to write this is *rbegin channel 0 "yes"*.

Another form for response handles the situation when a response is detected, but not matching any specified responses. The format is shown in the following.

**rbegin allothers**

... reactions to the response not matching any responses specified

**rend**

## 3.4 Interactions

An interaction is a sequence of actions and responses. NPCDL allows the number of potential responses to an action to be unbounded (can be any number) and the number of actions and responses in the sequence to be unbounded. This is achieved by (1) allowing each action to have any number of potential responses (see the following generic format for an interaction), and (2) each response recursively triggering a new interaction (see the response format in Section 3.3).

Action :
Response 1
……

Response n

Lines (7) to (19) in Figure 3 specifies an interaction that allows the NPC to perform the following interactions among others:

| Interaction 1: | Interaction 2: |
|---|---|
| NPC: Can you check it?<br>Trainee: Yes<br>NPC: Is it correct (with sound)?<br>Trainee: Yes<br>NPC: Thank you (with sound). | NPC: Can you check it?<br>Trainee: Yes<br>NPC: Is it correct (with sound)?<br>Trainee: No<br>NPC: Ok, we will work on this. |

## 3.5 Triggers

NPC interactions can be triggered by one of the two mechanisms: time triggered and command triggered. Each NPC state can have any number of command triggered interactions and one time-triggered interaction.

A time trigger has two forms: the fixed time trigger with the reserved word '*time*' followed by a number and the random time trigger with the reserved word '*randomtime*' followed by two numbers. Line (20) in Figure 3 *'time 30'* specifies that the interaction is triggered every 30 seconds. Line (4) in Figure 3 '*randomtime 20 30'* specifies that the interaction is triggered in a random time interval between 20 and 30 seconds.

A command trigger specifies the channel for the command and the command string. Its form is a *channel* reserved word, followed by a number specifying the channel number, which is then followed by a command string. Line (3) in Figure 3 shows a command trigger: *channel -1 "-a1":* the command is the string "-a1" on the state control channel. Facilitators, other avatars, and NPCs can use this command to trigger the NPC to perform the interaction.

## 3.6 An Example

MILE training modules are designed based on Q-matrix [7] that arranges the set of test items according to the specific subset of attributes measured by each individual item. For each item at the lowest level of the Q-matrix, one or more activities are designed. One activity designed for our physics lab scenario consists of the following steps:

1. The NPC raises a hand
2. If the trainee ignores the hand-raising after sometime, the NPC responded with "Excuse me." and repeat. Otherwise if the trainee calls the NPC name, goto Step 3.
3. The NPC says "Hi, I think I need your help. The digital multimeter is broken."
4. If the trainee asks how the wires were connected, the NPC will respond "Yes, I misconnected the wires." If the trainee asks other questions, the NPC will respond "Yes, I have done it."

The NPC described in the NPCDL description in Figure 4 will support the NPC functionality in this activity. Note that a typical interactive NPC supports many of such activities.

In this example, the *askaction* in line (3) supports the raising hand, name calling, and question asking (Steps 1, 2, and 3; see the discussion in Section 3.2). Lines (4) to (6) support the response when the trainee asks how the wires were connected while Lines (7) to (8) cover the case when the trainee did not ask that question. Note that the response in Line (4) is rigid. The designer may choose to use a less rigid checking and change Line (4) to

*rbegin channel 0 1 ["wires", "connected"]:*

In this case, as long as the trainee's answer includes one of the keywords "wires" and "connected", it is considered that the trainee has asked the right question. It is a design choice to decide which method is more appropriate.

```
(1)  NPC example_NPC begin
(2)    state default begin
(3)       channel -1 "-activities1": askaction (say "Hi, I think I
                 need your help. The digital multimeter is broken.");
(4)        rbegin channel 0 1 ["How the wires were connected?"] :
(5)          action (say "Yes, I misconnected the wires.");
(6)        rend
(7)        rbeginallothers:
(8)          action (say "Yes, I have done it.")
(9)        rend
(10)   end
(11) end
```

**Figure 4. An example NPCDL description.**

# 4. NPC_GEN

NPC_GEN takes an NPCDL program and translates it into an LSL program with the same functionality. Due the space limitation, we will only give a high-level overview about the translation and briefly describe key techniques used in NPC_GEN. NPC_GEN uses tools called flex and bison [8] to perform lexical analysis and syntax analysis. After that, NPC_GEN uses the syntax-direct translation technique [9] to perform semantic checks and translate the NPCDL program into an LSL script.

LSL scripts have a special format: all global variables must be declared before all subroutines, which in turn must be placed before all LSL states. To generate such LSL scripts, we first use syntax direct translation to collect all specified information in the NPCDL program into internal data structures. After the complete NPCDL program has been successfully parsed and all information has been collected, the whole LSL script is generated based on the internal data structures.

## 4.1 Code Generation

After NPC_GEN parses the program and collects all information, it generates the LSL script for the NPC. As discussed earlier, there are many LSL implementation details that are not specified by the NPCDL description. Such details include declaring and initializing variables, functions commonly used by all NPCs, setting up message channels, some control functions such as creating and destroying NPCs, common NPC functionality such as self-removing when being touched, and common commands that apply to all NPCs in all states. These details are not specified in the NPCDL program but must be included in the NPC LSL script. NPC_GEN pulls the implementation details information from a hand-coded NPC script template. As such, for these non-essential, but necessary features, all NPCs generated by NPC_GEN share common functionality and have a similar feel. For the common functionality, the generated NPC script can customize the NPC behavior by modifying the value of one variable, *myid*. NPCs with different *myid*'s will have different names, different pre-defined channels, etc.

The overall LSL script code generation sequence in NPC_GEN is shown in Figure 5. First, the file header is generated by the gen_header() routine; common variable declarations are generated by the gen_common_variables() routine: the generated code is directly extracted from the NPC script template. After that, gen_interaction_variables() converts the internal representations of interactions into LSL lists, generates the interaction variables,

and assigns the LSL lists (interactions) to the variables. After that, shared routines from the NPC template are generated by gen_common_routines(). Then, program specific routines are generated by gen_program_specific_routine(). Finally, the common and NPCDL program specific LSL states are generated.

```
1) gen_header();
2) gen_common_variables();
3) gen_interaction_variables();
4) gen_common_routines();
5) gen_program_specific_routine();
6) gen_common_states();
7) gen_program_specific_states();
```

**Figure 5. NPC_GEN code generation sequence.**

## 4.2 Program Specific States

The generated LSL script has 1 common state, generated by gen_common_states(), that performs the NPC creation function. For each NPC state in the NPCDL program, NPC_GEN generates four LSL states: idle, ask, interact_1, and interact_2. These four states are the key to support the flexible interactions. For all NPC states in a NPCDL program, the four LSL states are adapted from the same template. The idle state performs timed interactions if specified and waits for channel triggers. Once a channel trigger happens, if the interaction starts from the askaction (raising hand, waiting for name calling before performing the action and reaction), the NPC moves to the ask state that supports the functionality. If the interaction starts with a regular action, the NPC will perform the action and move to the interact_1 state. In the interact_1 and interact_2 states, the NPC checks if any responses have arrived based on the current interaction (stored in variable *curr_interact* in the generated LSL script). When a response arrives, the NPC checks which branch that interaction proceeds, and starts the new interaction (triggered by the response) by properly setting the *curr_interact* variable and moving to the other interact state. Note that interact_1 and interact_2 states are identical (except for the state name). It is necessary to have both since in LSL, moving to the same state has not effect and will not reset the variables and restart a new interaction (and move to another state will). We need these two states to deal with this feature of LSL and ensure that each response triggers a new interaction. Note also that the LSL logic in the interact states understands the internal representation of interactions and can proceed with the interaction properly.

## 4.3 Generating Triggers

The triggers for interactions are realized in a routine called process_state_specific_msg_*statename*(). This routine is invoked in all four LSL states that correspond to the NPC state named *statename* in the NPCDL description. NPC_GEN generates a list of triggers and the corresponding actions to start the interaction in this routine.

# 5. USER STUDY

We conducted a small-scale user testing of NPC_GEN with three non-programmer, education-majored designers including 2 females. A 25-minute orientation and explanation of NPCDL grammar was provided via a 6-slide PowerPoint presentation. Another 25 minutes were then used to explain three example NPCDL programs and to demonstrate their functionality in the OpenSimVR environment. The structure of NPCDL programs and the NPCDL concepts were explained and demonstrated. One of the examples contains a complex interaction with three response options and three levels of interaction.

After the 50 minutes training, the participants spent the rest of the 1.5-hour session developing an NPCDL program for an NPC with two two-layer interactions from scratch. Such an NPC is at a medium complexity level and is typical in our MILE modules.

All participants demonstrated a satisfactory level of comprehension within the 25-minute direct instruction of NPCDL. They self-reported 7.5 (out of 10) for their confidence toward using NPCDL.Their self-efficacy was even more obvious after the case demonstration, with frequent utterances like “ok,” “aha,” or “that’s easy” during and after the case demonstration.

All three participants finished their programming tasks independently, in 30-32 minutes. All completed scripts are correct in grammar and design logic, except for a couple of minor typos. All generated NPC scripts were tested successfully in OpenSim.

# 6. CONCLUSIONS

We introduce NPC_GEN, an automatic NPC script generator. NPC_GEN allows NPC to be developed and modified by novice, non-technical people and significantly reduce the cost for developing and maintaining NPCs. We have placed NPC_GEN on the Internet (See footnote 1 in page 1) and encourage the community to explore the capability of the tool. Our future plan is to build on this work to automatically generate all scripts for the whole MILE scenario.

# 7. ACKNOWLEDGMENT

This research is supported in part by NSF under grant 1632965.

# 8. REFERENCES

[1] Ke, F., and Im, T. 2013. Virtual-reality-based social interaction training for children with high-functioning autism. *The J. of Educational Research*, *106*(6), 441-461.

[2] Hew, K. F., and Cheung, W. S. 2010. Use of three-dimensional (3-D) immersive virtual worlds in K-12 and higher education settings: A review of the research. *British Journal of Educational Technology*, *41*(1), 33-55.

[3] Ke, F., Lee, S., & Xu, X. 2014. Immersive learning in a Kinect-Integrated Virtual Reality Environment. Paper presented at *Asso. for Educational Comm. and Technology International Convention 2014*, Jacksonville, FL.

[4] OpenSimulator, http://opensimulator.org. [accessed: June 10, 2018]

[5] Heaton, Jeff. 2007. *Introduction to Linden Scripting Language for Second Life*. Heaton Research, Incorporated. ISBN-13: 978-1604390049.

[6] Dede, C. 2009. Immersive interfaces for engagement and learning. *Science*, *323*(5910), 66-69.

[7] Tatsuoka, K. K (1983). Rule-space: An approach for dealing misconceptions based on item response theory. *Journal of Educational Measurement*, 20, 345-354.

[8] Levine, John. 2009.*flex & bison: Text Processing Tools.* O'Reilly Media. ISBN-13:978-0596155971.

[9] Aho, Alfred V., Lam, Monica S., Sethi, Ravi, and Ullman, Jeffrey D. 2006. *Compilers: Principles, Techniques, and tools.* Addison Wesley, 2nd edition.