MODDING FOR EMERGENCE: USING CELLULAR AUTOMATA, RANDOMNESS, AND INFLUENCE MAPS IN THE SOURCE GAME ENGINE

A Thesis

by

BENJAMIN THEODORE BERTKA

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2010

Major Subject: Visualization Sciences

Modding for Emergence: Using Cellular Automata, Randomness, and Influence Maps in the Source Game Engine

MODDING FOR EMERGENCE: USING CELLULAR AUTOMATA, RANDOMNESS, AND INFLUENCE MAPS IN THE SOURCE GAME ENGINE

A Thesis

by

BENJAMIN THEODORE BERTKA

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	Vinod Srinivasan
Committee Members,	Philip Galanter
	Dennie Smith
Head of Department,	Tim McLaughlin

December 2010

Major Subject: Visualization Sciences

ABSTRACT

Modding for Emergence: Using Cellular Automata, Randomness, and Influence Maps in the Source Game Engine. (December 2010)
Benjamin Theodore Bertka, B.A., University of California at Santa Cruz
Chair of Advisory Committee: Dr. Vinod Srinivasan

Recent advances in the field of educational technology have promoted the re-purposing of entertainment-oriented games and software for educational applications. This thesis extends a project developed at Texas A&M University called Room 309, a re-purposed modification of Valve Software's Source Development Kit that models classroom scenarios to pre-service teachers. To further explore effectiveness in the area of re-playability, this work incorporates emergent game behaviors and environments using cellular automata, randomness, and influence maps within the existing non-emergent structure. By introducing these qualities game play is expected to become less predictable, thus increasing the effectiveness of Room 309 as a learning tool.

DEDICATION

To my family

ACKNOWLEDGEMENTS

I would like to thank my committee chair, Dr. Vinod Srinivasan, and my committee members, Dr. Dennie Smith, and professor Philip Galanter for their support throughout the course of this research. Special thanks to Tim McLaughlin, Douglas Bell, Nathan Bajandas, and Michelle Simms for their teamwork and ingenuity during the earliest stages of the original Room309 development. Thanks to the Valve Developer Community for their experience, knowledge and wisdom. Finally, thanks to my mother and father for their encouragement, and to my wife for her patience and love.

NOMENCLATURE

AI	Artificial Intelligence
CA	Cellular Automata
I/O	Input/Output
Mod	Modified Video Game
Modding	The Process of Modifying a Video Game
NPC	Non-player Character
SDK	Source Development Kit
Vignette	Map/Level in Room309

TABLE OF CONTENTS

CHAPTER		Page
I	INTRODUCTION	1
II	BACKGROUND, RELATED WORK, AND METHODOLOGY	5
	II.1. Background	5
	II.2. Related Work	8
	II.3. Methodology	11
III	DEVELOPMENT OF THE EMERGENCE MOD	13
	III.1. Behavioral Model	13
	III.2. Cellular Automata Model	15
	III.2.1. NPC States	16
	III.2.2. Neighborhood Influence	21
	III.2.3. Environmental Influence	23
	III.3. Pre-visualization	26
	III.4. Automata Entity	28
	III.5. Player Influence	30
	III.6. Randomizing NPCs	32
IV	VIGNETTE IMPLEMENTATION	35
	IV.1. Cheating Vignette	35
	IV.2. Cheating Mechanics	36
	IV.2.1 Cheating Thwarted!	39
	IV.3. Statistical Feedback for Players	40
V	ANALYSIS OF THE EMERGENCE MOD	42
	V.1. Testing the Model	42
	V.2. Randomization Evaluation	45
	V.3. Environmental Influence Comparison	46
	V.4. Modified Cheating Evaluation	49
VI	CONCLUSIONS	50
	VI.1. Summary	50
	VI.2. Future Work	52
REFERENCES		54
APPENDIX A		56
APPENDIX B		67

Page

APPENDIX C 78

VITA 85

LIST OF FIGURES

FIGURE		Page
1	A goal map for a simple game with multiple paths to win	1
2	Teacher's perspective while inside Room 309	9
3	Animated behaviors for an NPC student	15
4	NPC neighborhood and influence values	22
5	Neighborhood types	22
6	Screen shots of three main influential regions in Room 309	24
7	An environmental influence map for Room 309	24
8	Calculated influence values (A), for influence map (B). Colors from the influence map are interpreted as behavioral states.	26
9	Pre-visualization tool used to test the model during development	27
10	Flow map describing information transfer with the automata entity	29
11	automata_logic entity placed in the editing environment	30
12	A trigger box placed around the NPC can detect the player's presence	31
13	Increased texture pool for male NPC	32
14	Texture pool comparison	34
15	Cheating phases 1-4	38
16	Decision panel for the cheating vignette	40
17	Screenshot with various statistical data displayed as text	41
18	Evolution without an environmental influence.	43
19	Evolution with an environmental influence	44
20	Students with blue shirts clustering together	45
21	Environment influence maps A-E used for testing the CA model	46
22	Pre-visualization outputs A-E	47

LIST OF TABLES

TABLE		Page
1	The various behavioral states and corresponding values.	17
2	State changes for aging states.....	21
3	Possible cheating sequences.....	37
4	Influence map comparison from the first 100 seconds of game play.....	47

CHAPTER I

INTRODUCTION

Modern video games are complex, usually consisting of elements such as story, graphics, and game mechanics that make up the structure of the game, working together to shape experiences for the player. Emergent game systems enhance player experience by creating variation in the gaming environment so that each time the game is played, the initial conditions can be the same, yet variations during play, as well as the game result, remain unpredictable. While the objective of game play may be to win the game, the ways to the win may change depending on the events unfolding during the game. Some events may originate from the player's actions, and some from the game itself. Figure 1 shows a goal-map with paths depicting unique story lines and various paths fulfilling major goals, leading to a winning outcome. Path intersections represent possible decision-making points during which the player determines the next steps,

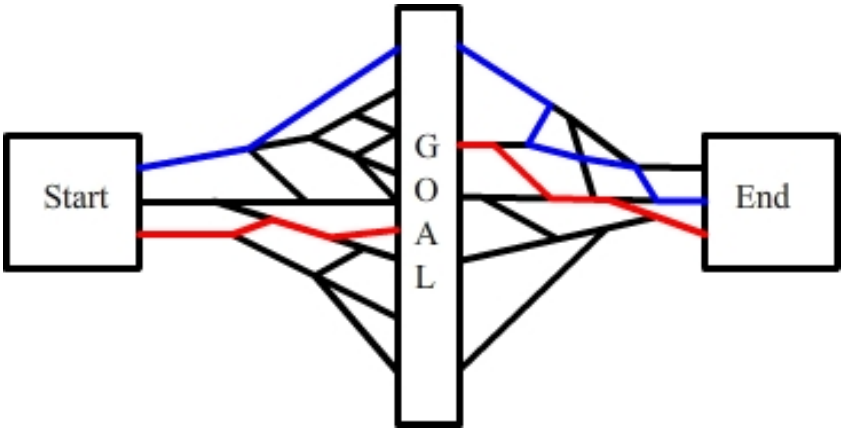


Fig. 1. A goal map for a simple game with multiple paths to win.

This thesis follows the style of the *ACM Transactions on Computing Education*.

each with the possibility of reaching a major goal, or win.

Emergence describes the properties, behaviors, and structures that occur at higher levels of a system, which are not predictable at lower levels [Sweetser et al. 2003]. Games with little or no emergence usually have very linear stories and predictable paths that lead to a win, after which the player may lose interest in continued game play. Creating a game with significant emergence can make the path to win unpredictable, altering the story, such that the game can be enjoyed repeatedly. One of the goals of this thesis is to use an existing educational game and introduce aspects of emergence to create variation within the game, creating unpredictable behaviors amongst non-player characters (NPCs).

The creation and development of video games occurs with a game engine, or software system to drive graphics and game play. Some games are developed as original works utilizing a custom game engine and unique visual assets used only once. Other games are developed using popular game engines, with borrowed features from other games, slightly altering the visuals, characters, and/or story lines. While the game engine remains unchanged, the modifications feel and play like a new game.

Amongst players and developers, modified games are known as mods, where the act of developing a mod is known as modding. Simple mods require the original game to be present for it to run because graphics, sounds, and characters are being reused or expanded upon, with the original plot, story and game play unaltered. A total conversion is a modification where a new game is created using only the essential features of the game engine, replacing story, assets and game play, thus creating an entirely new game. One popular mod is the first-person shooter game Counter-Strike, a total conversion of Valve's Half Life 2 that has become more popular than the original game.

While modding is common practice for creating entertainment-oriented games, it can also be a good technique for making educational games. Game engines like Epic Games Unreal Development Kit [Epic Games 2010] and the Valve® Source™ SDK [Valve Corporation 2010] are known for providing good graphics and extendable source code. These engines are publicly available and include scripting capabilities, which

allow for an easy implementation of new assets, and modifications to story. While both engines have game play geared towards action-combat, changing the fundamental building blocks requires complicated low-level modifications to the source code. Creating custom entities for the level editor that work in conjunction with existing entities can provide an easy, lower-level mod, such that the desired features are easily integrated into the design and development process.

Room 309 is the title of one such mod that builds on the functionality of the Source engine, while introducing new game play and an entirely new aesthetic. Originally conceived as a tool to train pre-service teachers, the principle investigators were interested in creating a virtual classroom full of students that the player could interact with to practice classroom management skills. Experts in the field of education worked together with game developers to create scenarios found in a typical classroom for a pre-service teacher to make real-world decisions in. While successful as a game mod, Room 309 was not engaging as a re-playable learning tool. The work of this thesis extends upon that modification, further modifying Room 309 to introduce emergence through cellular automata, randomness, and influence maps.

This project creates a cellular automata (CA) behavioral model in order to control the non-player characters (NPCs) within an educational game for pre-service teachers. The game takes place in a virtual classroom, in which the NPCs are the students, who exhibit behaviors and reactions. The player, in the role of teacher, must recognize the emergent behaviors and respond correctly in order to progress through the game. To make the game more engaging, students in the classroom must behave realistically, with meaningful animation sequences that vary in subsequent plays. One approach to creating this variation is to use level building, or mapping, to manually create numerous copies of the same level, or map, where the same NPCs are programmed with different behaviors. Random maps for a particular level can then be invoked by the game during successive plays. While this maximizes the control for the design process, creating a map for every possible behavioral combination would be

costly and tedious. A more efficient approach is to create one map, and use cellular automata to govern the NPC movements.

Cellular automata (CA) is widely studied in mathematics, physics, and many other fields. It is a model that is best understood as a grid of cells, in which each cell exists in one or more states, and is responsive to neighboring cells as dictated by rules. In a CA simulation, each cell is updated depending on what the rules dictate, with simple rules often resulting in complex output.

Superimposed on a virtual classroom, each cell in the CA model represents one NPC student sitting at their desk. In this approach the NPC students may have a variety of different states, such as alert, distracted, or tired. Using CA to influence typical behaviors of students will enable the same classroom to be used for multiple scenarios, or vignettes, in which NPCs are seated at desks, yet responsive to each other and the player, without the need to create more than one version of a map.

This approach implements the CA model into a stand-alone visualization used to tune and test the model before a final transfer of functionality, or port, to the Valve® Source™ engine to help create an emergence mod of Room 309. The model is tested within a prototype-cheating vignette and evaluated for its performance.

Chapter II provides necessary background for the key topics regarding linear games, emergent games, artificial intelligence, NPCs, cellular automata, and influence maps. Also a snapshot of related work is given with a methodology of how these subjects and ideas fit into this research. Chapter III details the development process for the emergence mod of Room 309. Chapter IV discusses implementation of the game mechanics for a modified cheating vignette. In Chapter V, analysis is performed on the CA model, randomization scheme, environmental influences, and cheating vignette. Chapter VI concludes this paper summarizing this work and its significance, discussing room for improvements, and outlining possible directions for future work.

CHAPTER II

BACKGROUND, RELATED WORK, AND METHODOLOGY

II.1. Background

Linear games are centered on what a game designer has planned out for the player. Games with tight choreography in story and game mechanics are considered linear games. Scripted cinematic cut-scenes between key moments or player accomplishments don't change much upon successive plays, and the player isn't an active part of the story. Non-player character (NPC) movements usually happen in the same place every time. Therefore, to beat a linear game a player must only play the game often enough to memorize where all the enemies and goals are. Linear games include side-scrolling games such as Super Mario Bros and Sonic the Hedgehog, as well as first person shooters such as Doom and Star Wars: The Force Unleashed. In Super Mario and Sonic, the player has no option to follow more than a pre-set path to win the game. In Doom and Star Wars, mazes and physics make the game environment feel more open. Linear games like these allow some exploration by the player, but the path to win is already set when the game begins. The entire experience for a player can feel scripted, and pre-planned.

In contrast, a game environment can be emergent if simple rules for interactions between game objects, player, and environment can yield unpredictable results. The book *Emergence in Games* characterizes emergence as local and global emergence [Sweetser 2008]. Local emergence occurs when a small area in the game world allows for unpredicted behavior that doesn't affect the rest of the game, while global emergence occurs when simple rules for interactions of game elements affect the entire game world. Popular games with local emergence are The Sims, and Half-Life 2 where NPCs in the game can be affected by the environment. Popular games with global emergence include The Elder Scrolls IV: Oblivion, and World of Warcraft. In Elder Scrolls, the player is

able to explore an expansive game world and constantly have new experiences, however the player's decisions do not affect the final story or outcome of the game. World of Warcraft has an expansive game world a player may carve their own story in, however the game world provides an entire economy where in an auction house a player is able to affect the entire game for all users. While a totally emergent game world may not exist yet, results of adding emergent qualities to games help create enjoyment and re-playability.

One of the main ingredients in a game is its artificial intelligence (AI) which helps create intelligent looking NPCs and good game play. Sweetser's work on emergence describes agents as decision-making entities in games that sense and react to the game world [Sweetser 2008]. Agents can query the state of the environment with probing, and the environment can broadcast its state to the agent. Probing involves the agent querying the state of the environment, while broadcasting involves the environment sending its state information to the agent.

The book, *AI and Artificial Life in Video Games* [Lecky-Thompson 2008] gives a comprehensive breakdown of various types of game AI where general types of AI are classified as movement, planning, and environmental. Movement AI is the manipulation of NPCs within the game universe, while planning AI is a high level control of behavioral threads. Interaction AI is the management of the interface between game universe, entities, and player. Environmental AI controls the appearance of the game universe toward the player. Along with the various types of AI, Lecky-Thompson [2008] also provides an overview of popular AI paradigms with examples on ways of implementing AI. Of the various types discussed, the most common are simple statistical analysis, finite state machines, alpha-beta pruning, A* heuristic search algorithms, neural networks, expert systems, fuzzy logic, and stochastic hopefield nets. This work is most focused on the finite state machine AI used for modeling behavior when every possible state and results of transitions between states are either known or can be determined.

Sweetser [2008] discusses the similarities between a fuzzy state machine and a finite state machine. While a finite state machine selects a particular state, fuzzy state machines use a sliding scale between states, where a state can lie in the continuum between extreme states. An analogy is with animation blending where an NPC can run more than one animation at once, such as bouncing due to being hit by bullets while running.

Funge and Millington outline multi-tiered AI in *Artificial Intelligence for Games,* where the AI is broken up into various levels that allow for tactical AI [Funge and Millington 2009]. An NPC may be controlled by the player, but also interact with other NPC teammates, and simultaneously watch out for enemies and other obstacles. Their approach to AI is either top-down or bottom-up. Top down AI requires the NPC to first know about the global state of the environment and then use that information to control the NPC. Bottom-up allows the NPC to first be autonomous and take the game state as a second consideration.

Cellular Automata (CA) models are usually either one or two-dimensional arrays of cells, where at discreet points in time a cell may be in one of any available states from a finite state set. The cells evolve at incremental time steps due to a local rule, which is the same rule for all cells in the array. The CA doesn't have any inputs, it just obeys a rule, so therefore it is said to be autonomous. Cellular Automata (CA) is a well-studied subject and Sarkar [2000] gives a complete overview of the various types of CA in a historical survey. Sarkar's work presents an up-to-date survey covering classical CA as well as more obscure types used by mathematicians and computer scientists.

Conway's game of life is a popular example of CA that is set upon a two-dimensional grid where cells are either alive or dead [Garnder 1970]. The purpose of his game was to model macroscopic behaviors of a population. The two-dimensional grid is a population of cells where each cell has up to a 3x3 square neighborhood of cells around it resulting in a maximum of eight neighbors, which are used in the local rule. By location, cells on the edge or corner of the grid have neighborhoods with less than eight neighbors, however all cells would have eight neighbors if the grid was placed on a

torus where opposing edges and corners meet one another. Conway's rule states that if a cell were alive, it would stay alive only if it had two or three neighbors that were also alive. If a cell were dead it would be reborn if it has three living neighbors. An alive cell would die of loneliness if it had zero or one neighbor, or die of overcrowding if it had four or more neighbors.

Briefly mentioned in Sarkar's article is a special type of CA called polygeneous CA. Polygeneous CA models allow each cell to have a different set of states that are able to interact with one another. This type of CA is discussed by Holland in Burkes' *Essays on Cellular Automata*, yet his work was published without mention of its potential applications or usefulness [Burks 1970].

Influence maps reside in the game environment to provide direction for game elements at certain locations. Sweetser and Wiles [2005a] combined influence maps with CA to help the character react to the changing environment, and look more intelligent to the player. Using influence maps with cellular automata was easy in their research because the influence map and cellular automata were able to share similar data structures, where the values from influence maps were easily accessed, updated, and used in local rule calculations for cells.

II.2. Related Work

Educational game mods developed with popular 3D game engines have been researched for their usefulness for teaching overviews on general subjects, as well as more narrow topics that teach specific skills. For example, using an iterative prototyping methodology, experts in software, learning technology, and food safety designed Serious Gordon to teach food safety skills using the Source engine [Mac Namee et al. 2006; Ritchie et al. 2006]. Serious Gordon was designed around the capabilities of the Source engine, making use of good graphics and scriptable NPCs, thus creating a useful learning tool, yet is predictable with no major variations in the path to win. Furthermore, in trials designed to determine which game engine would be the most effective to support a

surgical training purpose, Marks et al. [2007] found the Source engine to be superior to The Unreal and idTech4 engines. A mod was developed using each engine for creating an interactive environment with props, and game play to reach specific objectives. Custom assets were created, and pre-packaged assets from the game engines repurposed. Compared to the mechanics of other game engines, the Source engine excelled for its ease in scripting game play.

The scripting capabilities, pre-packaged assets, extensible source code, and high quality graphics of the Source engine made it a good game engine for modding educational games. For these reasons, the Source engine served as a starting place for a serious game project developed at Texas A&M University, as a collaborative project between the department of Teaching, Learning, and Culture (TLAC), and the Department of Visualization. In this project, the Source engine was used to create a virtual classroom, complete with animated students, in which the player takes on the role of teacher, playing from a first person perspective (see Figure 2).

Fig. 2. Teacher's perspective while inside Room 309.

The Room 309 game recreates scenarios presented in Dr. Dennie Smith's Decision Points video, in which students act out situations typical to a normal day in the classroom so that the audience of pre-service teachers can make decisions about

classroom management [Smith 1987]. Room 309 had game play that was scripted and linear to best match the video. In spite of the linear progression, recreating it in a virtual setting allowed an interactive learning experience. Room 309 became a pilot for test groups of experienced teachers, yet remained a simple game and not further developed.

Heavily scripted animations can make NPCs appear like robots and not very lifelike. Since the player needs to observe and interact with the students, a more lifelike NPC is necessary to create a better experience. In an effort to help developers make better AI, a study that asked questions on the importance of character behaviors measured realism, intelligence, social interaction, and ease of communication in first person shooter games [Sweetser et al. 2003]. It was found that players who prefer to play against humans do so because it is more challenging playing against someone who is unpredictable, smarter, harder to beat, and often results in a meaningful experience. Hence, it was determined that developing AI for the players who played against humans should be the goal for game developers.

In *The Perils of AI Scripting*, Tozour [2002] found that scripted AI often ends up generating very narrow game play, and suggests that adding randomness to the scripted AI would improve game play. Rather than being a test of skill, games with too much scripted AI end up becoming a test of memorization.

Creating an emergent game environment is unique in that the design process must be extra sensitive towards the control of the story and game environment. Sweetser and Wiles [2005b] showed that to create an emergent game system one would have to employ a different design process, and that a completely emergent system takes away creative control from the developers. In her Ph.D. thesis, Sweetser [2006] proved the value and validity of an emergent approach to game design. Her investigations included using CA to create an emergent environment, and showed that player enjoyment was positively affected in an emergent game world. Serious and educational games do not typically have emergent qualities to them. This is in part due to the different design process required to create an emergent game, as well as the pedagogical concerns of the designers. Kickmeier-Rust and Albert [2009] proposed a model for developing

educational games with emergent components, which allowed scripted control for the lesson designer, while using emergence to reduce the time it takes to develop various learning scenarios. In this model, they began work on an autonomous character to guide learning, however they found challenges with controlling an emergent story. Smith and Smith [2004] proposed a practical and iterative, emergent game play design process that could be easily employed by designers of all types of games. In this process, aesthetics are considered first, then technological considerations secondarily. The game rules are then planned and the world is populated with mechanics. Finally the entire game system is evaluated and the iterative development process begins.

II.3. Methodology

This work creates an emergent Room 309 by modifying the original game in order to create re-playability in an unpredictable and engaging environment. The AI for NPC students is redeveloped and randomness inserted into game play, so successive plays result in new experiences when starting a lesson, even if it is the same lesson. Cellular automata and influence maps are used to create local emergence where NPCs, the environment, and player may affect each other. Generic visualization of the CA model is employed for testing and tuning the model before transfer into the game environment. The modifications result in custom entities for the Source engine that are available to level designers. A cheating vignette is created where NPCs randomly cheat, and an evaluation is performed to determine the effectiveness of the game system.

The AI system for NPC students in the virtual classroom uses a bottom-up approach as detailed by Funge and Millington [2009], where NPCs are autonomous before considering the rest of the game environment. Here, a multi-tiered CA model is created for a local rule of the CA that considers the internal state of an NPC, its relationship with other NPCs, and the effect that the environment, including player, has on NPC behaviors. The AI to be effected is a special case of movement, where NPC students remain sitting at their desks, exhibiting behaviors and interactions with the

player and other students. The CA takes control of the students, probing their immediate neighborhood and considers the environmental influences upon them in order to determine their next state, or animation.

While Sweetser and Wiles [2005a] created a local rule within a CA model to dynamically update environmental influence maps, the modifications presented here use a local rule within a CA model to control behaviors of NPCs directly, with environmental influences dynamically updated outside of the local rule. In this model, the CA works as a finite state machine controlling behaviors; however, as the player broadcasts information to the environment by updating influence maps, the states of nearby NPCs are dynamically affected thereby creating local emergence in the classroom. Since these modifications utilize core functionality of the Source engine, an NPC may only run one animation at a time; hence, even if the behavior model decides an NPC should be in between states, or in a fuzzy state, the final visual result would still have to be a discreet visual representation.

In light of suggestions by Tozour [2002] for randomizing scripted AI, along with Sweetser et al. [2003] finding that better AI leads to more player engagement, NPCs in the emergent Room 309 are always randomized for appearance, behavior, and interactions so that game play improves from the original game. Implementation within a playable vignette includes randomizing the location of goals such that a player will rely on skill rather that memorization.

As in the work of Kickmeier-Rust and Albert [2009], an emergent Room 309 that makes use of level-editing entities will allow scripted control for the level designers and save time in creating new vignettes. The emergent game play design process detailed by Smith and Smith [2004] is followed, where the emergent Room 309 development repurposes assets from a linear educational mod. In creating a more emergent environment, the game system is planned out using a generic pre-visualization. The technical considerations of the game engine are taken into consideration and the generic model transferred and available within the level editing process. With a working game prototype, features may be improved or added during an iterative development process

CHAPTER III

DEVELOPMENT OF THE EMERGENCE MOD

To create an emergent Room 309, a multi-state cellular automata model which uses neighborhood and environmental influence maps is created to control NPC student behaviors. A pre-visualization tool is created for testing the model during development. Behavioral states are animated, entities created for level design, and the virtual classroom randomized.

III.1. Behavioral Model

Emergent Room 309 utilizes a discreet set of behaviors that approximate the visual actions and internal states of students in Room 309. These behaviors are simple, fairly easy to animate, and shared amongst students. Each NPC has the same set of behaviors as all the rest, with some behaviors linkable for creating interaction. The following set of behaviors will be used for both states in the CA model, and as animations for NPCs:

- Alert. By default, a student will appear in an alert, or attentive state. This is the ideal behavior for a student. NPCs in the alert state will pay attention to the teacher, or raise either hand.
- Working. Students busy with assignments, or taking notes will appear focused on their own work at their desks.
- Bored. A student may appear bored if the lesson is not interesting. Bored students will be slouched over their desk, with an elbow on the desk and their head resting on their hand.
- Tired. If a student is excessively bored or disinterested in class, they may appear tired. Tired students will be sitting low in their desk to help avoid being seen by the teacher.

- Distracted. Looking around at other students in the classroom is a characteristic of distraction. In this state a student will look at the left and right neighbors, or to the student behind. Instances of cheating and other forms of mischief may occur in the distracted state.
- Tapping. A student may tap their neighbor to try to get their attention, distracting them from the lesson. This behavior is used to begin a sequence of actions in the cheating scenario.
- Showing. A student can gesture to another student by showing or pointing to an answer on their paper. This behavior is triggered by a neighboring student tapping them to get their attention.

Each animated behavior in emergent Room 309 is a cycle that an NPC replays until the CA model controlling the NPC signals a change is necessary. Figure 3 shows screenshots from the various animation cycles made available to NPCs in Room 309. To create the behavior cycles, specific tools are used to generate animations according to Valve's specifications. This work creates the animations using the Autodesk® SoftImage® Mod Tool™ software [Autodesk 2010]. Using the Mod Tool™ software is the most efficient method of generating animations for Room 309 because it comes pre-packaged with tools that output the required file types used in the Valve® Source™ engine, is free to download, and has been developed specifically for modding games. Appendix B.3 contains an overview of creating Room 309 animations using the Mod Tool™ software and compiling character models for use in Valve's Hammer World Editor.

With a basic set of behaviors and animations defined, the behavioral model can be further divided into internal, collective, and visible states, which rely on both neighborhood and environmental influences.

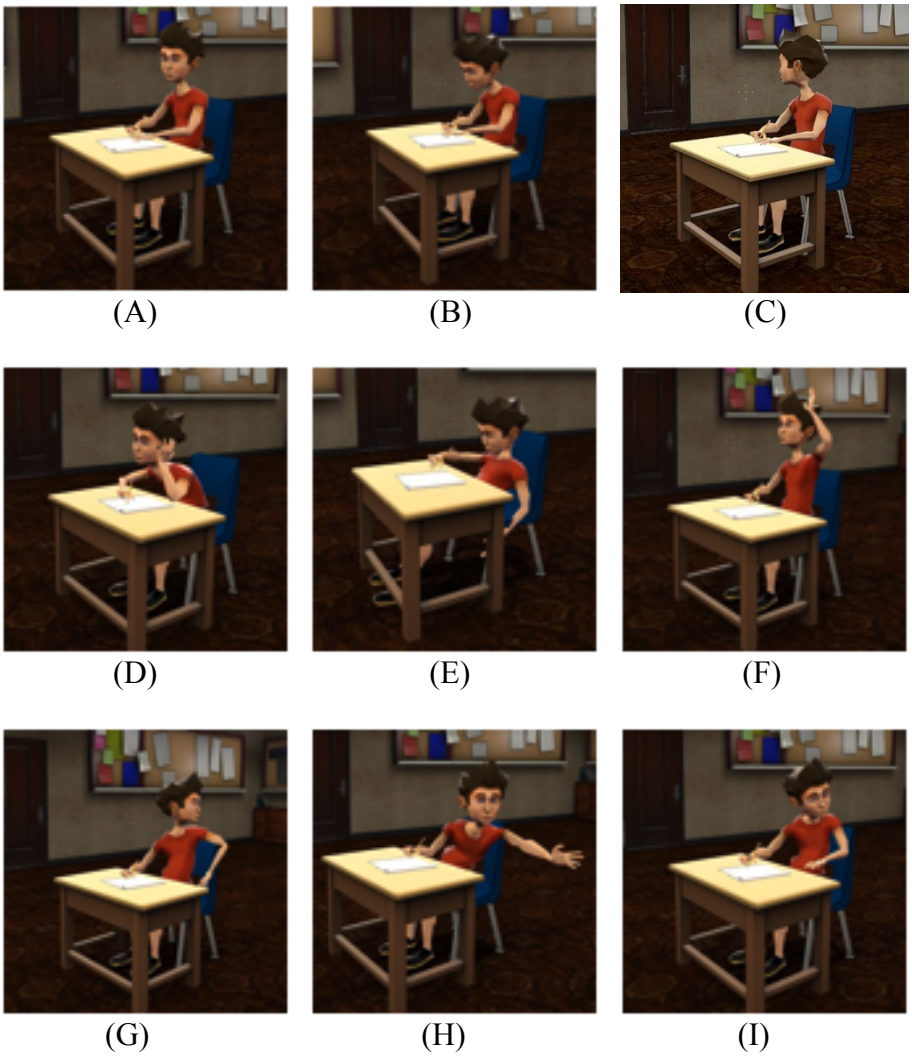


Fig. 3. Animated behaviors for an NPC student. (A). Alert. (B). Working. (C). Distracted to the right. (D). Bored. (E). Tired. (F). Raising left hand. (G). Looking back. (H). Tapping left. (I). Showing left.

III.2. Cellular Automata Model

Cellular automata was chosen as a method of controlling the behaviors of students in Emergent Room 309 because the students are always at their desks, similar to fixed cells populating a traditional cellular automata grid. Traditional implementations of cellular automata utilize the states of neighboring cells for state updating, as well as age

of a cell's own particular state. CA with aging states, or generations, is a variation in CA that keeps track of the length of time a cell is in a particular state, using this information in the local rule calculations. In emergent Room 309, NPC students can remain in a particular behavioral state for a period of time before they move to another state, so we use both aging states as well as neighborhood influence to compute the next state for a student. The seating chart or classroom environment can also have an effect on a student, such that any student seated next to a window will tend to look out the window. Environmental influence maps are used to give particular desk locations in Room 309 increased potential for a certain type of behavior. All together, neighborhood and environmental influences, with aging states are used in computing an NPC's visible behavioral state, and ultimately the final animation seen by the player.

III.2.1. NPC States

Traditional cellular automata states are either alive or dead, where living cells get older after the local rule has been applied to each cell in the grid. Age is a factor and ensures that if a particular state is held for too long a time, a change in state will occur. NPC students are always alive in Room 309, so binary cell-state representations do not apply. We now take the approach of thinking of each NPC as a student in a classroom, each with his or her own internal and visible states, and even a perceived neighborhood collective state. Doing this allows for a more complex local rule function, rather than just checking the neighborhood for alive or dead neighbors. The local rule computations rely on the fact that NPC state components use a numerical representation. Visible states require an integer to represent an animation sequence, while internal, and collective states can fall in between the visible state values. For instance, a collective state may be somewhere between alert and working, or distracted and bored. Table 1 shows the various behavioral states and their values.

Table 1. The various behavioral states and corresponding values. States lie on a spectrum similar to a number line. An NPC can increase or decrease to another state by passing through the states in sequence.

Behavior	In-game Value
ALERT	5
WORKING	4
DISTRACTED	3
BORED	2
TIRED	1

In order to formalize the behavioral state updates, we let C, I, V, W, E be defined as the collective state (C), internal state (I), visual state (V), neighborhood influence (W), and environmental influence (E), respectively. Separating the state update into three parts allows for a total of three main NPC state-update functions:

1. **Collective state.** An NPC should be able to visually perceive the neighborhood around them, so we maintain a collective state that depends on the external representations of neighbors. At time t+1, the next collective state C_{t+1} is computed for the current NPC (i, j) by iterating through its neighborhood map W. A weighted sum of the products of each neighbor's previous visible states V_t and corresponding neighborhood influence $W_{(k,l)}$ results in a collective neighborhood state. The function for the next collective state is defined by the nested summation:

$$C_{t+1}(i,j) = \sum_{k=i-1}^{i+1} \sum_{l=j-1}^{j+1} \left(V_t(k,l) * W_{(k,l)}(i,j) \right) \quad (1)$$

Each NPC (k, l) is in the neighborhood of NPC (i, j), including NPC (i, j).

2. **Internal state.** An NPC has an inner self-representation, which can be different from the outer visible state seen by other NPCs and the player. At time t+1, the next individual inner state I_{t+1} is computed for NPC (i,j), resulting in a weighted average of its own previous internal state I_t, local neighborhood collective state C_t, and environmental influence E. The function for the next internal state is defined by taking a scalar product:

$$I_{t+1}(i,j) = \langle \alpha, \beta, \delta \rangle \bullet \langle I_t(i,j), C_t(i,j), E(i,j) \rangle \tag{2}$$

Constants $\propto$, β, and δ add up to one, representing tunable weighting factors for the internal, collective, and environmental components, respectively.

3. **Visible state.** The next visible state of an NPC can be defined as a function of its previous internal, neighborhood collective, and visible states. Perturbations to any of these components may have an effect on an NPC's new visual state. If a visible state ages and is held too long, the visible state is automatically updated according to max-age rules. If the environmental influence becomes strong enough, the visible state will change to match the environmental influence.

The next visible state V_{t+1} is calculated as a numerical value corresponding to one single animation sequence. The visible state of an NPC is located within the behavior spectrum, so state changes occur either in a positive or negative direction along this spectrum. One way to determine the state change direction is to consider the closeness in value between the collective, internal, and environmental states and extrapolate an average state S_{avg} to use as a benchmark for comparisons. This average state can then be compared to the previous visible state V_t. The first step in finding S_{avg} is to use a distance metric to compare how close the NPC's previous internal state I_t is to the previous collective state C_t and environmental state E.

We let d_1, and d_2 represent the distance of an NPC's previous internal state I_t from

the previous collective state C_t and environmental state E, respectively in the following manner:

$$d_1 = \text{abs}\left(C_t(i,j) - I_t(i,j)\right)$$

$$d_2 = \text{abs}\left(E(i,j) - I_t(i,j)\right) \tag{3}$$

The average state S_{avg} is selected by determining the smaller value. If d_1 is less than d_2, then S_{avg} is set to one half of the sum of the previous visible state V_t and previous internal state I_t. Likewise, if d_1 is greater than d_2, then S_{avg} is set to one half of the sum of the previous visible state V_t and environmental influence E. If d_1 equals d_2, then S_{avg} is set to one third of the sum of all three values, V_t, I_t, and E. This is illustrated with the following conditions:

$$\begin{aligned}
&\text{if } d_1 < d_2, \text{ then } S_{avg} = \frac{1}{2}\left(V_t(i,j) + I_t(i,j)\right) \\
&\text{if } d_2 < d_1, \text{ then } S_{avg} = \frac{1}{2}\left(V_t(i,j) + E(i,j)\right) \\
&\text{if } d_1 = d_2, \text{ then } S_{avg} = \frac{1}{3}\left(V_t(i,j) + I_t(i,j) + E(i,j)\right)
\end{aligned} \tag{4}$$

The effect that (4) provides is in giving certain components more weight in determining what the next visible state should be. If the distance between C_t and I_t is smallest, then the collective state C_t is given more influence; however, as shown previously in (2), since C_{t-1}, I_{t-1}, and E are already embedded within the calculations for I_t, we use I_t in computing the average state S_{avg}. Similarly, if the distance between E and I_t is smallest, then the environmental state E is given more influence in the calculation. This determination of an average state is but one of many possibilities. The point is not to create an optimized simulation of human behavior, but to create a local rule that embeds personal and environmental information relevant to the NPC.

The average state S_{avg} is then compared to the previous visible state V_t plus or

minus a threshold value ε in order to determine if the next visible state V_{t+1} will increase or decrease along the behavior spectrum. If the average state S_{avg} is less than the previous visible state V_t minus the threshold value ε, then the visible state for the NPC is decreased by one unless already at the tired state. Likewise, if the average visible state S_{avg} is greater than the previous visible state V_t plus the threshold value ε, then the visible state for the NPC is increased by one, with an upper limit of the alert state. Otherwise, if the average state is within range of V_t plus or minus ε, no change to the current visible state if performed. This increase or decrease is illustrated with the following conditions:

$$\text{if } S_{avg} \le V_t(i,j) - \varepsilon, \text{ then } V_{t+1}(i,j) = V_t(i,j) - 1$$

$$\text{if } S_{avg} \ge V_t(i,j) + \varepsilon, \text{ then } V_{t+1}(i,j) = V_t(i,j) + 1$$

$$\text{if } S_{avg} < V_t(i,j) + \varepsilon, \text{ and } S_{avg} > V_t(i,j) - \varepsilon, \text{ then } V_{t+1}(i,j) = V_t(i,j) \quad (5)$$

After (5) has determined the new visible state, the NPC is checked to see if it has already held this state long enough to constitute a forced state change. The effect is that NPCs will cycle between behaviors to simulate changing behaviors of real students in the classroom. For example, students that are paying attention and taking notes may watch their teacher for a while, write notes down on paper, and then repeat this process. This would occur for a particular NPC if they were in the alert state for a maximum amount of time, and then forced to update to the working state, or vice-versa. Table 2 shows example state change rules for each of the main behavioral states. Once a new visible state has been reached for the entire population, each NPC is updated and ready for the next state update. While this assumes the entire population is updating all at once, within the game environment, the firing of animations are set to perform at staggered intervals to prevent the robotic appearance of synchronized NPCs.

Table 2. State changes for aging states. A current state is only held for a maximum amount of generations before a new state is forcefully assigned.

Current State	Max Generations	Next State
Alert	6	Working
Working	3	Alert
Distracted	3	Working
Bored	6	Tired
Tired	9	Bored

III.2.2. Neighborhood Influence

Neighborhood influence maps used in this model's local rule calculations control the amount of influence any neighboring student has upon the collective state for the NPC being updated. Imposing a field of view upon an NPC allows neighbors within a radius of one desk to have more of an influence than neighbors that are farther away. The one-desk radius neighborhood for an NPC is defined as a 3x3 grid where each member is a neighbor to the center NPC. In typical CA, each cell has either a 360-degree field of view, or no view at all. In emergent Room 309, each NPC has a forward facing field of view, which approximates what peripheral vision can account for.

The neighborhood influence map contains values for each member, including the center, that add up to one when summed. Figure 4 shows a neighborhood with a field of view, and interpolated values for the corresponding influence map. Each neighbor's value in the influence map corresponds to how much influence they have in the neighborhood. For example, students sitting in front of the class will be more influenced by the students sitting to the left, right or in their peripheral vision than students sitting directly behind them, so values in the influence map will reflect this.

Not all desk locations have the same number of NPCs within a neighborhood. Figure 5 shows specific regions in the classroom such as front, back, sides, corners, and middle that each contains a unique neighborhood type. Corner and edge seats have

locations within the neighborhood that are empty, while seats within the interior of the classroom have a full neighborhood of 8 influential members.

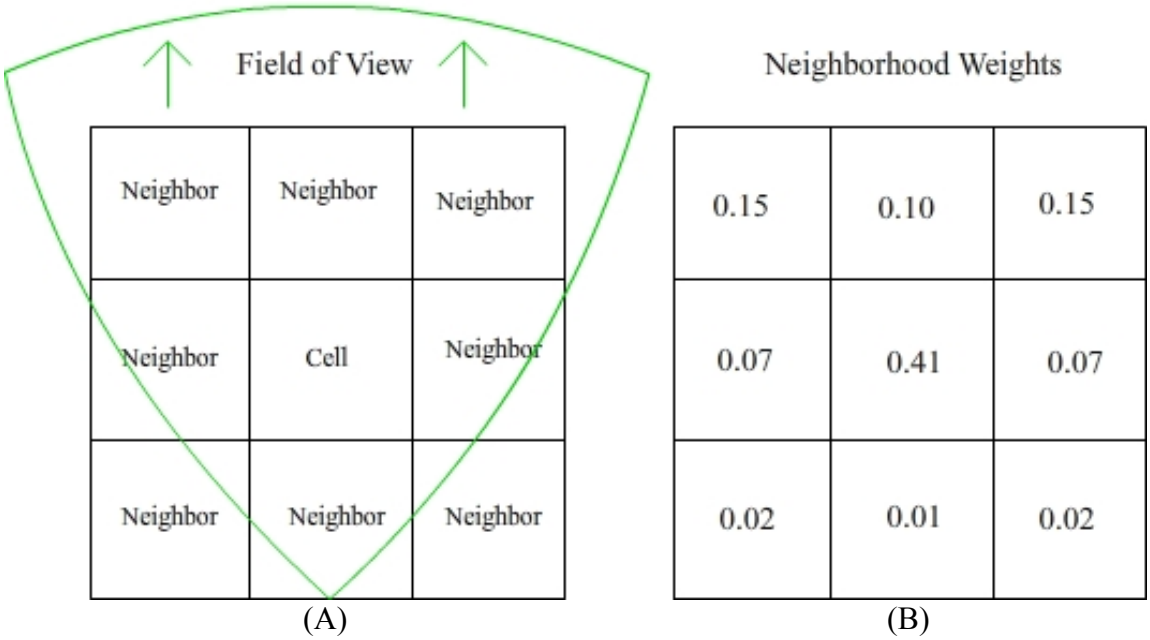


Fig. 4. NPC neighborhood and influence values. Arrows indicate direction of sight. (A) Superimposed field of view. (B) Possible neighborhood values.

Fig 5. Neighborhood types. Each color represents a unique type of view field for a particular NPC. Blue indicates three influential neighbors. Orange indicates five influential neighbors. Pink indicates eight influential neighbors.

III.2.3. Environmental Influence

The emergent Room 309 game environment contains important spatial influences that can further affect the behavior of an NPC student such as windows, teacher's desk, and the back row. The proximity of a student to these areas may determine their potential for engaging in behaviors associated with these regions. Figure 6 shows how these regions appear in the game. A student sitting near a window may be distracted from what is going on around them and be constantly trying to see what is outside. If a student is sitting near the teacher's desk they may have the potential for being on better behavior than other students, remaining in an alert state. Students sitting along the back wall are farthest from the teacher's desk, thus "proper" classroom activities may be reduced in this area, with increased instances of bored, tired, or distracted states. The environmental influence can direct an NPC student towards a specific state that can then influence the rest of the model.

Neighborhood influence maps allow unique weighed averages of visual states for NPCs, which depend on their location in the classroom. Similarly, we may apply neighborhood influence map techniques to find an environmental influence value for each desk location in the classroom. The goal is to find a weighted average of an environmental influence upon the potential behaviors of a particular student. This is performed by first separating the different environmental influences. If we let each main classroom influence be represented by a separate color, we can define an environmental influence map (see Figure 7). We let green represent the window region, blue represent the teacher's desk, and red represent the back row. Windows are associated with distracted states, the teacher's desk is associated with alert states, and the back row is associated with bored states.

(A) (B)

(C)

Fig. 6. Screen shots of three main influential regions in Room 309. (A). The leftmost wall lined with windows. (B). The teacher's desk. (C). The back row.

Fig. 7. An environmental influence map for Room 309. Green is for the windows, blue is for the teacher's desk, and red is for the back row.

A convenience of using an image based environmental influence map is that several different versions can be painted and easily read into the model. Having more than one map available to represent different environmental influences is useful for creating variation between successive plays, so that even the environmental influence can change based on context. In Room 309, the contexts may range from the player giving a lecture, to administering a quiz. If students are supposed to be quiet during a lecture, the window zone can be expanded, or if the back row has some troublemakers, the back row influence may be extended forward. Depending on the player's actions in the classroom, the environment can also be dynamically updated.

The size of the environmental influence map is only 100x100 pixels. This is enough resolution for a 5x5 grid of students because the map is divided into 25 blocks, one for each desk location. Each block is 20x20 pixels, where the pixels within each block are averaged to find a representative red-green-blue (RGB) pixel vector (P). The vector P is then normalized in order to scale the RGB values within a range of [0,1], which is illustrated by the following:

$$P(i,j) = \langle R, G, B \rangle$$

$$\hat{P}(i,j) = \frac{1}{R+G+B}\langle R, G, B \rangle \tag{6}$$

With a normalized pixel vector, the environmental influence E at the location of NPC (i,j) is determined by taking the scalar product between the RGB components of normalized P, and a vector of the RGB associated behaviors. Equation (7) illustrates this as follows:

$$E(i,j) = \hat{P}(i,j) \bullet \langle \text{Bored}, \text{Distracted}, \text{Alert} \rangle \tag{7}$$

Values for the RGB associated behaviors are integers found along the behavior spectrum. Figure 8 shows the calculated influence values for the environmental influence map shown above.

3.00	4.53	5.00	5.00	5.00
3.00	3.98	5.00	5.00	5.00
3.00	3.77	4.90	5.00	5.00
2.80	2.94	4.28	4.39	4.38
2.46	2.16	2.20	2.25	2.08

(A)

(B)

Fig. 8. Calculated influence values (A), for influence map (B). Colors from the influence map are interpreted as behavioral states.

The environmental influences act as constants in the model, and can be inserted into computations for determining the behaviors of a student. It's also possible to have a dynamic environmental influence value, hence updating the influence map based on how the player interacts with the NPCs. This possibility is described in later sections.

III.3. Pre-visualization

A tool for a 2D visualization, or pre-visualization, was developed in order to isolate the CA model to test and fine-tune parameters, while saving development time. The tool gives the developer controls to evolve or pause the model, clear the grid, and step forward one generation at a time. A randomization control allows the developer to completely randomize the model at any time during the simulation. The pre-visualization is useful because influence maps can be tested and their effect on the general behavior model evaluated before implementation into the game engine.

The code that drives the CA is the Automata data class, which is mostly portable to the Room 309 source code already embedded in the game engine. However it requires restructuring to fit in with the Source specifications. Appendix A.1 discusses the basics of the Automata class representation with a brief description of its data types and functions. Figure 9 shows a screen shot of the pre-visualization program used throughout the testing and modification of emergent Room 309.

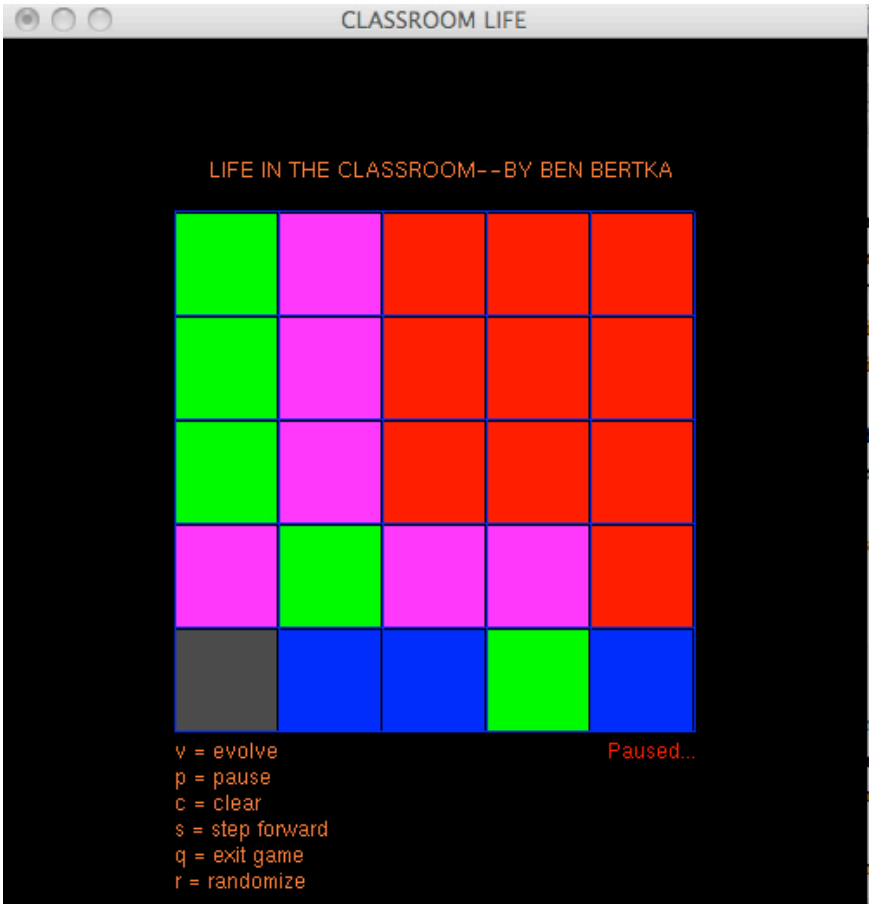


Fig. 9. Pre-visualization tool used to test the model during development. Depicted is output while testing the environmental influence map shown above.

III.4. Automata Entity

The original Room 309 was built using the default toolset within Valve's Hammer World Editor, which required that each scenario or interaction between NPCs to be choreographed in advance. According to the Valve specifications, all animations used by NPCs are required to have an entity representation within the map they are to be used in. Each NPC must have its own set of scripted animation sequence entities available upon map spawn, or their animation cannot be played. With the many animation possibilities per NPC in emergent Room 309, the amount of time and effort to create every possible scenario of interaction within a classroom of 25 students is time consuming and nearly impossible to do. The cellular automata model is designed to give the level designer the ability to just load the map with the minimum set of entities required, and not think of how the scenarios play out in advance. Figure 10 shows a basic flow-map of how an automata entity is integrated into a framework for controlling the NPC animations. The automata entity acts as an interface to the cellular automata model such that other map entities can link to it and access information.

When a map spawns, a timer is initialized that sends incremental signals to the automata entity requesting to update the NPC population. The state update calculations result in an integer being forwarded to a logical case manager for each NPC. The logic case acts like a switch statement, accepting an integer that corresponds to a particular animation sequence, forwarding it to a scripted animation sequence node. The scripted sequence is linked to the appropriate NPC, and when notified by the logic case, instructs the NPC to play the respective animation. Appendix A.2 discusses how a map was set up for emergent Room 309 using a combination of both custom and pre-packaged entities.

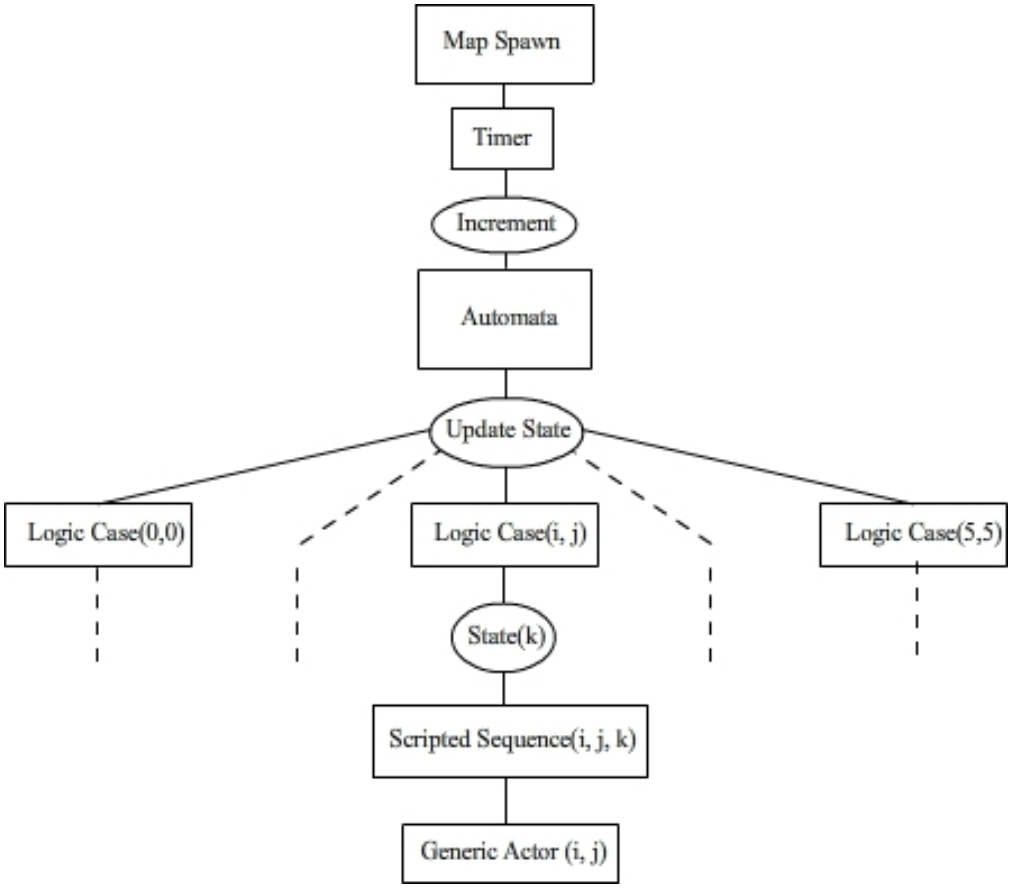


Fig. 10. Flow map describing information transfer with the automata entity.

The Automata data class, which represents the cellular automata model is made available to the game, and level designer in the form of a Hammer entity called automata_logic. Since the automata_logic entity has inputs and outputs that manage AI, and it is used with other logical entities, its class can be extended from the Source engine's CLogicalEntity class, and defined as CAutomataLogic. The main functionality of the automata_logic entity is to initialize the Automata class, as well as update and query NPCs. Appendix A.3 discusses the CAutomataLogic class, and how the automata_logic entity interfaces with the Automata class. Figure 11 shows how the automata_logic enity appears to the designer in the Hammer editor.

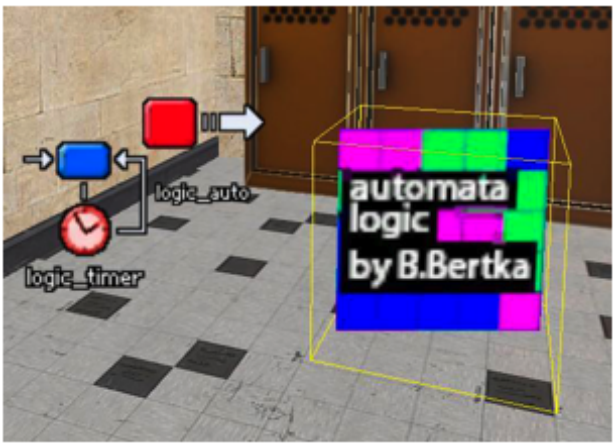


Fig. 11. automata_logic entity placed in the editing environment.

III.5. Player Influence

The environmental influence map helps direct the behavioral states of each NPC by giving them a potential to remain in their state unless provoked by an outside influence. The player acts as an influence upon the NPCs by becoming an influential factor in the environmental state. The environmental influence map is dynamically updated to account for the location of the player in relation to NPCs. The simultaneous effect of the player's proximity to the NPC is to heighten the alertness of the NPC, and force the environmental influence to adopt an alert state influence at the corresponding map location. The result is a dynamic environmental influence upon the NPCs, with the player's movements controlling the influence.

In order to allow the player to dynamically interact with the model, a means of notifying the automata of any changes the player makes is created. This notification is in the form of a trigger event initialized when the player is within range of an NPC. Pre-packaged Hammer entities exist that allow for the checking of special conditions relating to the player and NPC, such as ai_script_conditions; however, due to poor performance, they were not used in emergent Room 309. Instead, trigger boxes are placed around an

NPC to detect whether the player has started, finished, or is continuously touching the trigger. Figure 12 shows a trigger placed over an NPC.

An engaged trigger notifies the automata entity that the player is near and updates the environmental influence map, immediately setting the enclosed NPC to a new state. For example, if an NPC were distracted or tired, it would be put into an alert state because the player is within the trigger box. During the next state update, the other NPCs can be affected by this subtle change of states. Various game events may occur because of the player's influence such as setting NPC states, or as will be demonstrated in later sections, thwarting a cheating scenario. The environmental influence will revert back to the original image-based influence map value if the player leaves the trigger box.

The exact effects of player influence on the model are largely unpredictable, but what's certain are that the NPCs with triggers that overlap where the player is located will be internally, and visually updated, hence the automata model is dynamically influenced by the locality of the player. Details regarding the use of triggers with the automata entity are further discussed in Appendix B.4.

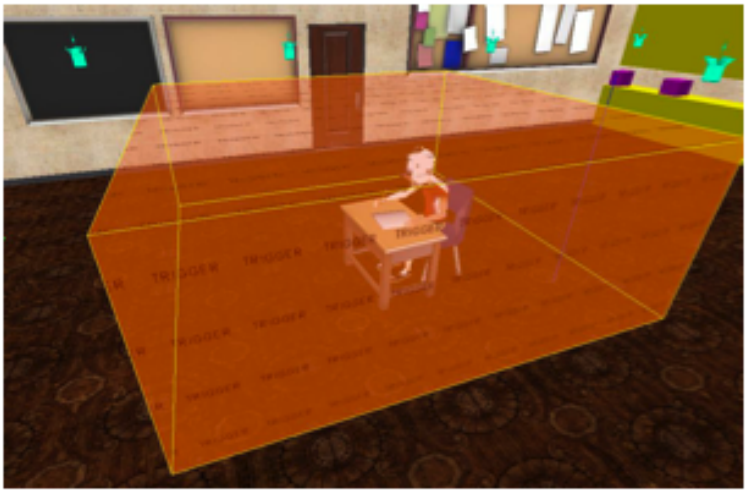


Fig. 12. A trigger box placed around the NPC can detect the player's presence.

III.6. Randomizing NPCs

The original visual development for Room 309 was centered about a caricatured stylization of the game environment that fosters attention to the behaviors of NPCs more so than attention to graphics. Due to the small texture pool initially present in scripted Room 309, a simple randomization scheme that merely chooses random textures and models for any given NPC would risk the occurrences of twins, or duplicate NPCs. While twins are common in nature, an average classroom might not contain more than one set of twins, and it would be even less common to have the twins sit next to each other. The visual style of Room 309 is maintained, and twins prevented in a more visually complex classroom by creating a larger texture pool for the NPC, along with randomizing genders. Each 3D model can be textured with a large enough set of textures such that randomizing the seating arrangement is efficient, and the stylized simplicity of Room 309's aesthetics is maintained. Figure 13 shows an increased texture pool for a male NPC in Room 309.

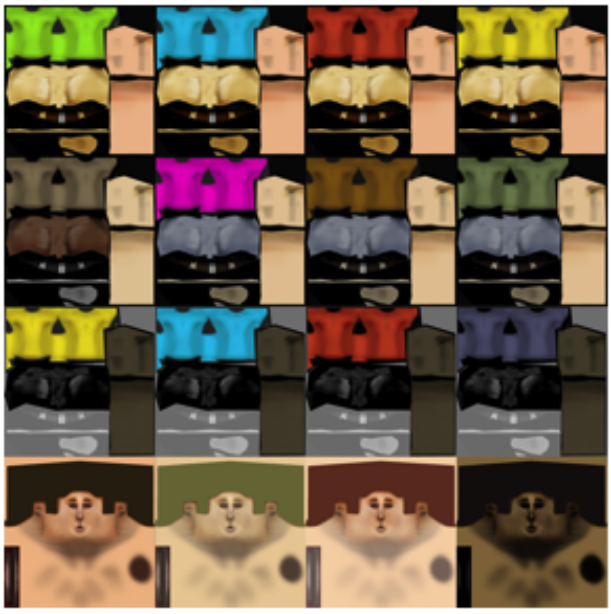


Fig. 13. Increased texture pool for male NPC.

Randomizing the NPCs allows the game designer to place a generic NPC into the map without preference or thought to the gender or texture distributions. This makes it easy to create variations of the same map without rearranging the NPCs because different configurations of NPCs spawn when a map loads. To create such an NPC, the pre-packaged generic_actor entity is modified so that upon NPC spawn, a random gender and texture are applied. The randomized gender and texture distribution occurs due to drawing without replacement from a pool of genders and textures.

To create a randomized distribution, the pool of textures is created to be at least half the size of the entire class for both male and female NPCs. When the Automata class instantiates, a set of gender-texture pairs is created by choosing random gender-texture pairs, while checking to see if the pair is in the list of already chosen pairs. If a pair is found to already be in the set, a new pair is selected and the set checked again. The minimum amount of available textures per gender should be at least half the total number of NPCs in the map. This allows for an evenly distributed gender assignment for the population, while preventing twins.

Random selection from a larger pool of textures yields a more colorful classroom where each student is uniquely paired with a head and body texture. The larger texture pool enables the use of a duplicate exclusion scheme, thereby eliminating the frequent sets of twins found in early versions of Room 309. The emergent Room 309 is balanced with more body textures than head textures per gender, so the classroom keeps to the original visual intent of a stylized, yet simple aesthetic. Figure 14 shows a comparison between a randomized classroom using the new texture pool and an un-randomized classroom using the smaller texture pool. Random spawning of genders using the original limited set of Room 309 textures yields a satisfactory distribution of male and female NPCs, however the small set of textures creates too many sets of twins in the classroom.

To create the random NPCs, the CRandomGenericActor class is created as a duplicated extension of the CGenericActor class provided by Source, and modified in order to directly interface with the Automata class for randomization. Appendix B.1

discusses how random NPCs were modified from the original Source generic_actor entity, and Appendix B.2 discusses how the random NPCs were textured.

(A)

(B)

Fig. 14. Texture pool comparison. (A). Smaller texture pool with several sets of duplicates. (B). Randomization with the larger texture pool, with duplicates forbidden.

CHAPTER IV

VIGNETTE IMPLEMENTATION

IV.1. Cheating Vignette

The first scenario implemented in the original Room 309 was a vignette where two students in the class engage in cheating with one another. While the class is taking a quiz, a male student gets a female student's attention, and is able to copy her work. The goal of the cheating vignette was to have the player recognize that cheating was occurring in the classroom, and then have the player handle the situation properly using a pop-up multiple-choice panel. The choice the player makes gets logged, and the next vignette begins. Game play was completely scripted, with the cheating scenario occurring between the same two students every time the map loads.

The level of predictability in the original vignette is very high, and upon successive plays, a player need not recognize cheating behavior; rather, they just need to remember where to navigate, and how to handle the situation. Using this vignette for implementing emergence is a challenge due to the scripted nature of the Source engine, how the original vignette was implemented, and the effects emergence has on game play.

Creating an emergent Room 309 has forced a new setup for mapping vignettes. Now, each NPC is placed into the scene without worry of twins, or level designer bias. For each NPC there are multiple entities containing the animation scripts, and a single entity that controls the firing of new animations. Triggers fill the room and are around each NPC listening and waiting for the player. These new additions to the game represent a very large increase in the amount of entities, and I/O connections between entities within a particular map.

The first goal of the original cheating vignette was to create cheating behaviors between NPCs that players must recognize. The NPCs were always in a constant state, and cheating happened for no other reason than it being scripted to occur. The cellular

automata behavior model takes the control over the NPC population, so specific animations and interactions no longer need to be scripted in advance, they can happen whenever the conditions of the CA allow it. Since cheating behaviors are a subset of the distracted state, special cases exist within the automata model that dictate when and how to initialize cheating.

The need for unpredictability of the location and frequency of cheating creates a problem for implementing player engagement with NPCs. In the original Room 309, a trigger was placed around the cheating students that detected the player's proximity to the NPC. If the player wanted to pull up the panel and handle cheating, they must be in range before hitting a keyboard button to do so. With an unpredictable pair of cheating students comes the need to change the way a player is able to engage a student. New triggers for each possible pair of students can be added to the map, however this further clutters the scene with more entities to manage. Due to the player influence described earlier, there are trigger entities to detect the player's proximity to each student, so it's possible to utilize them and check that when a panel is requested, the player is inside a trigger box of a cheating student.

IV.2. Cheating Mechanics

To implement a randomized cheating scenario in emergent Room 309, specific requirements are met in order to ensure the path to successful completion of the vignette is clear for the player. The first requirement is that only two random students are engaging in cheating at any one time. Many instances of simultaneous cheating are redundant because each situation is handled in the same manner, with the player just running through the aisles in the classroom trying to engage cheating students. Also, if there are other objectives and behaviors to recognize, too many instances of cheating could clutter the classroom and make other situations impossible. To prevent simultaneous occurrences of cheating, the Automata class checks each state update for instances of cheating and only allows the possibility of selecting random students for

cheating when none already exists. The second requirement is that cheating must occur with a student engaging either their left or right neighbor, which is an upgrade from the original scripted Room 309. Every NPC has enough programmed animations that any left-to-right pair of NPCs is capable of acting out a cheating scenario. The chances of a pair of students being selected for cheating depends on how many students are in a distracted state. Selection occurs when a random index is chosen and matched to an NPC. If the chosen NPC is distracted, it will begin cheating with either the left or right neighbor. Selecting the neighbor for cheating is performed via random number selection of either a zero or one; zero queues the left student, and a one queues the right.

The set of animations that make a choreographed cheating scenario are tapping, showing, looking, and working. Table 3 shows the possible cheating sequences between both the NPC who is selected to initiate cheating, and their neighbor. The choreography for the animations is scripted in the sense that the animations themselves must occur in the same order every time. For example, if the selected NPC is chosen to cheat toward the right, then they will begin Phase-1 cheating by tapping to the right, getting the attention of their right neighbor. Phase-2 begins after the next generation, with the neighbor set to play a showing animation, while the tapping NPC looks on. Phase-3 occurs on the next generation, with each NPC switching roles from the previous phase. If undisturbed, the cheating scenario ultimately concludes at Phase-4 with both NPCs put into a working state. Figure 15 shows cheating, with NPCs acting out the cheat phases.

Table 3. Possible cheating sequences.

Cheat Phase:	Phase-1	Phase-2	Phase-3	Phase-4
Selected NPC	Tap Right	Look Right	Show Right	Working
Neighbor NPC		Show Left	Look Left	Working
Selected NPC	Tap Left	Look Left	Show Left	Working
Neighbor NPC		Show right	Look Right	Working

Phase 1 - Tapping

Phase 2 – Showing/Looking

Phase 3 – Showing/Looking

Phase 4 - Working

Fig. 15. Cheating phases 1-4.

IV.2.1. Cheating Thwarted!

Once cheating has started the player may intervene with the students and interrupt their behavior. Since the player's proximity to the trigger boxes located around each player has the immediate effect of changing the state of the environment and NPC, a check is performed that tests if the student in the trigger box is cheating. If a student is in the middle of cheating when the player is near, the total number of cheats engaged by the player is saved for score keeping, and the cheating behavior is thwarted. This type of handling of cheating behaviors is not very different from the original Room 309, however this new system allows for any distracted student to be a cheater, and to allow cheating to continue happening at unpredictable intervals.

In the original Room 309, once cheating was noticed, the player would pull up a multiple-choice panel that was used to handle the situation in various ways. A view of the panel is shown in Figure 16. The panel was originally restricted to display only if the player was near the students cheating, for the trigger box would notify the game client that the player was near and any requests to show the panel were allowed at that time. This technique is used within emergent Room 309 by adding functions to the source code that determine if conditions are met for the player to bring up the decision panel. To allow the game environment to be aware of cheating in any location, the animation sequences involved with cheating set a flag that says cheating is occurring at a particular location. Trigger boxes are then used to determine weather or not a player is near the NPC. When it is determined that a player is near an NPC playing a cheating animation, only then may the panel appear. Implementation overviews for cheating choreography are discussed in Appendix C.1 while details on the decision panel are discussed in Appendix C.2.

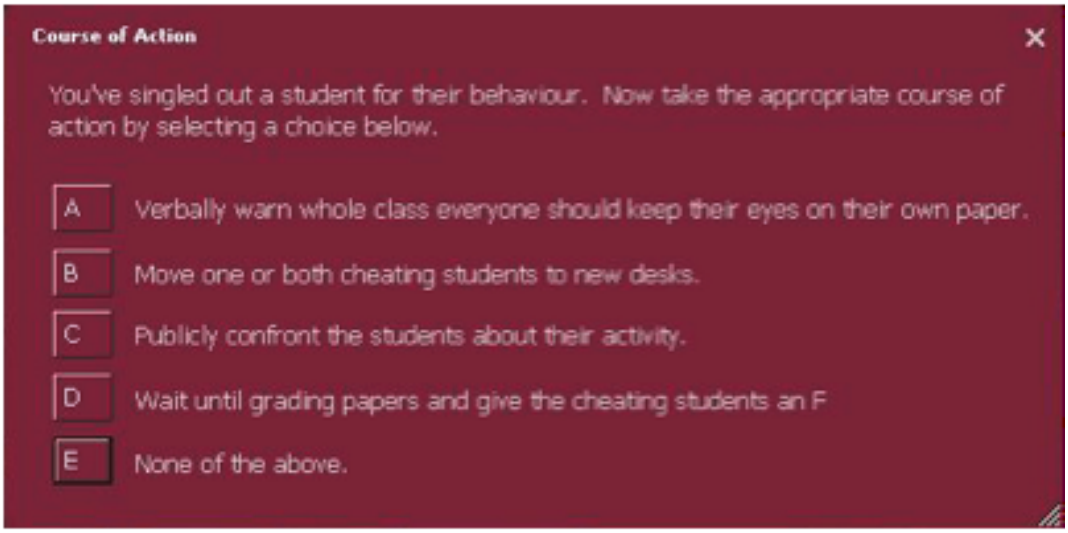


Fig. 16. Decision panel for the cheating vignette.

IV.3. Statistical Feedback for Players

Statistical feedback about the game environment is displayable on the screen and can be used by the players to track their own progress. Game data from the Automata class is made available such as playtime, generation number, average visual state, number of cheating attempts, and thwarted cheating attempts. Showing all the data at once may not be necessary or desired during a learning session, so the messages can be turned on and off independently.

The Source engine provides an entity for displaying static messages on screen, however in order to display dynamically created data, a new entity that can pull real time information from the Automata class is more desirable. Appendix C.3 discusses the implementation of an automata_stats entity that displays dynamical data on-screen. The final result of creating dynamic text is a verbose game environment with useful data for the developers and player (see Figure 17).

Fig. 17. Screenshot with various statistical data displayed as text.

CHAPTER V

ANALYSIS OF THE EMERGENCE MOD

Evaluation requires attention be paid to the performance of the CA model, effectiveness of influence maps, and overall randomization of NPCs. Fused together, these attributes have a major effect on the look and feel of the game play. The automata model and environmental influence maps dictate how the NPCs behave, so it is important to know what constitutes a good performing influence map. The results from successive map spawning show how well the randomization scheme works for NPC gender and texture distribution.

V.1. Testing the Model

Using the pre-visualization, we can test different values for constants in the state-update functions, try various state change rules, and input different environmental influence maps. Ideal performance of the model occurs when overall variation between cell states on the grid remains high for many generations.

The state update functions have four constants that can be changed. The internal state calculation in (2) has $\propto$, β, and δ for weighting the internal state components, while the visible state calculation in (5) has ε, the deviation threshold value. The natural setting for $\propto$, β, and δ is for each to have an equal value of 0.333 so that each component has an equal weight in the internal state calculation. While there are many different ways $\propto$, β, and δ can be tuned, making them equal is a good baseline when testing various values for the ε value, however the various parameters can be tuned for interesting results, such as keying in and out the environmental influence δ. In testing various values for ε, it was found that the maximum amount of visual diversity between states was held between a range of 0.3 and 0.5. Therefore, for testing purposes, ε was set to 0.4 as a benchmark.

Using these values within the pre-visualization tool, the model can be tested with or without environmental influence. Figure 18 shows a sequence with zero environmental influence. Without environmental influence, the population doesn't settle down towards a cohesive group behavior. Over time, the grid patterns are highly varied, and do not repeat. A waving effect is seen when a particular state overtakes most of the population, then multitudes of other states take over. This grid behavior can be used in the game to model a classroom of students that are becoming excited, or out of control.

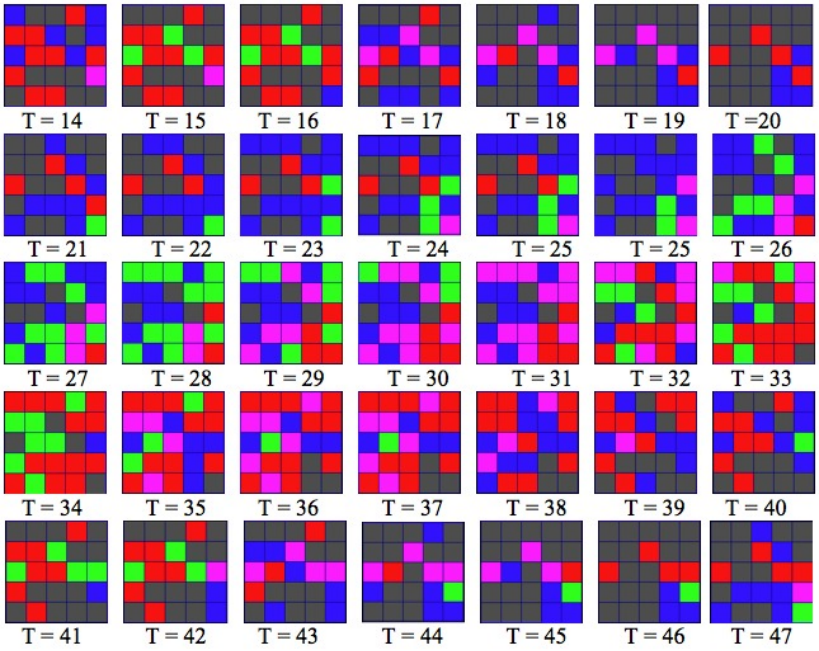


Fig. 18. Evolution without an environmental influence.

Using a full environmental influence has the effect where the population settles into a more organized state that matches up with expected behaviors for cells at particular grid locations. Figure 19 shows a sequence using full environmental influence. While this may seem like it creates linear and predictable game play, it is actually stabilizing the model. Stabilization is a good thing because it means the cells are

responding to the environment as well as each other. In this test, at T=69, distracted states (green) begin to settle into alignment with the left column of the grid which corresponds to window seats. At T=81, a heavier concentration of alert states (red) appear in the top right part of the grid corresponding to the teacher's desk area. From T=60 through T=81, working states (pink) are had in the top and center regions of the grid, and then eventually become alert states. Bored states (blue) occur at the bottom row of cells throughout the sequence. Tired states (grey) do not appear very often with only one being shown at T=60. Results show the CA model is highly responsive to the environmental influences.

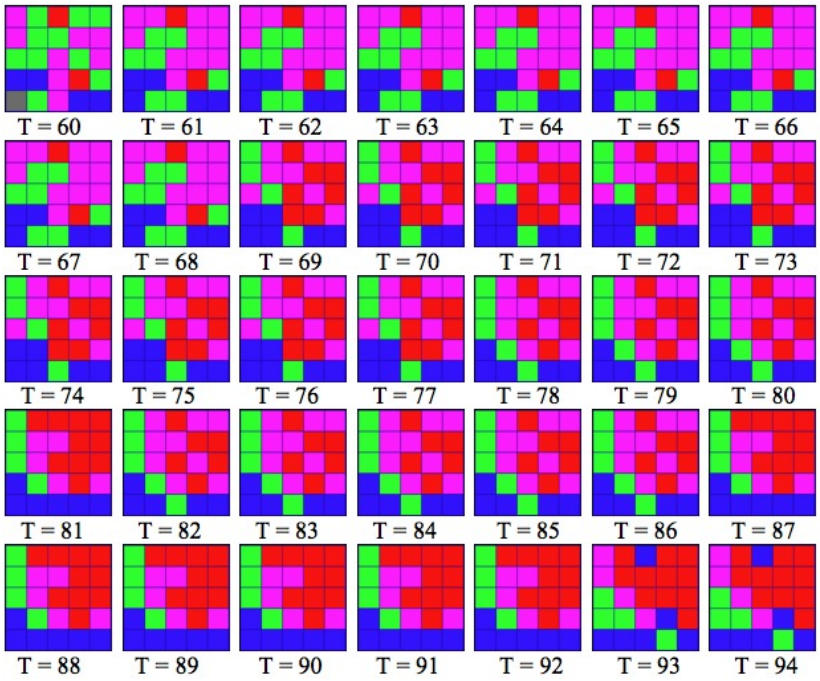


Fig. 19. Evolution with an environmental influence.

V.2. Randomization Evaluation

To check the effectiveness of the randomization scheme, a map was spawned 25 times and checked for duplicate NPCs, and distribution of gender and texture. The clustering of similar genders in neighborhoods did occur, although there was never a large enough region of the same gendered NPCs that the classroom looked unnaturally distributed. Overall the gender distribution did not have as great of an effect on the look and feel as did the texture distribution. While there was enough variation in per-gender texturing, the fact that similar colors are used for both male and females created clusters of similarly colored NPCs, (i.e. same colored shirts). Figure 20 shows clustering of students wearing blue shirts. However this could be remedied by using a wider range of colors for NPC body textures. Over all, the randomization scheme prevented duplicate NPCs, yet could be tuned for maximum color balance.

Fig. 20. Students with blue shirts clustering together.

V.3. Environmental Influence Comparison

Several environmental influence maps were painted and used to test the CA model under various conditions. Figure 21 shows maps A-E that were created for this purpose, while Figure 22 shows the pre-visualization outputs, which depicts the environmental influences for the input maps. The pre-visualization tool was utilized for the initial test of each influence map and allowed for observation of long-term effects from their use. Influence maps A and B are variations of the same map, where map B was created by using a blur filter on map A. Maps A and B yield comparable output, localizing the associated behaviors similarly over time. Map C allows the back row influence to be drawn up into the center of the grid yet does not create more bored (blue) regions in output C. Instead, the distracted (green) states are emphasized taking over the entire lower left side of the grid. Map D uses polka dots, which noticeably keep corner and center cells of output D in the tired (blue) state. The bull's eye of map E generates a ring of distracted (green) cells surrounding a bored (blue) center cell. The observations show that indeed the influence maps create unique conditions that allow the CA model to help stabilize the system towards the environment's influence, even after many generations of the automata model.

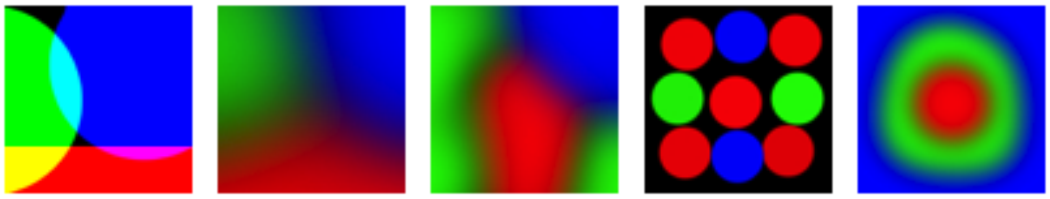


Fig. 21. Environment influence maps A-E used for testing the CA model. Note that here, blue is for the alert state, green for distracted, and red for bored. This color scheme differs slightly from the pre-visualization's output colors.

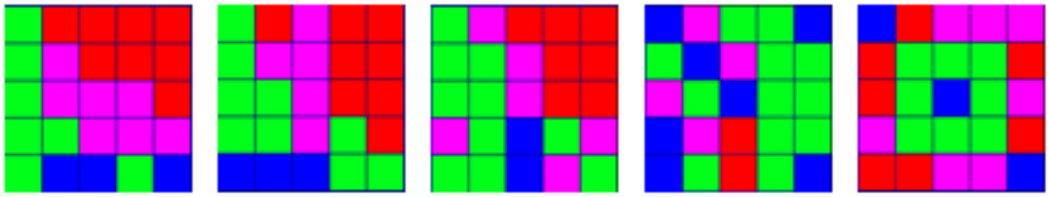


Fig. 22. Pre-visualization outputs A-E. These correspond to the environmental influences in Figure 21A-E respectively. Here, red is for the alert state, green for distracted, and blue for bored. This color scheme differs slightly from the environmental influence map color scheme.

To evaluate the impact that the influence maps have on the overall distribution and timing of animations, five successive spawnings of a single map loaded for each environmental influence was run for 100 seconds of play time each, and compared to one another. Of interest are the differences in plays found in intervals of relatively few tries of the same map, since a map in emergent Room 309 would be played only a few times in a row by the same player. In the tests, the player's influence on the environment was disabled and had no effect on the states of NPCs. Instances of cheating behavior were singled out for their frequency, as well as the number of repeat cheating behaviors between the same pair of NPCs. The average number of cheating attempts, repeated cheating attempts, and ratio of repeated cheating attempts per number of cheats is shown in Table 4.

Table 4. Influence map comparison from the first 100 seconds of game play. Maps A-E correspond to the influence maps in Figure 21.

Influence Map	Avg. Cheating Attempts	Avg. Repeated Cheating Attempts	Repeats/Cheating
A	2.2	1	0.45
B	3	1	0.33
C	1.8	0.2	0.11
D	2.6	0.2	0.07
E	3.4	0.6	0.17

The results in the environmental influence comparison trials show that for the short intervals of play, the largest amount of average cheating attempts occurred in map E, with the least occurring in map C. Maps A, B, and D held a mid range amount of average cheating attempts. Maps A and B were tied with the largest number of average repeated cheating attempts, with maps C and D tied for smallest average of repeated cheating attempts. Map E held the mid range of average cheating attempts. The averages for cheating attempts and repeated cheating attempts are used together to create a statistic for measuring the ratio for repeated cheating per total cheating attempts. Larger ratio values imply that cheating occurs more often between the same students, while a smaller ratio implies the cheating occurs less often between the same students. Map D has the smallest ratio, while map A has the largest. The surprising result is that Map D had the lowest repeats per cheating ratio, even with its polka dotted state distribution.

In all cases, cheating occurrences were located in the areas of classroom where distracted students are located, which is also the location of a distracted-alert state overlap region on the influence map itself. From observation, we can infer that influential maps with the largest areas of distracted (green) and alert (blue) areas that touch or overlap are more likely to have a larger number of cheating attempts in the first 100 seconds of game play than maps with less areas of similar states touching.

If cheating repeats are desirable in real game play, then any map presented here would do, however maps C, D, and E would be a good choice for minimizing repeated cheating between duplicate sets of NPCs. However not all game play scenarios need optimized cheating performance, so using maps that have high predictability in cheating locations would still be useful.

Using various environment influences such as the ones presented here is an efficient way to control game play and help make creating variation between successive plays unique. An entire game could be created using this approach by starting a map with a random environmental influence map, then after a certain amount of time, choose another random influence map. Cheating behaviors could be thwarted when and if they

occurred and the player would not be able to predict the next occurrence of cheating. A classroom experience that evolved overtime could make use of various sets of influence maps, with certain scenarios only playable if the environment map was loaded. The more influence maps that get painted, the greater variation in game play.

V.4. Modified Cheating Evaluation

The modified cheating vignette presents new game mechanics to the player in a dynamically reactive environment. The player is allowed to walk through the classroom thereby dynamically changing the environmental influences. When the player walks up and down the aisles in the classroom, nearby students who are working, distracted, bored, or tired immediately assume the alert state and look up to the player. If the player stands in the same location for a period of time, a local radius of alert NPCs forms, and spreads out to the other students, yet never overtakes the classroom.

Cheating interactions occur between NPCs with unpredictable locations and frequency. The player's navigation within the room disrupts the environmental influence, thereby creating new regions of distracted students other than what the environmental influence predicts. The unpredictability of cheating is most effective with the randomization and cycling of environmental influence maps.

Engagement of cheating NPCs by the player within the modified vignette is the same as in the original Room 309. The player is able to navigate towards the cheating pair of students and invoke the decision panel to handle the situation. The decision panel is only usable if the player is near a cheating student.

Multiple plays of the cheating vignette are always unique, and use of memorization alone won't allow a player to thwart cheating attempts. The vital game mechanics from the original Room 309 are preserved, and enhanced via local emergence. The final result is that the modified cheating vignette creates multiple chances for the player to practice their classroom management skills in an unpredictable and dynamic environment.

CHAPTER VI

CONCLUSIONS

V1.1. Summary

This work incorporates emergent game behaviors and environments using cellular automata, randomness, and influence maps within the existing non-emergent structure of an educational game. After introducing these qualities, game play became less predictable, potentially increasing effectiveness of Room 309 as a learning tool.

The modification process for emergent Room 309 included creating a cellular automata model for NPC student behaviors that accounted for collective, internal, and visible states of each member of the population. Image based environmental influence maps were used in local rule calculations of the CA model, and were dynamically updated based on player location. A randomization scheme was developed that prevented duplicate NPCs without bias for particular genders or textures and helped ensure unpredictability during successive plays. Custom entities for level building were created specifically for incorporating random NPCs, the CA model, and game statistics into the level-editing environment.

A playable vignette was created that allowed for random distribution of cheating interactions within the virtual classroom. Game play was borrowed and modified from the original Room 309 to incorporate random instances of cheating between NPCs and player interaction with NPCs. Results of evaluating multiple play tests with various environmental influence maps showed that certain maps allowed cheating to occur more often that others, and that an emergent game environment for Room 309 could be controlled with environmental influence map functionality.

The major contribution of this work was in the successful modification of the Source engine to include a cellular automata control system for NPC movement and interaction using influence maps, while using the system to improve game play within a

linear and scripted educational game. While this work is an overall success, improvement may be had with various aspects performed differently.

First, the local rule in the CA model was not optimized for efficiency, so algorithm analysis should be performed before developing a more complicated local rule. The multi-state model was created as a natural extension to the binary state model as found in Conway's Game of Life [Garnder 1970]. However, using integer values to represent the NPC states was limiting. Rather than creating an optimized version of this local rule within Source, it would be more beneficial to take the general idea of combining influence maps and cellular automata to control NPC states without the use of discreet numerical values, and create a fuzzy system that still considered the internal, collective, and environmental influences for determining a visible state. This could be performed in a different game engine, yet pre-visualized with a tool similar to the one used herein.

Secondly, since NPCs could only have one discreet animated state, accurate visible representation (or even fuzzy states) for the student population was limited or impossible, yet a local rule was formed around this limitation. These issues are in part due to the limitations of the Source engine that limits us to only assign one animation per NPC at any given time. This research could be carried out again with another game engine that allows NPCs to blend between animations in real time, which would be a good way to visualize a multi-state behavioral model.

Finally, a larger animation pool could have been used for all the behaviors, so that NPCs in similar behavioral states could have more than one way to express the same behavior.

Documenting the development of an emergent Room 309 adds to the body of research concerned with modding educational games built with first person shooter engines. This research showed that a scripted and linear game could be retrofitted with emergent qualities to improve game play and that using cellular automata was a feasible approach to control NPCs that require scripted animation choreography. While there is documentation of the use of the Source engine and other first person shooter engines for

educational games, none were performed with the intent of controlling NPCs with cellular automata and influence maps for the specific purpose of modeling the actual behavior of NPCs in a virtual classroom.

VI.2. Future Work

While this work shows it is possible to modify a scripted game in order to allow for some level of emergence into the game environment, a complete modification using the Source engine would be more work than necessary. This emergence modification allows for the reworking of other vignettes with mechanics similar to the cheating scenario, yet works against the natural workflow of the Source engine's scripted animation choreography requirements, and preference for lightweight maps with few NPCs. Too many animation sequence entities are required by the automata model to make regular level editing with Source a manageable task, especially when more vignettes are to be added. While it is possible to continue using Source, using a different game engine that doesn't require as much scripted choreography, and provides a more flexible world building environment, may allow for a different approach to the emergence modifications.

The procedures and methods for creating the single modified cheating vignette are a proof of concept that a scripted game engine could be used to create an emergent game environment, however designing for emergence in the first place needs to be a prerequisite, especially if the player is to become an integral part of the story. Moving forward with an emergent Room 309 would entail a redesign of the game mechanics, complete with emergent narrative. The visual development already present has its enhancements with the added touch of unpredictability, and this can be followed up with designing of an unpredictable day in the classroom, where the classroom evolves over time with many situations overlapping.

In order to juxtapose an emergent game environment with a typical day in the classroom so that learning outcomes are efficiently developed, and even possible to

begin with, a completely emergent Room 309 would require a focus group of educators that are knowledgeable in game design (including modding) and/or a group of game designers who have classroom teaching experience. With a planned out scheme for emergent environment and narrative, the most up to date game engines can be evaluated for their usefulness in creating the emergent classroom, with focus on implementing global emergence (within the classroom). A globally emergent Room 309 could be play tested and evaluated for its usefulness, engagement levels, and re-playability. In evaluating the benefits of such a designed gaming environment to the education community, creating a new heuristic for measuring the levels of useful emergence could be possible. If such a heuristic was created, studies by Yue and Zin [2009] that heuristically evaluate history games could take into consideration the emergent qualities of educational games.

REFERENCES

Autodesk, Inc. September 2010. Autodesk® Softimage® Mod Tool™. http://usa.autodesk.com.

Burkes, E. 1970. *Essays on Cellular Automata*. essay 15. University of Illinois Press, Champagne, IL.

Epic Games. September 2010. Unreal Development Kit. http://www.udk.com.

Funge, J. and Millington, I. 2009. *Artificial Intelligence for Games* 2nd Ed. Morgan Kaufman Publishers, San Francisco, CA.

Gardner, M. 1970. The fantastic combinations of John Conway's new solitare game "Life". *Scientific American 223*, 120-123.

Gregg, R. and Jedrzejewski, N. September 2010. VTFEdit. http://nemesis.twl.net.

Kickmeier-Rust, M. and Albert, D. 2009. Emergent design: serendipity in digital educational games. In *Proceedings of the 3rd International Conference on Virtual and Mixed Reality (VMR'09)*. 206-215.

Lecky-Thompson, G. 2008. *AI and Artificial Life in Video Games*. Charles River Media, Hingham, MA.

Mac Namee, B., Rooney, P., Lindstrom, P., Ritchie, A., Boylan, F., Burke, G. 2006. Serious Gordon: using serious games to teach food safety in the kitchen. In *Proceedings of the 9th International Conference on Computer Games: AI, Animation, Mobile, Educational and Serious Games (CGAMES'06).*

Marks, S., Windsor, J., and Wunsche, B. 2007. Evaluation of game engines for simulated surgical training. In *Proceedings of the 5th International Conference on Computer Graphics and Interactive Techniques in Australia and Southeast Asia (GRAPHITE'07).* 273-280.

Ritchie, A., Lindstrom, P., and Duggan, B. 2006. Using the Source engine for serious games. In *Proceedings of the 9th International Conference on Computer Games: AI, Animation, Mobile, Educational and Serious Games (CGAMES'06)*.

Sarkar, P. 2000. A brief history of cellular automata. *ACM Comput. Surv. 32*, 80-107.

Smith, D. 1987. *Decision Points*. Insight Media, New York, NY.

Smith, H. and Smith, R. 2004. Practical techniques for implementing emergent gameplay: would the real emergent gameplay please stand up? In *Proceedings of the 2004 Game Developers Conference (GDC'04)*.

Sweetser, P. 2006. An emergent approach to game design - Development and Play. Unpublished doctoral dissertation. School of Information Technology and Electrical Engineering. The University of Queensland, Australia.

Sweetser, P. 2008. *Emergence in Games*. Charles River Media, Hingham, MA.

Sweetser, P. and Wiles, J. 2005a. Combining influence maps and cellular automata for reactive game agents. In *6th International Conference on Intelligent Data Engineering and Automated Learning (IDEAL'05)*. 524–531.

Sweetser, P. and Wiles, J. 2005b. Scripting versus emergence: issues for games developers and players in game environment design. *International Journal of Intelligent Games and Simulations 4*, 1-9.

Sweetser, P., Johnson, D., Sweetser, J., and Wiles, J. 2003. Creating engaging artificial characters for games. In *Proceedings of the 2nd International Conference on Entertainment Computing (ICEC'03)*. 1-8.

Tozour, P. 2002. The Perils of AI Scripting. In *AI Game Programming Wisdom*. Charles River Media, Hingham, MA.

Valve Corporation. September 2010. Valve® Source™ SDK. http://www.valvesoftware.com.

Yue, W. and Zin, N. 2009. Usability evaluation for history educational games. In *Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human (ICIS'09)*. 1019-1025.

APPENDIX A

A.1. Automata Class

As implemented in the pre-visualization and within the Source™ engine modifications, the private data for an instance of the Automata class includes various two-dimensional arrays for storing state values of each member in the population. Figure A.1 shows the Automata class definition as used in the pre-visualization. Since there are a maximum of five behavioral states in the model, the range of state possibilities must exist within the interval [1, 5]. The visible states for NPCs exist in an integer array because the visible states of NPCs are whole values that refer to possible animation sequences. Since the internal and collective state calculations result in values that may fall between visual states, their data types exist as doubles. Since the states age over time, an integer array contains the amount of generations a NPC has been in a particular state.

The environmental influence map can be broken up into its main area components, of window, front desk, and back row influences:

```
double windowInfluence[5][5];
double frontdeskInfluence[5][5];
double backrowInfluence[5][5];
double environmentalState[5][5];
```

There are also 25 members to these arrays because the environment influences each NPC. These values are doubles because like the internal and collective state, the environmental influence may fall between whole valued states. The environmental influences may also be modified during game play when appropriate.

```
//=========================================================
// Class: automata()
// Purpose: Basic data class for the cellular automata model
//=========================================================
class automata{
public:
   automata();                              //constructor
   int ppmLoadMaps();                       //ppm loader
   void randomize();                        //grid randomization
   void computeEnvironmentalState();        //environmental state calculation
   void getNextGen();                       //state update function

  //collective, internal, and visible state calculations
   double getCollective(int (*tempVisible)[5], int i, int j);
   double getInternal(double (*tempInternal)[5], double collectiveState, int i, int j);
   int getVisible(int (*tempVisible)[5], double internal, double collective, int i, int j);

   int CheckAgeState(int visibleState, int i, int j);   //test age of state
   int getCellState(const int, const int);                 //age update

private:
   int visibleState[5][5];                  //visible state values
   double internalState[5][5];          //internal state values
   double collectiveState[5][5];            //collective state values
   double windowInfluence[5][5];            //window influence map
   double frontdeskInfluence[5][5];         //front desk influence map
   double backrowInfluence[5][5];           //back row influence map
   double environmentalState[5][5];  //environment influence map
   int ageArray[5][5];                  //age of the current state

   double influenceCenter[5][5];            //NPC Neighborhood influence maps
   double influenceTopLeft[5][5];
   double influenceTopRight[5][5];
   double influenceLeft[5][5];
   double influenceRight[5][5];
   double influenceBottomLeft[5][5];
   double influenceBottomRight[5][5];
   double influenceBottom[5][5];
   double influenceTop[5][5];
};
```

Fig. A.1. The automata class definition for the pre-visualization.

The final stored data members are the neighborhood influence maps. Nine regions exist in the classroom that each require their own neighborhood map type including the four corners, four edges, and interior. Since the distribution of NPCs is different in these areas, a separate influence map must exist for each of the nine regions. These regions are each accounted for with a three-by-three array of doubles:

```
double influenceCenter[3][3];
double influenceTopLeft[3][3];
double influenceTopRight[3][3];
double influenceLeft[3][3];
double influenceRight[3][3];
double influenceBottomLeft[3][3];
double influenceBottomRight[3][3];
double influenceBottom[3][3];
double influenceTop[3][3];
```

Each NPC uses one of these neighborhoods, and each NPC in the neighborhood provides a certain amount of influence during state updates.

The Automata() constructor creates an instance of the model, sets up initial conditions, and determines values for influence maps. It makes a call to the randomization function, reads in the neighborhood influence maps, and makes a call to the environmental influence map function which sets up the environment influence map.

Randomizing the initial states for each NPC occurs in the randomize() function. In this function, ages are set to zero, with random internal and visible states chosen for each NPC. The collective states for NPC neighborhoods are computed during the first grid update, so the collective state is left to a default value of zero.

Called by the constructor, the computeEnvironmentalState() function computes the environmental influence map using pixel data from a PPM image format. It computes the weighted average of associate behaviors from each environmental region for each NPC.

The main update loop is getNextGen(), and computes the states for the next generation of each NPC. It creates a temporary grid from previous states, and makes

calls to functions that calculate the next collective, internal, and visual states, respectively.

The collective state function getCollective() finds a value for the NPC at (i, j), using neighborhood influence maps. It determines which region of the grid to use, and returns the weighted average of the previous neighboring visible states to the main update loop.

After the collective state is determined, the getInternal() function computes the internal state for the NPC at (i, j) using the previous internal state. The current collective state is used in this calculation because it is based on the previous visible state. The return value is a weighted sum of the previous internal, collective, and environmental influence. Variable constants representing weight value for the visible, collective, and environmental influences are included in the calculation.

The getVisible() function is then called to compute a visible state for NPC (i, j) using the previous visible, previous internal, current collective, and environmental states. It generates an average visual state based on closest proximity of the internal state to either the collective or environmental state. It uses a variable constant to determine if the average state is within a threshold value of the visual state. If it is outside this threshold, a change in visual state will occur; otherwise the visual state will not change. Before a return value is sent to the main loop, a check is performed on the age and proposed visual state. The checkAgeState() function checks for the number of generations the visible state of NPC (i, j) has been in because certain visible states are not allowed to last longer than a certain amount of time before they are forced into another state. These forced changes can be changed to add different effects.

A.2. Automata Mapping

In order to create the map using the Automata class, the following pre-built and custom entities are utilized:

- logic_auto. This entity is initialized immediately and begins to fire outputs when the map is finished loading.
- logic_timer. This entity fires outputs during set intervals, or randomly.
- random_generic_actor. This is a modified *generic_actor* entity for the NPC, and has parameters for attributes such as a random model and texture.
- scripted_sequence. This entity holds one main action animation for a particular NPC. Each NPC has its own complete set of scripted sequences.
- logic_case. This entity is much like a switch statement in that a certain input is compared to a value, and depending on which one, a particular output is fired off.
- trigger_multiple. This entity is controlled by movement of the player or other entities. Depending on if the player, or other entity steps within its perimeter, unique events are possible.
- automata_logic. This is the main custom entity in the map which controls the game environment, and manages the cellular automata model. It is derived from the logical entity class; hence it's name automata_logic.

Placing an automata_logic entity in the map will automatically create an instance of the Automata class. However, the entity must be connected to other entities in order for it to have any effect on the NPCs. The automata entity must be triggered to update the states of the NPCs, as well as have a place to send the state information after each update. When the map spawns, the logic_auto entity will output a signal to the timer's input that tells the timer to begin sending incremental signals to the automata_logic

entity. Figure A.2.1 shows the output properties of the logic_auto entity and its connection to the timer as viewable in the level-editing environment.

The automata_logic entity will provide an interface to the Automata class that allows for updating and retrieving the states of all NPCs via UpdateState() and GetState() input functions, respectively. The timer can update the state of the automata as often or seldom as preferred. Figure A.2.2 shows the output properties of the logic_timer entity and its connection to the automata_logic entity as viewable in the level-editing environment. Since there are a lot of computations to make, and the animations for the NPCs take some time to run, we can set the timer to fire every five seconds. This is in contrast to the pre-visualization model where the states were updated much more frequently. Once the timer requests an update and the automata computes the next generation of the grid, the timer calls the GetState() function for each NPC sending a parameter that designates the index of the NPC queried. The GetState() function determines what the next visual state should be, and fires an appropriate output to a logic_case entity.

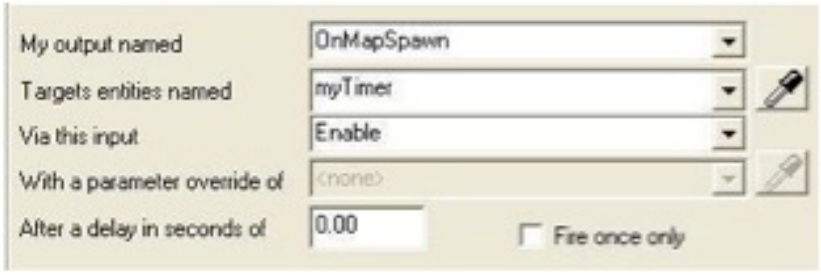
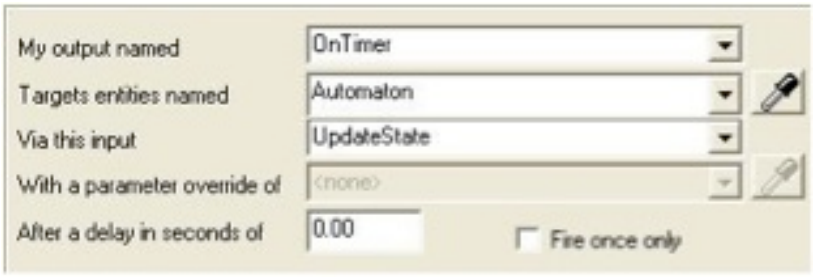
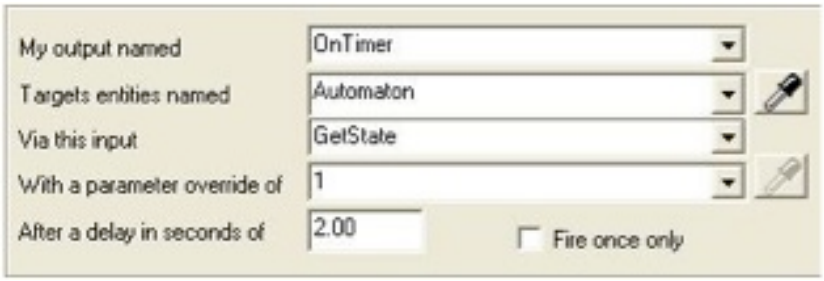


Fig. A.2.1. Output properties of the logic_auto entity.



(A)



(B)

Fig. A.2.2. Output properties of the logic_timer entity. (A). Linking the UpdateState() command (B). Linking the GetState() command for NPC no. 1.

The logic_case can be set to compare input values, and send the appropriate message to the corresponding location, in this case a scripted_sequence, hence it is placed in the map for each NPC, and then linked to each scripted_sequence. When activated, the logic_case fires an output to the appropriate scripted_sequence, instructing the NPCs to begin an animation. Each NPC in the map has the potential to be animated with either the tired, bored, distracted, working, and alert states. Since the distracted state includes separate animations for cheating and looking in different directions, while the alert state includes separate animations for being attentive and raising either hand, there are 14 possible animation sequences per NPC. With 25 students in a Room 309 classroom, and 14 possible animations each, there are 350 scripted_sequence entities required for a single map. Figure A.2.3 shows the level-editing environment for Room 309 loaded with both logic_case, and scripted_sequence entities.

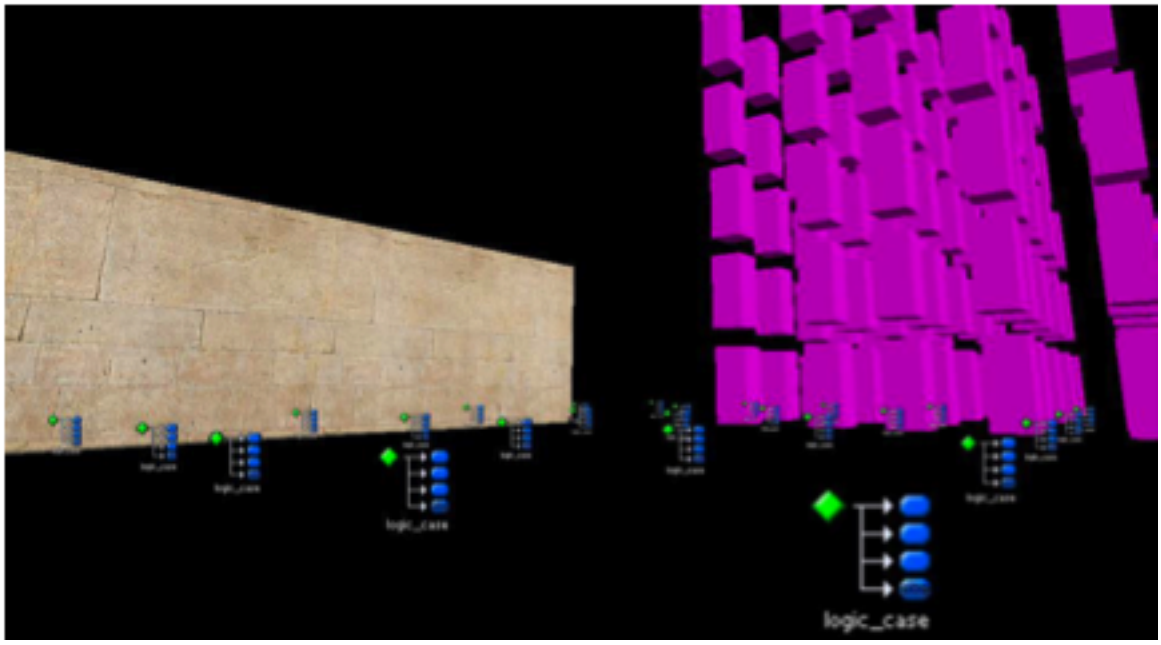


Fig. A.2.3. logic_case and scripted_sequence entities placed outside the room.

A.3. Automata Entity

An entity is defined by both source code and listings within a Forge Game Data (FGD) file. The FGD file contains definitions of all the available Hammer entities for mapping. A thorough explanation of FGD files can be found on Valve's developer page, http://developer.valvesoftware.com. As shown in Figure A.3.1, we can make the automata entity available by creating an FGD listing for inclusion with the rest of the entities.

```
@include "base.fgd"
@include "halflife2.fgd"
@PointClass base(Targetname) iconsprite("entity/automata_logic.vmt") =
automata_logic :
        "Provides an interface to the automata class. " +
        "Can be queried to provide appropoiate NPC state for specified index. " +
        "Must call the update function for each grid update. " +
        "Can randomize the grid when appropriate. "
[
// Inputs
input UpdateState(void) : "Evolve the CA Grid one generation."
input GetState(integer) : "Get the state of the queried NPC, and fire state output."
input RandomizeAutomata(void) : "Randomizes the grid when needed"

// Outputs
output OnState_0_TIRED(integer) : "Fired when the NPC is tired."
output OnState_0_BORED(integer) : "Fired when the NPC is bored."
output OnState_0_DISTRACTED(integer) : "Fired when the NPC is distracted."
output OnState_0_WORKING(integer) : "Fired when the NPC is working."
output OnState_0_ALERT(integer) : "Fired when the NPC is alert."

//output definitions for other animations, and NPCs are omitted for brevity
]
```

Figure A.3.1. The FGD entry for the automata_logic entity.

The class CAutomataLogic class is created as an extension of the public class CLogicalEntity, which other logical entities are derived from. Figure A.3.2 shows the data class listing for CAutomataLogic. The line containing LINK_ENTITY_TO_CLASS is just outside of the CAutomataLogic class definition and is the call that links the CAutomataLogic class to its specific FGD entry. In our case, the Hammer entity will be known as automata_logic, and will be linked to the CAutomataLogic class. We then begin the data description for this relationship and make linkages between each input and output. The calls that define the input functions declare the type of input data the function will accept, as well as a displayed name that will be made visible in the Hammer editor, and the name of the CAutomataLogic function to link with. Output definitions link the COutputEvent definitions from the CAutomataLogic class to the output definitions in the FGD file.

The CAutomataLogic class provides a function that tells the Automata class to update the entire classroom called UpdateCommand(). This function serves as a direct call to the getNextGen() function of the Automata class. Likewise, the GetStateCommand() input function is provided to query the state of an NPC, and fire an appropriate output. The input parameter is the index of the queried NPC. This function serves as a direct call to getState() function of the Automata class, but also fires outputs to other entities. When the Automata class returns the state of an NPC to GetStateCommand(), the appropriate output corresponding to a single animation for a specific NPC is fired to a logic_case. The logic_case entity for the particular NPC then decides which scripted_sequence should begin playing. Any function in the Automata class can be accessed via the automata_logic entity, such as the randomize() function. In order to provide a way of dynamically updating the environmental influence map based on player actions, an UpdateEnvironment() input function is provided.

```
//=========================================================
// Class: CAutomataLogic
// Purpose: Interface to the automata class
//=========================================================

class CAutomataLogic : public CLogicalEntity{
public:
DECLARE_CLASS( CAutomataLogic, CLogicalEntity );

COutputEvent m_State_0_TIRED;
COutputEvent m_State_0_BORED;
COutputEvent m_State_0_DISTRACTED;
COutputEvent m_State_0_WORKING;
COutputEvent m_State_0_ALERT;
        // COutputEvent variables for other NPCs are omitted for brevity

void UpdateCommand( inputdata_t& inputdata );
void GetStateCommand(inputdata_t& inputdata );
void RandomizeCommand( inputdata_t& inputdata );
void PrintState();

automata board;
int generationNum;
DECLARE_DATADESC();
};
LINK_ENTITY_TO_CLASS( bb_automata_logic, CAutomataLogic );

BEGIN_DATADESC( CAutomataLogic )

DEFINE_INPUTFUNC( FIELD_STRING, "UpdateState", UpdateCommand ),
DEFINE_INPUTFUNC(FIELD_INTEGER, "GetState", GetStateCommand),
DEFINE_INPUTFUNC(FIELD_VOID, "RandomizeAutomata", RandomizeCommand),
DEFINE_OUTPUT(m_State_0_TIRED, "OnState_0_TIRED"),
DEFINE_OUTPUT(m_State_0_BORED, "OnState_0_BORED"),
DEFINE_OUTPUT(m_State_0_DISTRACTED, "OnState_0_DISTRACTED"),
DEFINE_OUTPUT(m_State_0_WORKING, "OnState_0_WORKING"),
DEFINE_OUTPUT(m_State_0_ALERT, "OnState_0_ALERT"),

//DEFINE_OUTPUT functions for other NPCs are omitted for brevity

END_DATADESC()
```

Fig. A.3.2. The CAutomataLogic entity class, with I/O definitions.

APPENDIX B

B.1. Modified NPCs

The source code for generic_actor is modified such that upon spawn, an NPC is randomly assigned a model and texture. This modification in the source code of the NPC, as uses functionality present in the Automata class randomize() function. When the Automata class is instanced and the randomize() function called, the randomized texture and model set is determined, letting the NPC grab this information when it spawns into the map. The following do-while loop generates a pair of numbers referring to a gender, and texture respectively:

```
do{
        gen = rand()%2+1;  //value between 0 and 1, two genders
        tex = rand()%TEXTURES+1; // value in range of textures
}while(!checkGridforDupes(gen, tex));
```

With a randomized set of genders and textures, the generic_actor entity can be modified to pull this information from the Automata class and assign them to NPCs as they spawn into the map. To customize the NPC, the CGenericActor class provided by the Source engine is duplicated, and renamed to CRandomGenericActor. The corresponding FGD entry for generic actors is also duplicated so that a random_generic_actor entity is available in the level editor.

The modification to the source code for CRandomGenericActor consists of changing the way an NPC is spawned, and adding storage space for the random model's relative path from the mod directory, and texture lookup ID. Figure B.1 shows the CRandomGeneriActor class modifications. A method called PickRandomModel() is inserted into the top of the Spawn() function that assigns a gender-texture pair to the NPC. Since the NPC's are spawned in the order they were placed into the map, rather

than an ordered desk location, the location of the gender-texture pair within the automata class's set of pairs is arbitrary; however it is noted that a 5x5 data structure is made

```
//=========================================================
// Class: CRandomGenericActor
//=========================================================
class CRandomGenericActor2 : public CAI_BaseActor{
public:
	DECLARE_CLASS( CRandomGenericActor, CAI_BaseActor );
	Void Spawn( void );			//spawns the NPC
	void PickRandomModel( void );		//gets random model
	const char* randomModel;		//MDL path
	int randomTexture;			//texture ID
	  //unmodified code
};
void CRandomGenericActor::Spawn() {
	  PickRandomModel();
	  m_nSkin = randomTexture;
	   //unmodified code
	  NPCInit();
}
void  CRandomGenericActor::PickRandomModel(){
	  int count = 0; int found = 0;
	  for(int i = 0; i < 5, found != 1; ++i){
	       for(int j = 0; j < 5, found != 1; ++j){
	            if(studentData.numOccupied == count){
	                    randomTexture = studentData.textures[i][j];
	                    int genType = studentData.genders[i][j];
	                    If(genType == 1)
	                            randomModel = "models/girl/girl_automata.mdl";
	                    else
	                            randomModel = "models/boy/boy_automata.mdl";
	                    studentData.numOccupied++;
	                    found = 1;
	                    break;
	            }
	            ++count;
	       }
	  }
}
```

Fig. B.1. The modified code for CRandomGenericActor.

available for future development in keeping track of exactly where each NPC is located, in case of mobile NPCs. In this current implementation a count is maintained on the number of NPCs in the room, and the first available gender-texture pair not already used is assigned.

The randomly selected gender value is binary, representing male or female, and stored in the studentData.genders array of the Automata class. A value will correspond to either a male or female gender, and the appropriate path to a model will immediately be used during NPC initialization. The inherited variable m_nSkin holds the texture information for the model and is already supplied in the Source engine code for generic NPCs, hence we set it to the randomTexture value from the studentData.textures array of the Automata class.

B.2. Texturing the Random NPC

Texturing the NPCs in Room 309 is a normal procedure, which includes unwrapping the 3D model to be textured and obtaining its UV layout. The UV layout is used to map the 3D points from the geometry onto a 2D image. This layout can then be used as a guide when painting a texture for a model, and saving the image for use as a texture to be applied in game. Each texture must be compiled into a proprietary Valve Texture File (VTF) file format. While Valve supplies a command-line tool for this conversion, there are various third-party plug-ins that allow the conversion to VTF. This work utilizes VTFEdit [Gregg and Jedrzejewski 2010].

With a large enough set of textures at hand, it is necessary to mention the QC files describing both the male and female actors. Later sections discuss animation and why the QC files are important, however for texturing we are most interested in the $texturegroup variable because it specifies groups of textures that can be applied to the model. With the increased texture pool, the texture group may be resized to fit as many textures available. A model's head textures can be mixed and matched with its various

body textures, allowing the classroom to look stylized, yet allow for greater visual diversity in the classroom. Following is an expanded texture group for the boy NPC:

```
$texturegroup “boy_textures”{
        {“boyBody0”, “boyHead0”}
        {“boyBody1”, “boyHead0”}
        {“boyBody2”, “boyHead0”}
        {“boyBody3”, “boyHead1”}
        {“boyBody4”, “boyHead1”}
        {“boyBody0”, “boyHead1”}
        {“boyBody1”, “boyHead2”}
        {“boyBody2”, “boyHead2”}
        {“boyBody3”, “boyHead2”}
        {“boyBody4”, “boyHead3”}
        {“boyBody0”, “boyHead3”}
        {“boyBody1”, “boyHead3”}
        {“boyBody2”, “boyHead3”}}
```

A total of 13 head-body pairs are present, but could be increased when more become available. Each pair is given an integer index that may be chosen via the Model Browser form within Hammer during map editing.

Before a texture is actually applied to a model, a material file description in the form of a Valve Material File (VMT) text file is created. The VMT provides material descriptions for naming, surface type, shading parameters, and other information that is used during compiling of the QC. Figure B.2.1 shows the VMT for the boy model used in Room 309. Special care should be made when preparing or handling any pre-compiled files, or else problems may occur when a map loads. For example, while increasing the texture pool during development, a typo was made within a VMT file description. Figure B.2.2 shows the result of the type where a model had a head texture applied, but a checkered body. The checkered body was due to the texture not being available when the map was loaded.

```
//=========================================================
// boyBody.vmt
// The material description file for the boy_automata model
//=========================================================
"VertexLitGeneric"{
        "$basetexture" "models/Education/boy/boyBody"
        "$bumpmap" "models/Education/boy/boyBody_normal"
        "$nodecal" "1"
        "$model" "1"
        "$phong" "1"                            //Enable Phong surfacing
        "$phongexponent" "10"                   // Phong exponent for local specular lights.
        "$phongboost" ".5"                      // Phong over-brightening factor.
        "$phongfresnelranges" "[.2 1 8]"        //Fresnal remapping
        "$halflambert" "1"
        "$rimlight" "1"                         //Rim lighting (for Phong)
        "$rimlightexponent" "4"                 //Rim lighting exponent
        "$rimlightboost" "4"                    //Ambient rim lighting boost
        "$cloakPassEnabled" "1"
        "360?$color2" "[ 0.8 0.8 0.8 ]"
}
```

Fig. B.2.1. Material file description for a boy's MDL.

Fig. B.2.2. Texture missing from girl's body from simple error.

B.3. NPC Animation

In the Valve® Source™ engine, an NPC may only run one animation at a time, so even if a behavior model can account for a fuzzy state behavior, the final visual result would still have to be a discreet visual representation. It would be impossible to create a single animation for every possible fuzzy state, so using a blend of multiple animations could help represent each component of the fuzzy state. Due to specifications of the Source engine, the emergent system described in this work is unable to utilize animation blending in providing a visual state representation for NPCs, yet is a consideration of future work via implementation within other game engines.

The first step in creating an animation is creating a character model, and fitting this model to an animation rig. Since this research builds off of the scripted version of Room 309, a small pool of textured models is already available. If new models were required, we would first create them in a preferred polygon editor and then import as an OBJ into XSI ModTools for animation. ModTools provides several useful tools, including a biped rig. Figure B.3.1 shows the biped rig as loaded in the Autodesk® SoftImage® Mod Tool™ software. The biped rig provides user controllable handles that can be easy manipulated in 3D space for key framing joints along the skeleton. Fitting the student model to the skeleton rig is a process called skinning that ensures polygon mesh deformations happen nicely at joint locations.

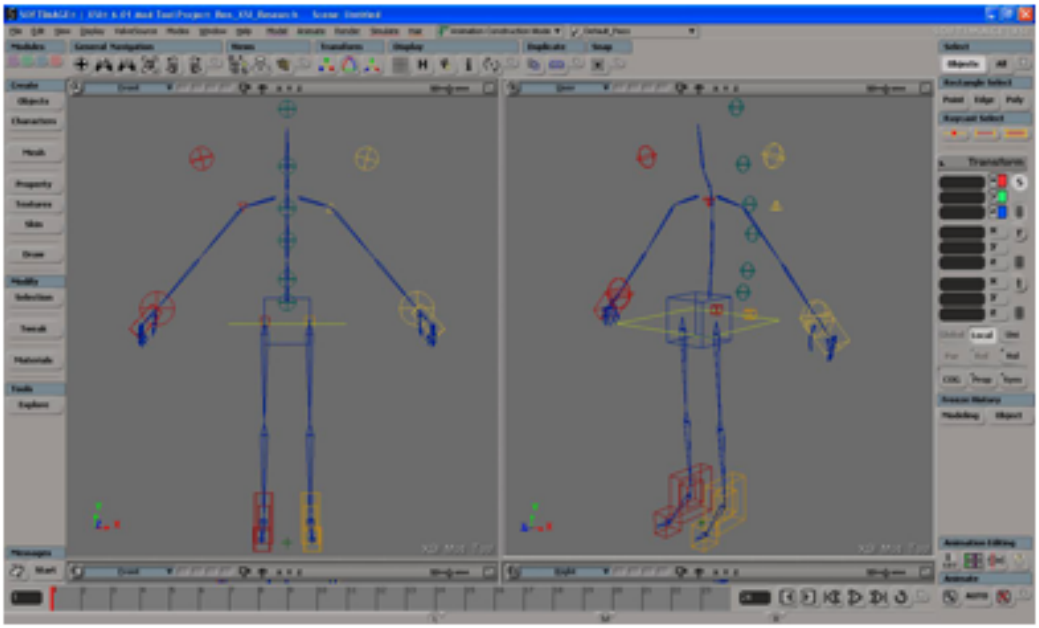


Fig. B.3.1. Front and perspective views of Valve's biped rig.

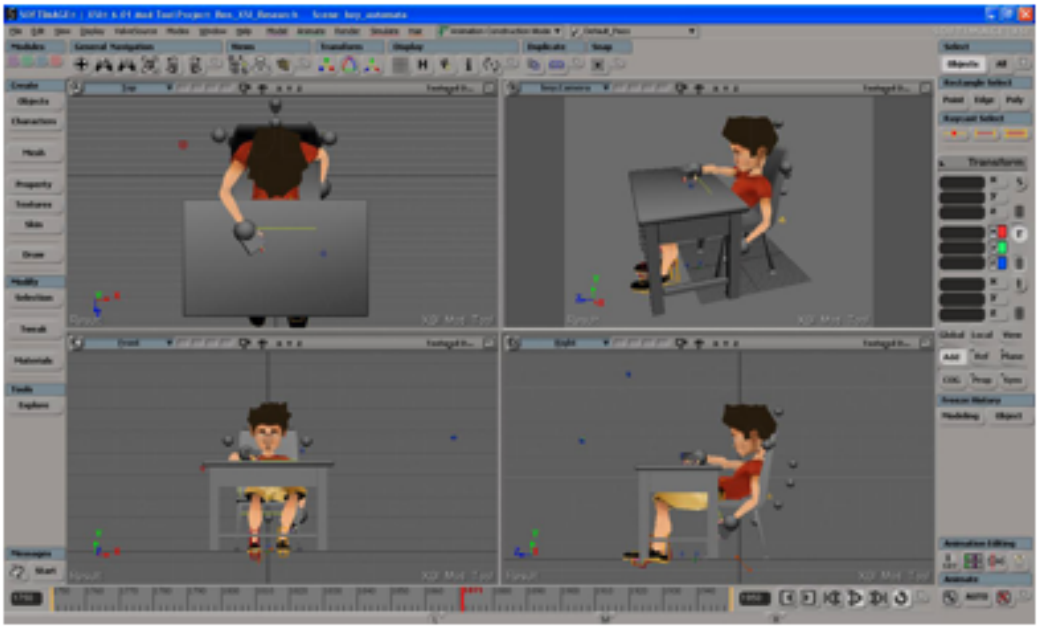


Fig. B.3.2. NPC student in tired pose.

Skinning requires some time and attention, and is one of the more tedious procedures in rigging for animation. With a skinned character rig we can create the default pose for an NPC student and create various animations for the behaviors defined for the model. Figure B.3.2 shows a character posed in a tired pose.

Each animation has its own time range in a single scene. Once the rig has been animated we can use one of Valve's plug-ins included with the Mod Tool™ software to export a Studio Model Data (SMD) file for each particular animation. Figure B.3.3 shows where to find the Valve SMD export option. An SMD is a text file format that describes bone transformation data for each bone in the rig, and for each frame in the animation cycle. The Source SDK provides a command line tool called studiomdl.exe that we will use later to compile the SMD with other animation cycles for a particular NPC. As mentioned above, only a single scene is all that is needed to define each animation sequence. Setting the scene's time slider to the desired frame range of a particular cycle is all we do before invoking the Export SMD plug-in.

Fig. B.3.3. The Export SMD menu option.

The Valve Biped rig is generic and allows us to create animations for any character sharing this bone structure. As we shall see below, using the generic rig will help us when it comes time to create a classroom of both male and female students.

While SMD's contain the animation data used by the NPCs, the model that will be brought into the level editor must first de described with a Quake C (QC) script file. The QC file contains information about the model such as name, materials, eye position, physics rules, bone, and animations. The $include option is used in the QC to include the QCI file which stores references to more information about the skeleton and the animation data. It is in the QCI file that the animation sequences are referenced by file name, and given global names, such as ACT_BOY_TIRED, used within the level editor Figure B.3.4 shows a sample QC and QCI file for a male character in Room 309.

Since both the male and female NPCs utilize the biped rig, they have identical skeletons; hence, we can make use of the same SMD data referenced in the female's QCI as in the boy's. We create a QC file for the female that references her own QCI file, yet makes references to the boy's SMD animation files. The $sequence option in the female QCI only has to create new variable names and values. In creating a whole classroom of students, only a master set of animations needs to be created, then each NPC can use any combination of relevant skeletal animations.

Once the SMD, QC, and QCI files are properly setup for a model, we can invoke Valve's command line tool, studiomdl.exe that compiles these files into Valve's proprietary model (MDL) format. The final MDL is then used as the model parameter for the random_generic_actor entity. Both a separate male and female MDL file is compiled for use in Room 309.

```
//=============================================================
// boy_automata.qc: A sample QC File for boy_automata.mdl
//=============================================================
    $modelname Education/boy/boy_automata.mdl
    $cdmaterials models/Education/boy
    $texturegroup “boy_types”{
        {“boyBody0”, “boyHead0”}
        {“boyBody1”, “boyHead1”}
        {“boyBody2”, “boyHead2”}
        {“boyBody3”, “boyHead3”} }
    $eyeposition 0 0 70
    $mostlyopaque
    $include "../standardhierarchy.qci"
    $include "../commonbones.qci"
    $include "../ruleshierarchy.qci"
    $include "boy_ragdoll.qci"
    $proceduralbones "../male.vrd"
    $pushd "../male_animations_sdk"
    $include "../male_animations_sdk/boy_automata.qci"
    $popd


//=============================================================
// boy_automata.qci: A sample QCI File for boy_automata.mdl
//=============================================================
    $sequence boyIdleTired "boy_automata_Tired" ACT_BOY_TIRED 28
    $sequence boyIdleBored "boy_automata_Bored" ACT_BOY_BORED 5
    $sequence boyIdleDistracted "boy_automata_Distracted" ACT_BOY_DIST 5
    $sequence boyIdleWorking "boy_automata_Working" ACT_BOY_WORK 5
    $sequence boyIdleWorking "boy_automata_Alert" ACT_BOY_ALERT 5
    $sequence boyIdleWorking "boy_automata_TapL" ACT_BOY_TAPL 5
    $sequence boyIdleWorking "boy_automata_TapR" ACT_BOY_TAPR 5
    $sequence boyIdleWorking "boy_automata_ShowL" ACT_BOY_SHOW_L 5
    $sequence boyIdleWorking "boy_automata_ShowR" ACT_BOY_SHOW_R 5
    $sequence boyIdleWorking "boy_automata_IdleL" ACT_BOY_IDLE_L 5
    $sequence boyIdleWorking "boy_automata_IdleR" ACT_BOY_IDLE_R 5
    $sequence boyIdleWorking "boy_automata_IdleB" ACT_BOY_IDLE_B 5
    $sequence boyIdleWorking "boy_automata_RaiseL" ACT_BOY_RAISE_L 5
    $sequence boyIdleWorking "boy_automata_RaiseR" ACT_BOY_RAISE_R 5
```

Fig B.3.4. Sample QC and QCI file for a character model. The second argument to the $sequence option refers to the exported name of the actual SMD animation file.

B.4. Triggers in Room 309

The triggers used in Room 309 are trigger_multiple entities and are very useful for general mapping. Each NPC may have its own trigger box assigned to it, with the index of the trigger matching the index of the NPC. The triggers overlap, thus affecting more than one NPC at once. When the player engages a trigger, the trigger's index is sent to the automata_logic entity for processing the environmental update event. To allow triggers to send environmental update messages to the automata_logic_entity, the FGD entry for the automata entity contains an environmental update input function called UpdateEnvironment(). Similarly, the CAutomataLogic class contains an environmental update function UpdateEnvironmentCommand(), which is linked to the corresponding UpdateEnvironment() input function in the FGD entry. This modification allows a trigger to be linked and ready to notify the automata_logic entity it wants to update the environment. When a trigger event occurs, the UpdateEnvironment() input function calls the UpdateEnvironmentCommand() function with the input parameter set to the index of the trigger that corresponds to both the NPC, and influence map location. The Automata class functionality allows this notification with an environment update function of its own called updateEnvironment() that takes the trigger index, and updates the environmental state to an appropriate state for the present scenario in the game.

APPENDIX C

C.1. Cheating Choreography

Every NPC has enough scripted_sequence entities in the map that any left-to-right pair of NPCs is capable of acting out a cheating scenario. In order to keep track of and handle cheating without scripting it into the map directly, the following methods and data structures are included in with the Automata class:

- cheatAge – This is an integer variable that keeps track of whether or not cheating is occurring in the game. A value greater than zero means that cheating is occurring, with the value encoding the phase within the cheating choreography.
- findDistractedPair() - This function is called to randomly find a suitable distracted pair of students for the choreography when cheatAage is zero. In this function, a random student is chosen and checked to determine if they are in a distracted state. A successful test is more probable when there are a greater number of distracted students in the classroom than there are non-distracted.
- cheatDirection – This is an integer value that keeps track of the cheating direction.
- incrementCheatingRight() – This function increments the cheating choreography to the right from the student sitting on the left.
- incrementCheatingLeft() – This function increments the cheating choreography to the left from the student sitting on the right.

When the next generation of states is being computed on a map, the getNextGen() function of the Automata class checks whether or not there is cheating happening in the room. If cheatAge is zero, then findDistractedPair() is called, and if a

distracted student is found, cheating is set to cheat in either the left or right direction, with cheatAge incremented to a value of one. Otherwise, if cheatAge is greater than zero, the direction of cheating is checked and either incrementCheatingRight(), or incrementCheatingLeft() is called to move the cheating choreography forward another step.

C.2. Decision Panel

The decision panel is a generic SDK feature with added functionality that notifies a score keeping class which button selections have been made. Player choices are logged into a text file, and reviewed by administrators of the game. Implementation of panels in the Source engine are discussed on the Valve Developer website http://developer.valvesoftware.com. The decision panel's use is restricted during the necessary conditions for cheating, so the game environment must be aware weather or not a student in the room has cheated, and if the player is near one of those students. This procedure is slightly different from the other modifications because up until now, all the source code implementation has occurred within the game server. The Source engine contains both a client and server that can be modified, yet are separate compiled files. Code for the Automata class and entities are implemented in the server code, while code for panels is implemented in the client code. This can be a problem for development if code from the server needs to utilize code from the client. In the case of Room 309, the decision panel relies on knowing who is cheating and weather or not the player is near, yet doesn't have immediate access to the Automata class or automata_logic entiy. To create a check for these conditions, a point_clientcommand entity is used which facilitates the execution of console commands within the game.

A console command is a type of command that may be executed directly by a developer through a console window during game play, or via entities within the game environment. This type of command is useful when creating generic functions that do not require hard coded data classes such as the checks for decision panel display

conditions. The point_clientcommand entities use a console command function as an argument in order to perform certain tasks within the game. In Room 309, a point_clientcommand named CheatPanelNotify is placed into the map. The scripted_sequences for cheating animations link to CheatPanelNotify, calling console commands that keep track of who has already cheated. Figure C.2.1 shows a scripted_sequence sending an output to CheatPanelNotify for letting the game environment know that NPC number 10 is in a cheating animation sequence. The parameter override is the name of the console command that keeps track of the corresponding NPC.

At the start of the cheating sequence, when the tapping animation starts, the scripted_sequence tells the CheatPanelNotify entity to issue a console command called CheatingBegin() that records the NPC who started cheating into a boolean array called hasCheated. If the player navigates through a trigger box, the trigger tells CheatPanelNotify to issue a console command called CheatingRange() that records whether or not the player is in close proximity to a student that has cheated by updating a boolean variable called inRange, which is only updated if the NPC has already cheated. Figure C.2.2 shows a trigger sending an output to CheatPanelNotify for letting the game environment know that the player is in legal distance for the decision panel to appear.

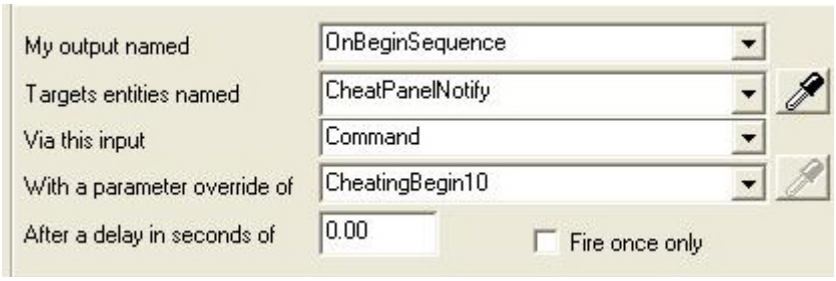


Fig. C.2.1. Output fired from a *scripted_sequence* commencing the beginning of cheating choreography for NPC number 10.

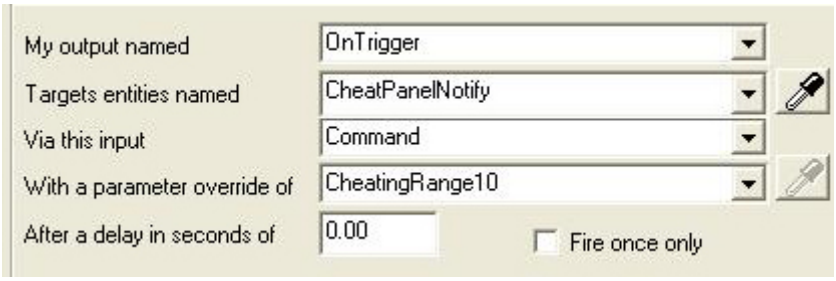


Fig. C.2.2. Output fired from a trigger when the player is in range of NPC number 10.

When the player decides its time to display the decision panel, a console command linked to a keyboard press named ToggleAutomataCheatPanel is called that checks to see if inRange has been set to true. If it has, then the panel will display. Figure C.2.3 illustrates how hasCheated and inRange are used to satisfy the display panel conditions within the console commands; however, for brevity, only the console commands for one NPC are shown, with the index for the NPC embedded directly into the console command function name, e.g. CheatingBegin0, and CheatingRange0. While this approach works well, the number of console commands to add to the client code is twice that of the number of NPCs in the map.

```
CON_COMMAND(CheatingBegin10, NULL){
	hasCheated[0][0] = 1;
}

CON_COMMAND(CheatingRange10, NULL){
	if(hasCheated[0][0] == 1){
	  inRange = 1;
	}else{
	  inRange = 0;
	}
}
CON_COMMAND(ToggleAutomataCheatPanel,NULL){
	if(inRange){
	  ToggleVisibility(cheat->GetPanel());
	}
}
```

Fig. C.2.3. Console command functions used in the cheating decision panel. For brevity, only the *CheatingBegin*, and *CheatingRange* functions for NPC no. 10 are shown.

C.3. Automata Stats Entity

To create an automata statistics entity an FGD entry is created and named automata_stats. The only input function this entity contains is DisplayData(), which takes an input integer corresponding to the desired message to display. Figure C.3.1 shows the FGD file for the automata_stats entity. The CGameText class is able to handle displaying messages on screen, so it is extended with a new class called CAutomataStats that is specifically designed to retrieve and display data from the Automata class. This extended class derives all its functionality from CGameText hence the only new feature added is to query the Automata class for relevant data. Figure C.3.2 shows the class definition for the CAutomataStats class.

Entities linked to an automata_stats entity will fire outputs to the DisplayData() input, with an integer signifying which piece of Automata data it wishes to retrieve. The native functionality of game_text only permits one message to be displayed at a time,

hence there must be a seperate automata_stats entity for each on-screen message. A logic_timer lets the automata_stats entity know when to retrieve this information. Figure C.3.3 shows four automata_stats entities and a logic_timer. The timer continuously fires outputs to these entities so that game data is continuously displayed on screen.

```
//=============================================================
// Entity: automata_stats
//=============================================================

@PointClass base(game_text) iconsprite("entity/automata_stats.vmt") =
automata_stats : "An entity that displays Automata Stats on player's screens."
[
        input DisplayData(integer) : "Various automata stats for inputs 0-3."
]
```

Fig. C.3.1. FGD entry for the *automata_stats* entity.

```
//=============================================================
// Class: CAutomataStats
// Purpose: Interface to game statistics provided by the automata class
//=============================================================

class CAutomataStats : public CGameText{
public:
  DECLARE_CLASS(CAutomataStats, CGameText);
  DECLARE_DATADESC();
  void InputDisplayData( inputdata_t &inputdata );
};
LINK_ENTITY_TO_CLASS( automata_stats, CAutomataStats );
BEGIN_DATADESC( CAutomataStats )
DEFINE_INPUTFUNC( FIELD_INTEGER, "DisplayData", InputDisplayData ),
END_DATADESC()
```

Fig. C.3.2. CAutomataStats class definition.

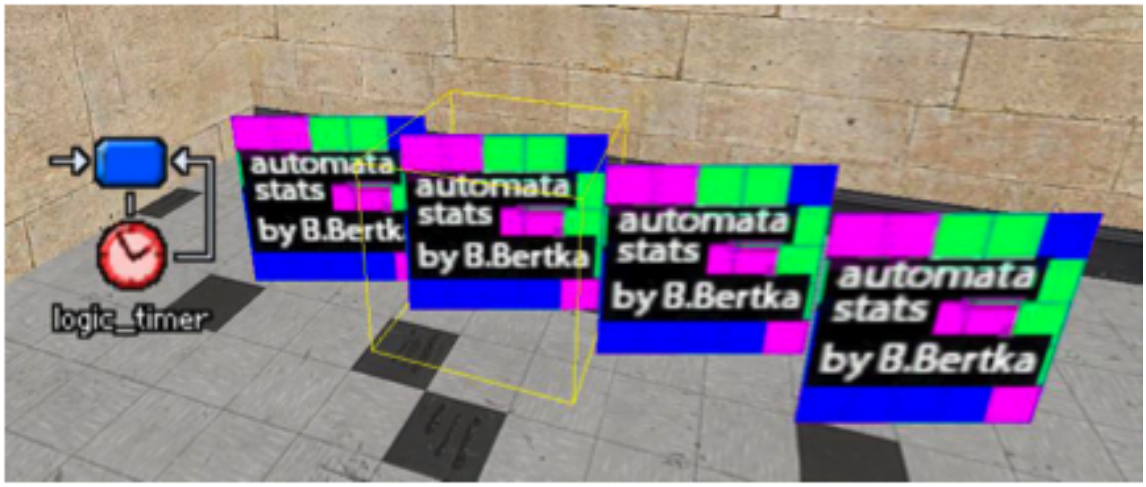


Fig. C.3.3. Four game statistics entities controlling dynamic text on screen.

VITA

Name:	Benjamin Theodore Bertka
Address:	Texas A&M University, C108 Langford Center College Station, TX 77843-3137
Email Address:	bbertka@gmail.com
Education:	B.A., Mathematics, The University of California at Santa Cruz, 2008 M.S., Visualization Sciences, Texas A&M University, 2010