Generating Narrative Through Intelligent Agents in Digital Games

by

Naomi A. Hinchen

S.B., Computer Science
Massachusetts Institute of Technology, 2011

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Computer Science

at the

Massachusetts Institute of Technology

September 2012

ARCHIVES

© 2012 Naomi Hinchen. All rights reserved.

The author hereby grants to MIT permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole or in part in any medium now known or hereafter created.

Author: ______________________________

Department of Electrical Engineering and Computer Science
August 22, 2012

Certified by: ______________________________

Clara Fernández-Vara
Postdoctoral Researcher, Department of Comparative Media Studies
Thesis Supervisor
August 22, 2012

Certified by: ______________________________

Philip Tan
Department of Comparative Media Studies
Thesis Co-Supervisor
August 22, 2012

Accepted by: ______________________________

Prof. Dennis M. Freeman
Chairman, Masters of Engineering Thesis Committee

Generating Narrative Through Intelligent Agents in Digital Games

by

Naomi A. Hinchen

Submitted to the Department of Electrical Engineering and Computer Science on August 22, 2012 in Partial Fulfillment of the Requirements for the Degree of Master of Engineering in Computer Science

Abstract:

An ongoing problem in game design is how to create story-based games that allow the player to have a variety of experiences if the game is played more than once, preferably without burdening the designer with a prohibitive workload. In this project, I approach the problem of creating a game with a mutable narrative from an AI perspective, designing a system called CharacterSimulator that generates a population of non-player characters (NPCs) with which the player will interact and assigns the NPCs a set of goals to carry out. Varying the set of NPCs and their objectives will create a different narrative experience for the player when the game is replayed. Although the NPC behaviors were originally modeled on Braitenberg vehicles, I have largely moved away from that model in the final version, focusing more on assigning NPCs goals that result in narratively interesting interactions.

Thesis Supervisor: Clara Fernández-Vara
Title: Postdoctoral Researcher

Acknowledgements

Thanks to my thesis advisor, Clara Fernández-Vara, for her help and guidance; Andrew Grant and Owen Macindoe for programming advice and bug-squashing help; and the rest of the Singapore-MIT GAMBIT Game Lab staff. And a general thank you to the many, many people who have patiently listened to me ramble on about this project and offered advice and brainstorming help.

Contents

Acknowledgements 3
Introduction 5
Background and Previous Work 5
Project Goals 10
High-Level Design 11
Early Design: Braitenberg Vehicles 13
Vehicle Types, Missions, and MacGuffins 14
The Player's Mission and Available Verbs 16
Story Events and the Newsfeed 17
Generating the NPC Population 20
Conclusion 21
Ideas for Future Development 23
Works Cited 25
Appendix A: Attached CD 26
Appendix B: Source Code 27

Introduction

One of the goals of video game design is that games should be replayable: players should be motivated to play more than once. This is no problem in action-based games, which encourage players to improve on their previous performance, or multiplayer games, where the challenge comes from facing different opponents. However, in heavily story-based genres such as interactive fiction or adventure games, gameplay is so strongly tied to the narrative of the game that the player's actions introduce little variation. After players have finished the game once, they have little motivation to play again, because the second playthrough will be very much like the first; for instance, advancing the narrative may require the player to complete a puzzle that always has the same solution.

In an attempt to address this problem, I designed the CharacterSimulator system to generate a population of non-player characters (NPCs) with varying behaviors, which will interact differently with the player character (PC) and with each other depending on the goals they are assigned. The narrative of the game is defined by the actions of the NPCs as well as the actions of the player, and has the potential to vary greatly between iterations of the game.

Background and Previous Work

A common method of designing a replayable story-based game is allowing the player's choices to affect the story, such that a game can be run many times with each playthrough leading to a different narrative. The easiest and most immediately obvious way to do this is to give the player a branching tree of choices, similar to the structure of a Choose-Your-Own-Adventure novel. Chris Crawford discusses some of the problems of this approach in [1]: in a

game where the player makes n choices and has two options at each decision point, the game designer must write 2^n different outcomes. The designer's workload will increase exponentially for only a linear increase in the player's number of choices, even assuming that each choice presents the player with only two options. One way of easing the designer's load is to include some choices that don't immediately affect what happens next, but do affect player stats in a way that will matter later; the company Choice of Games has produced a number of games using this structure, such as *Choice of Broadsides* [2]. This design allows the player to make more choices without requiring quite as much work from the designer, but still requires a large number of branches to allow the player's actions to meaningfully affect events. (Use of the player's stats to determine the success or failure of their actions is a technique borrowed from role-playing games. While role-playing games are designed to allow the narrative to go in many possible directions based on the players' actions, in practice this works best with tabletop and live action role-playing games with a live gamemaster present.)

The branching approach does at least have the advantage of allowing the designer complete control over the narrative arc. While the player can choose any one of a number of paths, the designer can make sure that any single path from beginning to end will constitute a coherent story with conflict, dramatic tension, and a definitive ending. By contrast, in sandbox games such as *The Sims* [3], the designer doesn't write a narrative at all, but creates a world in which players are free to experiment as they please. In *The Sims*, the player can dictate nearly everything the characters do, from their jobs to when they eat dinner to how they raise their children. The characters can act autonomously if not given other instructions, and can occasionally veto a player's commands, but for the most part the player is in control; what story the game has is defined by what the player chooses to do. While this does result in a game with

high replay value, it's less a narrative-based game than a set of dolls that the player can move around. The player's near-omnipotence, allowing them to step in and play puppetmaster at any time, robs the events of the game of what meaning they might otherwise have had.

Previous attempts have been made to strike a balance between these approaches. One notable example is Mateas and Stern's *Façade* [4], which was explicitly designed to combine the freedom of player action (also called *agency*) found in sandbox-style games like *The Sims* with the more unified plot of a game with an explicitly designed narrative. In the designers' own words:

> Game designs are moving towards rich, complex worlds offering open-ended, sandbox-style play. At the same time, we know that narrative structures (stories) have historically been an extremely successful way of representing human relationships. In order to create interactive experiences about human relationships, it makes sense to create a system that tries to shape open-ended play into narrative structures. That is, metaphorically speaking, to offer open-ended sandbox-style play in which the system knows how to make sandcastles, and tries to collaborate with the player to do so. [5]

Façade is run by a *drama manager*, an AI that guides the narrative of the game to resemble a preconceived story structure. The player interacts with two NPCs, Trip and Grace. The drama manager organizes the game as a series of what Mateas and Stern call *story beats*: microscenes within the story that define what behaviors Trip and Grace might perform at this point in their conversation with the player. Within the parameters of the current story beat, the NPCs can respond to the player's actions with dialogue and physical movements. The player's actions, and the subsequent reactions of Trip and Grace, affect the sequence of story beats chosen by the drama manager, culminating in one of several possible endings.

While Mateas and Stern succeeded in creating a game with a multitude of story possibilities without the limitations of a branching structure, they did not solve the workload

problem. *Façade* has a pool of about 200 story beats, within each of which the NPCs can react in different ways based on the player's actions, and the player has a range of about three dozen actions to choose from. This gives the player a high degree of agency, but at the price of requiring the designer to account for everything the player might do and determine appropriate reactions for the NPCs.

Façade is also constrained by the creators' stricture that the story had to adhere to a traditional Aristotelian dramatic arc. This is the primary reason for the complexity of the drama management system; the finished narrative is required to follow a pattern of rising tension, climax and denouement. While this guarantees the sort of narrative coherence that sandbox games usually lack, it requires the drama manager to be carefully constructed to ensure that the narrative conforms to this pattern.

The Snowfield [6], a game developed by the Singapore-MIT GAMBIT Game Lab, opts for a less structured approach. Rather than the drama management system favored by *Façade*, *The Snowfield* has no formally structured narrative at all. Instead, it relies on *emergent narrative*: a type of story that, rather than being explicitly designed, develops from the interaction of relatively simple independent elements causing complex results.

The Snowfield is particularly notable in the context of my research because I was a member of the team that worked on it. Among the aims of the project was to build a game that could generate a complex narrative without restricting ourselves by attempting to model the story on a predefined pattern. This was based on the ideas put forth by our product owner, Matt Weise:

> *The Snowfield* was a response to the idea, in both industry and academia, that there are principles of good storytelling that we need to systematize in order to marry video games with narrative effectively. This has led to ideas like "drama

> management," which assumes that emergent narratives require an A.I. storyteller who controls events to ensure the emerging story is satisfying... I wanted *The Snowfield* to illustrate how we could build an emergent, emotionally rich, player-driven narrative experience from the ground up by relying solely on art, audio, and character A.I. (refined via player testing), instead of trying to identify formulaic notions of "good" storytelling and work them into the game beforehand. [7]

In the finished game, the player character is a WWI soldier wandering a snowy battlefield, encountering a few NPC soldiers along the way. We used art and sound to build an evocative environment intended to invest the events of the game with meaning by provoking an emotional response from the player.

In the end, while we succeeded in getting players to respond to the game emotionally, the narrative possibilities of *The Snowfield* are limited. We were unable to implement all the NPC behaviors we had designed during the prototyping stage, which limits the scope of events that can happen in the course of the game. The player can still assemble those events into a narrative ("I looked for the letter to give to that one soldier, but he froze to death first, so then I found the soldier with the harmonica and... "), but the potential for the narrative to change on repeat plays is minimal. Each NPC soldier will react to the player character the same way every time: following the PC if the soldier has found the object he's looking for, and otherwise either staying put or roaming the field. The action of the game consists largely of repeatedly herding soldiers back to the house, since the closest thing *The Snowfield* has to a win condition is to save as many soldiers as possible from dying of exposure. Furthermore, since the soldiers start the game in fixed locations and die after a set amount of time, it's only possible to save everyone by collecting them in a certain order, meaning that one "winning" game looks almost exactly like another.

Project Goals

In light of the differing philosophies behind *Façade* and *The Snowfield*, I had to consider carefully what I was trying to achieve with CharacterSimulator, which aspects were most important, and what I thought qualified as a story. I chose to favor emergent narrative over drama management. Like Weise, I don't believe that storytelling necessarily requires the kind of rigid narrative structure that *Façade* attempts to model, but I wanted the general direction of the story to come from the designer, unlike in a freeform sandbox environment like *The Sims*. I was also less concerned with promoting agency than the creators of *Façade*; it was not necessary that the differences in game narrative result from player choices. In fact, I preferred to focus on generating new possibilities through elements outside of the player's control, to all but guarantee that the narrative would vary every time.

I decided to define the story or narrative of the game as the sequence of events that occurs during play. While this may seem simplistic, it has some important implications. First of all, this definition does not specify that the story conform to any particular structure. It is therefore extremely compatible with a reliance on emergent narrative, which is by its nature unpredictable. This definition also emphasizes what actually happens during the game over the emotional impact on the player, placing the focus of the project on the underlying game design and AI rather than on audiovisual elements. Finally, if the goal is to create a game where the story changes every time the game is played, this definition makes it easy to measure success: a different sequence of events is a different story. Later, I will discuss how I defined an "event" in this context.

Given this definition, my goals for this project were threefold. In order of importance:

1. **Create a game in which the story is different every time it is played.** By the above definition of story, this meant that a different sequence of events had to occur every time.
2. **Avoid the designer workload problem.** The system would ideally be designed in such a way that relatively little work on the game designer's part would result in many possible narratives. This may seem unrealistic when looking at the examples set by *Façade* and the many branching-narrative games, but I took inspiration from the board game *Clue* [8]. The solution to a game of *Clue* consists of three cards: a suspect, a room, and a weapon (e.g., "It was Miss Scarlet in the library with the knife!"). There are six suspects, nine rooms, and six weapons, totaling 21 cards, but those 21 cards form 324 possible combinations. Adding just one more room would add 36 new possibilities; one more suspect or weapon would add 54. This is a very simple example of how creatively combining elements of a game can yield a lot of value for little effort.
3. **Aim for reusability.** I considered the game a proof of concept. Ideally, even if the exact code I wrote was not reusable, I wanted it to be possible for someone else to use something similar to CharacterSimulator to build a game with a completely different story and characters. This was less important than the other two goals, but still a factor in some of my design choices.

High-Level Design

One element *Façade* has in common with branching narrative games is that potential changes in the narrative are primarily dependent on player choices. This contributes to the designer workload problem, since the designer must account for every choice the player could possibly make. I decided to take the less well-explored avenue of focusing on the non-player characters and their role in the story. While the player is still a participant in the narrative, the story is not solely defined by his or her actions.

To that end, I wanted to build multiple types of autonomous NPCs with their own goals and agendas, which would interact with the player and with each other. I also wanted to incorporate a certain amount of randomness into generating the NPC population, to introduce more variation between iterations of the game. This approach had the side effect of decentralizing the narrative; rather than being run by a central "brain" like *Façade*'s drama manager, it was composed of the actions of a multitude of independent agents.

I made some decisions early on to limit the scope of the CharacterSimulator system; I excluded dialogue entirely, and limited the graphics to simple geometric shapes. Initially, my intention was to define each character as a collection of motivations and reactions to stimuli by modeling characters on Braitenberg vehicles, as discussed in the next section. Later in the process, I moved away from this model in favor of assigning NPCs specific missions that they would work towards completing. In designing missions, I favored ones that involved interactions with other characters.

Since the project was all about narrative, I felt that it was important to decide early on what kind of story I wanted the game to tell. For my purposes, the story had to allow for a large and diverse group of characters to mingle in the same room, and the characters' interactions in this setting had to drive the plot. The second criterion proved more difficult than the first. For example, I thought of having a group of superheroes in a room together, but superhero stories are about what happens when they go out to fight supervillains, not about what happens during Justice League meetings. Court intrigue seemed like a good idea until I realized that it's about playing politics, which is hard to represent without complex dialogue.

Eventually I settled on a spy story, which worked well with my requirements for several reasons. It allowed me to bring together a large group of characters who would outwardly tolerate each other's presences while secretly pursuing their own potentially conflicting agendas. There was a variety of small yet plot-driving actions the characters might perform, such as picking another character's pocket. And there was lots of room to give characters hidden motivations that might affect their actions, motivations which could easily be changed when the game was replayed.

I chose to implement CharacterSimulator in ActionScript 3 using Flash Builder. This was partly because of my previous experience with Flash, and partly because it lends itself easily to creating simple graphics. The main class, Frame, inherits from Sprite and displays one of three different states, which are also Sprites: UserOptions, VehicleSim, and EndGame. UserOptions allows the player to customize game variables, VehicleSim runs the actual game, and EndGame displays a closing screen indicating whether the player won or lost. In practice, the SpySim subclass is used for VehicleSim.

Early Design: Braitenberg Vehicles

As a first step, I modeled the early versions of my NPCs on Braitenberg vehicles. Braitenberg vehicles, as initially conceived by Valentino Braitenberg in [9], are simple artificial intelligence agents that move in response to their environment. A Braitenberg vehicle consists of a body attached to two wheels, each driven by a motor, and some number of sensors, each connected to at least one of the motors. Based on the input to the sensors from external stimuli,

the wheels turn at a greater or lesser speed. Given fairly simple rules of movement, Braitenberg vehicles exhibit surprisingly complex behaviors.

Although Braitenberg vehicles can be designed to respond to any number of external stimuli, I was most interested in having them respond to their "social environment," i.e. the states and traits of the other vehicles surrounding them. For example, "friendly" vehicles would gravitate towards large groups of other vehicles, while "shy" vehicles would prefer to be alone.

The Braitenberg model proved to have advantages and disadvantages. It allowed vehicles to react simultaneously to a variety of stimuli, but while this sometimes produced interesting emergent behaviors, the model did not provide for the vehicles to perform actions other than moving and sensing their environment. I needed the NPC vehicles to be forced to interact, not just react. For this reason, I moved away from Braitenberg's model towards designing vehicle types that made direct efforts to achieve a specific mission. However, since I kept some aspects of Braitenberg's design, such as modeling each vehicle as having left and right wheels and two sensors in front as eyes, the term "vehicle" will henceforth be used interchangeably with "character."

The VehicleSim class has a list of all vehicles in the game. Every frame, each vehicle senses its surroundings, then updates its left and right speeds accordingly.

Vehicle Types, Missions, and MacGuffins

To make vehicles easier for the player to distinguish, each one is randomly assigned a name from the NATO phonetic alphabet. If there are more than 26 vehicles, a number is added to differentiate vehicles with duplicate names (Alpha1, Alpha2, etc.).

The four types of NPCs in the final version of the game are assassins (AssassinVehicle), thieves (MacguffinSeekVehicle), kleptomaniacs (KleptoVehicle), and civilians (WanderVehicle). Civilians are the simplest type; they simply wander around the space, colliding with physical obstacles but otherwise oblivious to their surroundings. The other three all inherit from the PursuitVehicle abstract class. A PursuitVehicle explores the space looking for another vehicle that fits some criterion, and performs a specified action upon finding it.

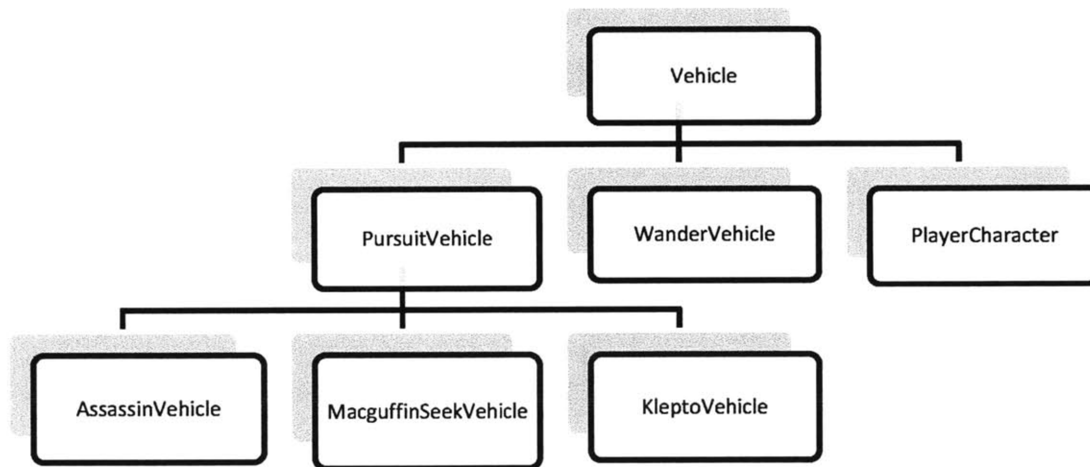

Figure 1: Hierarchy of vehicle class inheritance. The four NPC types are WanderVehicle, AssassinVehicle, MacguffinSeekVehicle, and KleptoVehicle.

Assassins are given a list of some number of other vehicles, and will attempt to find those characters and kill them. After crossing everyone off its hit list, an assassin will exit the gamespace. Thieves and kleptomaniacs, while they may sound similar, are actually two distinct types, though both of their missions rely on the existence of MacGuffins. MacGuffins are in-game objects that characters can carry around with them and steal from one another. Thieves are assigned a specific MacGuffin (e.g., the Maltese Falcon) to steal, and must find the character

carrying that MacGuffin, successfully pick their pocket, and make it out of the room with the MacGuffin. Kleptomaniacs are less focused; they will simply steal anything they can find.

Vehicles have certain deliberate limitations that make it more difficult for them to achieve their goals. A vehicle can only "see" an object or other vehicle that is within a certain visual range, meaning within a certain radius and in the direction it is facing. Moreover, if, say, vehicle Alpha attempts to perform an act upon vehicle Bravo, such as assassinating it or picking its pocket, Bravo must be within a small radius of Alpha, which is referred to as Alpha's *zone of control*. The vehicle performing the action, in this case Alpha, also has a 20% probability of failure, requiring Alpha to try again and giving Bravo an opportunity to escape.

The Player's Mission and Available Verbs

A single vehicle generated by CharacterSimulator will be of type PlayerCharacter and controlled by the player. The player's *verbs* are the in-game actions he or she can perform: in this case, the verbs are "move forwards" (up arrow), "move backwards" (down arrow), "turn" (left and right arrows), "kill" (shift key), and "pick pocket" (spacebar).

The player can choose to carry out either an assassination mission (kill someone) or a theft mission (steal a MacGuffin). After making this choice, the player is assigned a specific target vehicle or MacGuffin to pursue. The game ends upon either the player's successful completion of the mission and exit from the room, or the player's death, if an assassin has targeted the PC. The player can also voluntarily exit the game by pressing the tab key.

Story Events and the Newsfeed

Earlier, I defined the story of the game as the sequence of events that occurs during the game, but did not go into detail about what qualified as an event. I wanted story events to be significant enough to be relevant to the overall story. To a certain extent, the significance of an event is subjective; "Charlie sees Delta across the room" might be a significant event if Charlie is trying to kill Delta, but probably isn't if they couldn't care less about each other. The loose definition that I adopted for the purposes of my proof-of-concept game was that a significant event represented a character's attempt to change some aspect of the current state of the game, whether or not the attempt was successful. (So, for instance, a failed assassination attempt is considered significant even though the assassin has not succeeded in changing the target's state from living to dead.)

Due to the subjectivity of what constituted a significant event, I set up a system that could flexibly record whichever events the game designer considered important. Whenever something of note happens, some object will dispatch a StoryEvent. At the start of the game, the main game class is given a list of types of StoryEvent for which it should add event listeners, and whenever it gets an event of one of those types, it prints it to an onscreen newsfeed, as seen in Figures 2.1 and 2.2. There is also a second newsfeed on display, which records only events that involve the player directly (such as someone picking the player's pocket) and error messages when the player attempts an invalid action.

For the purposes of my proof-of-concept game, these are the events recorded by the newsfeed:

- Death of a character
- Discovery of a character's corpse by an assassin who would otherwise have targeted that character
- Successful theft of a MacGuffin
- Unsuccessful attempt to kill someone or pick their pocket
- Character enters or exits the gamespace

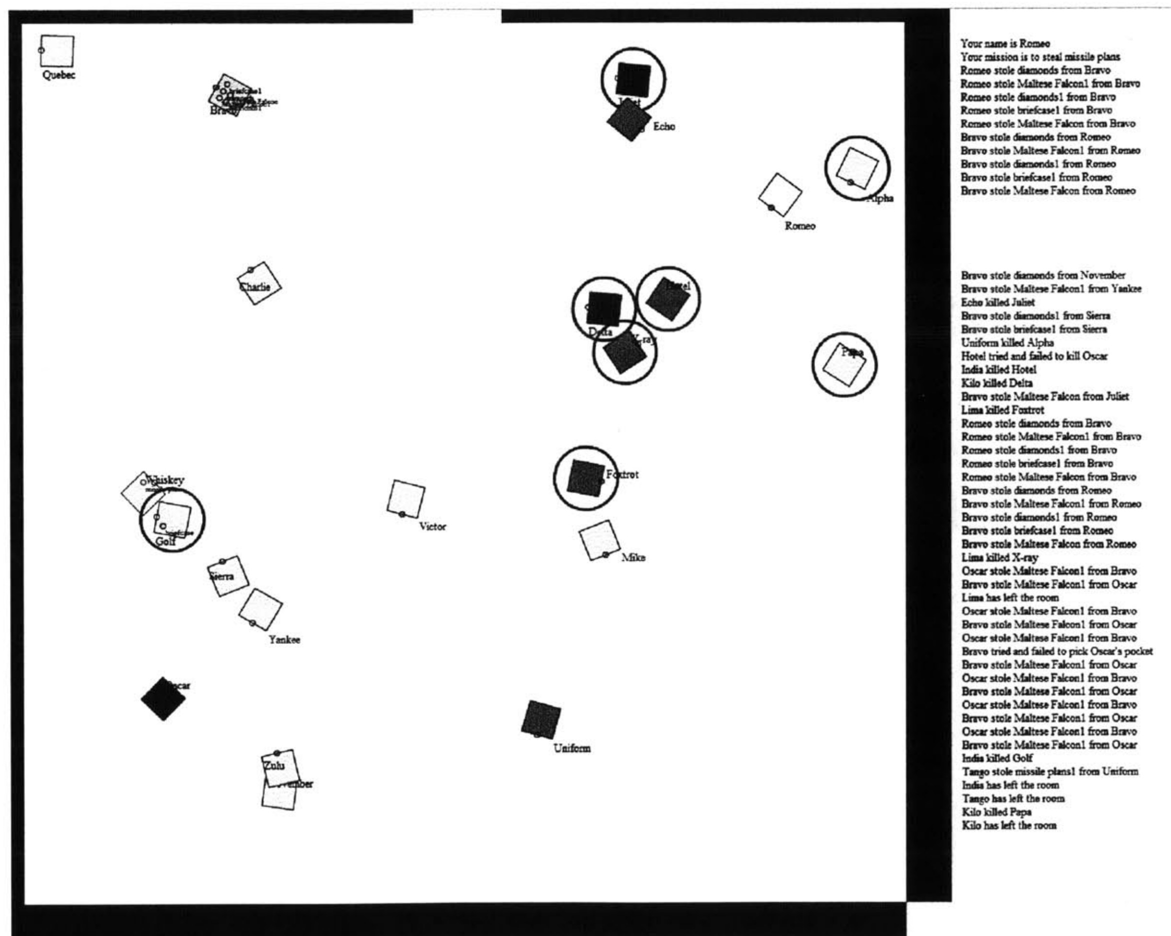

Figure 2.1: The block of text in the upper right corner is the player newsfeed. It lists only events directly related to the player (in this case, vehicle Romeo). The block of text below it is the general newsfeed.

Your name is India
Your mission is to steal diamonds
Can't do that--no targets in range
India stole diamonds from Mike
Echo stole diamonds from India
India tried and failed to pick Echo's pocket
Can't do that--no targets in range
Can't do that--no targets in range

Mike tried and failed to kill Juliet
Mike killed Juliet
November killed Oscar
Sierra killed Whiskey
Lima killed Tango
Echo stole missile plans from Oscar
Mike killed X-ray
Echo stole Maltese Falcon from Sierra
Lima killed Golf
India stole diamonds from Mike
Echo stole diamonds from India
India tried and failed to pick Echo's pocket
Lima killed Foxtrot
Uniform killed Romeo
Lima has left the room
Echo stole diamonds1 from November
Uniform killed Delta
Mike killed Kilo
November tried and failed to kill Yankee
November killed Yankee
Mike has left the room
Echo stole briefcase from X-ray
Sierra found the body of X-ray

Figure 2.2: Newsfeeds from a sample game. Note that all the events in the upper feed are related to the player character, India.

Generating the NPC Population

Now that I had several types of NPCs capable of interacting with each other, I had to think about the proportions in which each type should be present. Through experimentation, I discovered that the composition of the NPC population made a huge difference to the narrative because it could potentially completely change the dynamics of the game. For instance, the configuration I dubbed "murder convention" (all assassins, each with a hit list consisting of all the other characters) generates a literal sudden-death game in which the NPCs compete to be the last one standing. If the population of thieves is high and the number of MacGuffins in play is low, pickpocket wars tend to ensue, wherein two thieves repeatedly steal a MacGuffin back and forth—sometimes only to have it swiped from under them by a third thief or a passing kleptomaniac. If assassins are common enough to leave a lot of bodies lying around, kleptomaniacs and thieves will cheerfully loot the dead. The player's attempts to carry out the mission take on an extra level of challenge when the player has to dodge assassins while pursuing their target, whereas in a population with no assassins the game is much more forgiving.

Since my overall goal was to give the player access to as many different stories as possible, I decided to allow the player to determine the makeup of the population, as well as other key variables such as the number of MacGuffins in the game. However, to allow the player to rerun the same composition several times with greater variability, I did not give them strict control over the number of each type of vehicle; instead, the player sets the probability mass function that is used to choose the type of each vehicle. (See Figure 3.)

Enter numbers to indicate proportion of each type of NPC

3 Thieves

5 Assassins

1 Kleptomaniacs

16 Civilians

25 Total number of NPCs

5 MacGuffins in play

3 Targets per assassin

Player mission

Theft mission

Assassination mission

Click to continue

Figure 3: This opening menu allows the player to set game variables, including the composition of the vehicle population. In this case, each vehicle has a 12% probability of being a thief, a 20% probability of being an assassin, a 4% probability of being a kleptomaniac, and a 64% probability of being a civilian.

Conclusion

The finished CharacterSimulator certainly succeeds at generating different sequences of events every time; in fact, with so many independent NPCs interacting, it is nearly impossible to duplicate a previous game. While this made the system painful to debug at times, it achieves the stated goal of generating unique narratives. A single iteration of the game tends to generate many

story events in an unpredictable way, without the necessity of a designer explicitly laying out every possible permutation.

Moreover, the CharacterSimulator architecture is relatively simple, at least compared to complex drama management systems like the one behind *Façade*, but it can potentially generate increasingly complex narratives with the addition of more types of NPC. Although the game as it is now has only four vehicle types (excluding the player's vehicle), it can accommodate as many as desired. Some other vehicle types that I considered but did not implement were partnered vehicles, which would meet up with their assigned partner and exchange MacGuffins; security guards, which would arrest other vehicles caught committing theft or murder; and stalkers, which would follow another vehicle at a distance but attempt to stay out of its field of vision. A different game in a different narrative genre might suggest other ideas for vehicle types.

The extent to which the game changes when the composition of the NPC population is adjusted is also an asset. Besides nicely illustrating the impact that narrative can have on gameplay, it effectively gives the player several games for the price of one.

The finished product does have its flaws. By focusing on non-player characters, I ended up neglecting the player. I left the implementation of player missions far too late, and it shows; the player missions are far too simple. The player would be better served by more complex missions, especially ones that force him or her to interact with more of the NPCs; for instance, if the player not only had to steal a MacGuffin, but also deliver it to another character afterwards, or if the player had to be careful not to carry out an assassination while in any other character's field of vision. As it is, it's far too easy for the player to simply ignore most of what is happening and focus on the one or two NPCs important to his or her mission. This is only exacerbated by

the high volume of NPC events, many of which are not immediately relevant to the player's mission.

This is related to a more fundamental problem: the stories generated by CharacterSimulator tend towards incoherence. While there may occasionally be chains of cause and effect (Echo has the Maltese Falcon, but Foxtrot steals it, and while chasing after it Echo is killed by Golf), events frequently have little effect on vehicles not directly involved in them. Here, too, a possible solution is to design missions that encourage NPCs to interact with more of their fellows, in the hope that this will lead to more interconnection between events.

Ideas for Future Development

The immediately obvious avenue for further development of this game is adding more vehicle types. It might also be helpful to give the player more options when choosing a mission, as well as a larger arsenal of verbs. For that matter, NPCs would benefit from more verbs as well.

One feature that was cut during the development process was line of sight: making NPCs unable to see through solid objects, allowing other characters to avoid pursuit by hiding behind the furniture. The process of adding furniture and other scenery would also lend itself to adding more possible actions, such as "get a drink" and "poison the punch bowl".

While I don't regret the decision to move away from the Braitenberg model in favor of making vehicles more goal-oriented, one possible refinement to the system would be adding a Braitenberg-like mechanism wherein vehicles have their trajectory altered by reactions to external stimuli. For instance, they could be programmed to avoid security guards or agents

aligned with the enemy, or gather around dead bodies to rubberneck. This would give civilians something to do besides wander aimlessly, and allow them to have more of an effect on the story; what if an assassin has to get away from a dead body quickly before someone sees it and has them arrested for murder?

Finally, although the general CharacterSimulator architecture could easily be adapted to different frame stories, the actual code I wrote is often specifically tailored to the spy-themed game. I would like to edit parts of the code to make it more suitable for reuse, and perhaps make this abstracted version of CharacterSimulator available to other game developers via the Internet.

Works Cited

[1] C. Crawford, "Simple Strategies That Don't Work," in *Chris Crawford on Interactive Storytelling*. Berkeley, CA: New Riders, 2005, ch. 7, pp. 123-134.

[2] *Choice of Broadsides*. [Web-based game]. USA: Choice of Games, 2010. <http://www.choiceofgames.com/broadsides/>.

[3] W. Wright, *The Sims*. [CD-ROM]. USA: Electronic Arts, 2000.

[4] M. Mateas and A. Stern, *Façade, a One-Act Interactive Drama*. [Downloadable PC game]. USA: Procedural Arts, 2005. <http://www.interactivestory.net/>.

[5] M. Mateas and A. Stern, "*Façade*: An Experiment in Building a Fully-Realized Interactive Drama," presented at the Game Developers Conference, San Jose, CA, March 4-8, 2003.

[6] Team MIA, *The Snowfield*. [Web-based game]. USA: Singapore-MIT GAMBIT Game Lab, 2011. <http://gambit.mit.edu/loadgame/snowfield.php>.

[7] M. Weise, "Postmortem: *The Snowfield*," *Game Developer Magazine Annual Game Career Guide*, pp. 44-49, Fall 2012.

[8] A. E. Pratt, *Clue*. [Board game]. USA: Parker Brothers, 1949.

[9] V. Braitenberg, *Vehicles: Experiments in Synthetic Psychology*. Cambridge, MA: MIT Press, 1984.

Appendix A: Attached CD

Attached with each submitted copy of the thesis is a CD-ROM including all source code for the finished game demo and a runnable build. To play the build, open the file Frame.html in the "Final build" folder.

Appendix B: Source Code

```
package controllers
{
	import events.StateChangeEvent;
	import flash.display.Sprite;
	import flash.events.Event;
	import flash.events.KeyboardEvent;
	import flash.ui.Keyboard;

	[SWF(width="1400", height="1050", backgroundColor="#ffffff")]

	/**
	 * Main class which runs application.
	 * Can switch between three states:
	 * CONTROLS (user accesses game settings),
	 * GAME (the actual game),
	 * and END (displayed after game ends).
	 * Each state is represented by a different class of sprite:
	 * UserOptions, VehicleSim, and EndGame, respectively.
	 */
	public class Frame extends Sprite
	{

		// Indicates current state
		private var state:int;
		private const CONTROLS:int = 0;
		private const GAME:int = 1;
		private const END:int = 2;
		private var stateSprite:Sprite;

		// State sprites
		private var controls:UserOptions;
		private var game:VehicleSim;
		private var end:EndGame;
		// This object (global variable to allow other classes to send it
		// events)
		public static var frame:Frame;

		public function Frame()
		{
			super();
			frame = this;

			// Start in CONTROLS state
			controls = new UserOptions();
			addChild(controls);
			state = CONTROLS;
			stateSprite = controls;

			// Event listeners
			addEventListener(StateChangeEvent.START_GAME,startGame);
			addEventListener(StateChangeEvent.PLAYER_DEATH,endGame);
			addEventListener(StateChangeEvent.PLAYER_SUCCESS,endGame);
			addEventListener(StateChangeEvent.NEW_GAME,newGame);
```

```
			stage.addEventListener(KeyboardEvent.KEY_DOWN,cheatCodes);
		}

		// Change to GAME state
		private function startGame(e:StateChangeEvent):void{
			if (state != GAME){
				removeChild(stateSprite);
				// The following line can be edited to run a
				// different subclass of VehicleSim.
				var game:VehicleSim = new SpySim(e.probDist,e.total,
						e.macGuffins,e.numTargets,e.mission);
				stateSprite = game;
				addChild(game);
				state = GAME;
				game.init();
			}
		}

		// Change to END state
		private function endGame(e:StateChangeEvent):void{
			if (state != END){
				removeChild(stateSprite);
				// check reason for game end: did player win or lose?
				var won:Boolean = (e.type ==
							StateChangeEvent.PLAYER_SUCCESS);
				end = new EndGame(won);
				stateSprite = end;
				addChild(end);
				state = END;
			}
		}

		// Change to CONTROLS state
		private function newGame(e:StateChangeEvent):void{
			if (state != CONTROLS){
				removeChild(stateSprite);
				// Don't create a new sprite; the old one has
				// previously entered values saved.
				stateSprite = controls;
				addChild(controls);
				state = CONTROLS;
			}
		}

		// Keyboard shortcuts
		private function cheatCodes(ke:KeyboardEvent):void{
			// Keystroke to return to CONTROLS state: TAB
			if (ke.keyCode == Keyboard.TAB){
				dispatchEvent(new
					StateChangeEvent(StateChangeEvent.NEW_GAME));
			}
		}
	}
}
```

```
package events
{
	import flash.events.Event;

	/**
	 * Events dispatched to Frame to
	 * signal a change of state.
	 */
	public class StateChangeEvent extends Event
	{
		// Event types
		// switch to GAME state
		public static const START_GAME:String = "Start game";
		// switch to END state because of player death
		public static const PLAYER_DEATH:String = "Player death";
		// switch to END state because of player success
		public static const PLAYER_SUCCESS:String = "Player wins";
		// switch to CONTROLS state
		public static const NEW_GAME:String = "New game";

		// These variables are only used by START_GAME events,
		// which need the parameters to set up a new game.
		// This is the PMF used to determine vehicle types
		public var probDist:Vector.<Number>;
		// Total number of vehicles
		public var total:int;
		// Number of MacGuffins
		public var macGuffins:int;
		// Length of AssassinVehicle's hit list
		public var numTargets:int;
		// Mission chosen by player
		public var mission:String;

		public function StateChangeEvent(type:String,
				probs:Vector.<Number>=null,total:int=0,
				macGuffins:int=0,targets:int=0,mission:String=null)
		{
			super(type, false, false);
			probDist = probs;
			this.total = total;
			this.macGuffins = macGuffins;
			numTargets = targets;
			this.mission = mission;
		}
	}
}
```

```
package controllers
{
	import events.StateChangeEvent;

	import flash.display.Shape;
	import flash.display.SimpleButton;
	import flash.display.Sprite;
	import flash.events.KeyboardEvent;
	import flash.events.MouseEvent;
	import flash.text.TextField;
	import flash.text.TextFieldType;

	import ui.Button;
	import ui.InputBox;
	import ui.RadioButtons;

	import vehicles.PlayerCharacter;

	/**
	 * State allowing user control of game settings.
	 */
	public class UserOptions extends Sprite
	{
		// vector that normalizes to PMF of vehicle types
		private var inputs:Array = new Array();
		// displays invalid input warnings
		private var errorMessage:TextField = new TextField();
		// radio buttons to select player mission
		private var missionChoice:RadioButtons;
		// input box for number of MacGuffins in play
		private var macguffins:InputBox;
		// input box for number of targets per AssassinVehicle
		private var targets:InputBox;
		// input box for total number of vehicles
		private var totalVehicles:InputBox;

		public function UserOptions()
		{
			super();
			quickAdd("Enter numbers to indicate proportion of each type
of NPC",20,50);

			// Input boxes to set probability distribution of vehicles
			var types:Vector.<String> = new
<String>["Thieves","Assassins","Kleptomaniacs","Civilians"];
			var defaults:Vector.<Number> = new <Number>[3,5,1,16];

			for (var i:int = 0; i < 4; i++){
				var inputField:InputBox = new
					InputBox(defaults[i],types[i],100,50*i+100);
				inputs[i] = inputField;
				addChild(inputField);
			}

			// Button to start game (switch to GAME state)
```

```
			var button:Button = new Button("Click to
continue",submit,400,500);
			addChild(button);

			// Input boxes to set other variables
			macguffins = new InputBox(5,"MacGuffins in play",450,100);
			addChild(macguffins);
			targets = new InputBox(3,"Targets per assassin",450,150);
			addChild(targets);
			totalVehicles = new InputBox(25,"Total number of
NPCs",100,325);
			addChild(totalVehicles);

			// Radio buttons to choose PC mission
			missionChoice = new RadioButtons("Player
mission",[PlayerCharacter.THEFT_MISSION,PlayerCharacter.ASSASSIN_MISSION],450
,250);
			addChild(missionChoice);

			// Error printout to inform user of invalid values
			errorMessage.x = 150;
			errorMessage.y = 400;
			errorMessage.width = 600;
			errorMessage.textColor = 0xff0000;
			addChild(errorMessage);
		}

		// Add text to screen
		private function quickAdd(text:String,
						xCoor:Number,yCoor:Number):TextField{
			var newField:TextField = new TextField();
			newField.text = text;
			newField.width = newField.textWidth*text.length;
			newField.x = xCoor;
			newField.y = yCoor;
			addChild(newField);
			return newField;
		}

		private function submit(me:MouseEvent):void{
			// Check to make sure all inputs are valid first
			var values:Vector.<Number> = new Vector.<Number>();
			for (var i:int = 0; i < inputs.length; i++){
				var num:Number = inputs[i].inputValue;
				// All inputs in ratio of vehicle types must be
				// nonnegative numbers
				if (isNaN(num) || num < 0){
					errorMessage.text = "One or more invalid
entries in the probability distribution. Please enter a nonnegative number in
each box.";
					return;
				}
				else
					values[i] = num;
			}
```

```
                    // At least one number in ratio must be > 0 (or there
wouldn't be any NPCs)
                    var sum:Number = 0;
                    for each (var val:Number in values)
                          sum += val;
                    if (sum <= 0){
                          errorMessage.text = "At least one value in the
probability distribution must be greater than zero.";
                          return;
                    }
                    else{
                          // normalize
                          for (var j:int = 0; j < values.length; j++)
                                values[j] /= sum;
                    }
                    // Set other variables
                    var total:int = totalVehicles.inputValue;
                    var mcgTotal:int = macguffins.inputValue;
                    var targetTotal:int = targets.inputValue;
                    var pcMission:String = missionChoice.selected;
                    // Must have > 0 NPCs
                    if (isNaN(total) || total <= 0)
                          errorMessage.text = "Must have at least one NPC";
                    // Number of MacGuffins must be nonnegative
                    else if (isNaN(mcgTotal) || mcgTotal < 0)
                          errorMessage.text = "Invalid number of MacGuffins";
                    // If probability of MacguffinSeekVehicle > 0, must have at
                    // least one MacGuffin
                    else if (mcgTotal == 0 && (values[0] > 0 || pcMission ==
                                            PlayerCharacter.THEFT_MISSION))
                          errorMessage.text = "Must have at least one MacGuffin
if thieves are in play";
                    // Assassins' hit list can't have more vehicles on it than
                    // there are in game
                    else if (values[1] > 0 && (isNaN(targetTotal) ||
                          targetTotal < 0 || targetTotal > total))
                          errorMessage.text = "Invalid number of assassin
targets. Must be a nonnegative number no greater than the total number of
agents.";
                    // If everything checks out, start game (switch to state
GAME)
                    else
                          parent.dispatchEvent(new StateChangeEvent(
                                StateChangeEvent.START_GAME,values,total,
                                mcgTotal,targetTotal,pcMission));
              }
        }
}
```

```
package ui
{
	import events.UIEvent;
	import flash.display.Shape;
	import flash.display.Sprite;
	import flash.text.TextField;

	/**
	 * UI class to allow user to select one option from a list
	 */
	public class RadioButtons extends Sprite
	{
		private var buttons:Vector.<OptionButton> = new
				Vector.<OptionButton>();	// set of buttons
		private var selection:Sprite;		// selection indicator dot
		private var _selected:int = 0;		// index of option selected

		public function RadioButtons(label:String, choices:Array,
						xcor:Number, ycor:Number)
		{
			super();
			x = xcor;
			y = ycor;
			// caption
			var labelField:TextField = new TextField();
			labelField.text = label;
			addChild(labelField);

			// create a button for each option
			var idCounter:int = 0;
			var offsetY:Number = 40;
			for each (var option:String in choices){
				var nextButton:OptionButton = new OptionButton(
							20,offsetY,idCounter,option);
				buttons.push(nextButton);
				addChild(nextButton);
				offsetY += 20;
				idCounter++;
			}

			addEventListener(UIEvent.SELECT,select);
			dispatchEvent(new UIEvent(UIEvent.SELECT));
		}

		// new option selected
		private function select(uie:UIEvent):void{
			_selected = uie.id;
			// move selection indicator dot
			if (selection)
				removeChild(selection);
			selection = new Sprite();
			selection.x = 20;
			selection.y = 40+uie.id*20;
			selection.graphics.beginFill(0x0);
			selection.graphics.drawCircle(0,0,5);
			addChild(selection);
```

```
        }

        // return selected option
        public function get selected():String{
            return buttons[_selected].option;
        }
    }
}
```

```
package ui
{
	import events.UIEvent;
	import flash.display.Sprite;
	import flash.events.MouseEvent;
	import flash.text.TextField;

	/**
	 * Used as part of a RadioButtons object.
	 * Could also be used as part of a UI object allowing
	 * multiple selections.
	 */
	public class OptionButton extends Sprite
	{
		// ID number identifies each option in a set of radio buttons
		private var idNum:int;
		// option represented by this button
		private var _option:String;

		public function OptionButton(xcor:Number,ycor:Number,
										id:int,labelText:String)
		{
			super();
			x = xcor;
			y = ycor;
			idNum = id;

			// selection button
			graphics.beginFill(0xcccccc);
			graphics.drawCircle(0,0,10);

			// label
			var label:TextField = new TextField();
			label.text = labelText;
			label.x = 20;
			label.y = -10;
			label.height = label.textHeight * 1.5;
			label.width = 120;
			addChild(label);
			_option = labelText;

			// clicking selects this object
			addEventListener(MouseEvent.CLICK, select);
		}

		// Indicate that this button has been selected
		private function select(me:MouseEvent):void{
			parent.dispatchEvent(new UIEvent(UIEvent.SELECT,idNum));
		}

		public function get option():String{
			return _option;
		}
	}
}
```

```
package events
{
	import flash.events.Event;

	/**
	 * Events related to UI objects.
	 * Currently only used for radio button selection.
	 */
	public class UIEvent extends Event
	{
		public static const SELECT:String = "Select option";
		// used for selection of options in RadioButton

		public var id:int;		// indicates button selected

		public function UIEvent(type:String,idNum:int=0)
		{
			super(type);
			id = idNum;
		}
	}
}
```

```
package ui
{
	import flash.display.Sprite;
	import flash.text.TextField;
	import flash.text.TextFieldType;

	/**
	 * Editable text box with a label.
	 */
	public class InputBox extends Sprite
	{
		private var input:TextField;

		public function InputBox(defaultValue:Number,text:String,
						xcor:Number,ycor:Number)
		{
			super();
			x = xcor;
			y = ycor;
			// create text box
			input = new TextField();
			input.width = 30;
			input.height = 20;
			input.type = TextFieldType.INPUT;
			input.background = true;
			input.backgroundColor = 0xcccccc;
			// set default value displayed in box
			input.text = defaultValue.toString();
			addChild(input);

			// label
			var label:TextField = new TextField();
			label.text = text;
			label.width = label.textWidth*text.length;
			label.height = label.textHeight*1.5;
			label.x = 35;
			addChild(label);
		}

		// returns last value written in box
		public function get inputValue():Number{
			return parseFloat(input.text);
		}
	}
}
```

```
package ui
{
	import flash.display.Sprite;
	import flash.events.MouseEvent;
	import flash.text.TextField;

	/**
	 * Clickable button class for UI
	 */
	public class Button extends Sprite
	{
		// function called when user presses button
		private var callOnClick:Function;

		public function Button(labelText:String,onClick:Function,
				       xcor:Number,ycor:Number)
		{
			super();
			// draw button
			graphics.beginFill(0xcccccc);
			graphics.drawRoundRect(0,0,100,30,15);
			// label
			var label:TextField = new TextField();
			label.text = labelText;
			label.x = 8;
			label.y = 5;
			addChild(label);

			x = xcor;
			y = ycor;
			// set function to call when clicked
			callOnClick = onClick;
			addEventListener(MouseEvent.CLICK,clicked);
		}

		// When button is clicked, calls a previously specified function
		private function clicked(me:MouseEvent):void{
			callOnClick(me);
		}
	}
}
```

```
package controllers
{
	import events.PlayerMessage;
	import events.StoryEvent;
	import flash.geom.Point;
	import vehicles.*;

	/**
	 * Sprite that runs the game. Extends VehicleSim;
	 * levels with different layouts can be created
	 * by running different VehicleSim subclasses.
	 */
	public class SpySim extends VehicleSim
	{
		public function SpySim(probDist:Vector.<Number>,totalBots:int,
		macGuffins:int,assassinTargets:int,mission:String)
		{
			// vehicle names
			nameBank = ["Alpha", "Bravo", "Charlie", "Delta", "Echo",
				"Foxtrot", "Golf", "Hotel", "India", "Juliet",
				"Kilo", "Lima", "Mike", "November", "Oscar", "Papa",
				"Quebec", "Romeo", "Sierra", "Tango", "Uniform",
				"Victor", "Whiskey", "X-ray", "Yankee", "Zulu"];

			// Indicates which events should be displayed on newsfeed
			var events:Vector.<String> = new <String>
				[StoryEvent.ENTRY,StoryEvent.EXIT,
				StoryEvent.ASSASSIN_FAILURE,StoryEvent.DEATH,
				StoryEvent.FOUND_BODY, StoryEvent.THEFT_FAILURE,
				StoryEvent.ITEM_THEFT,StoryEvent.NULL_THEFT];
			// Indicates which events should be displayed on player-
			// specific newsfeed
			var messages:Vector.<String> = new <String>
					[PlayerMessage.NAME,PlayerMessage.TARGETED,
					PlayerMessage.INVALID_ACTION,
					PlayerMessage.ASSASSIN_MISSION,
					PlayerMessage.THEFT_MISSION];

			// set up fixed obstacles
			walls = [new Obstacle(450,50,0,-35),
				new Obstacle(450,50,550,-35),
				new Obstacle(1000,50,0,1000),
				new Obstacle(50,1000,-35,0),
				new Obstacle(50,1000,1000,0),
				new Obstacle(100,30,450,0,true) // door
			];
			// location vehicles aim for after mission completion
			exitLocation = new Point(500,15);

			super(new <Class>[MacguffinSeekVehicle,AssassinVehicle,
				KleptoVehicle,WanderVehicle], probDist, totalBots,
			macGuffins, assassinTargets, events, messages, mission);
		}
	}
}
```

```
package controllers
{
	import events.PlayerMessage;
	import events.StoryEvent;

	import flash.display.BitmapData;
	import flash.display.Shape;
	import flash.display.Sprite;
	import flash.events.Event;
	import flash.events.KeyboardEvent;
	import flash.geom.Point;
	import flash.text.TextField;

	import ui.NewsDisplay;

	import vehicles.*;

	/**
	 * Base game class. Meant to be extended by
	 * subclasses such as SpySim to set up different
	 * game layouts. Runs game and stores master lists of
	 * objects in game.
	 */
	public class VehicleSim extends Sprite
	{
		public static var world:VehicleSim;
		// the VehicleSim object running the game; designed to be
		// accessible to other classes

		protected var vehicleList:Vector.<Vehicle> = new
					Vector.<Vehicle>();	// all Vehicles in game
		protected var vehicleTypes:Vector.<Class>;
						// list of vehicle types in game
		protected var vehicleDistribution:Vector.<Number>;
							// PMF of vehicle types
		protected var sourceList:Vector.<Source> = new Vector.<Source>();
					// all Sources in game, including Vehicles
		public var obstacles:Vector.<Obstacle> = new Vector.<Obstacle>();
							// all Obstacles in gamespace
		public var walls:Array;
					// all walls in game
		protected var macGuffins:Vector.<MacGuffin> = new
				Vector.<MacGuffin>();	// all MacGuffins in game

		public var pc:PlayerCharacter;
					// player-controlled Vehicle

		public var dimensions:Point = new Point(1000,1000);
							// size of gamespace
		public var exitLocation:Point;
					// coordinates of exit
		protected static var textDisplay:NewsDisplay;
						// newsfeed
		protected static var playerTextDisplay:NewsDisplay;
						// player newsfeed
```

```
// Used to assign unique names to vehicles
protected var nameBank:Array;
private var namesUsed:Array = new Array();

private var nameCount:int = 0;

private var numVehicles:int;        // total vehicles in game
private var numMacGuffins:int;      // total MacGuffins in game
public var assassinTargets:int;
                        // number of targets per AssassinVehicle
private var pcMission:String;       // player's mission type

public function VehicleSim(vTypes:Vector.<Class>,
            vDist:Vector.<Number>,vehicles:int,macGuffins:int,
            targets:int,eventTypes:Vector.<String>,
            messages:Vector.<String>,mission:String)
{
      super();
      world = this;

      addEventListener(Event.ENTER_FRAME,onFrameEnter);

      // listen for newsfeed events
      for each (var event:String in eventTypes){
            addEventListener(event,print);
      }
      for each (var mType:String in messages){
            addEventListener(mType,playerMessage);
      }

      vehicleTypes = vTypes;
      vehicleDistribution = vDist;
      numVehicles = vehicles;
      numMacGuffins = macGuffins;
      assassinTargets = targets;
      pcMission = mission;
}

// Game setup takes place here
public function init():void{
      // add walls to space
      for each (var wall:Obstacle in walls){
            addChild(wall);
            sourceList.push(wall);
            obstacles.push(wall);
      }

      // build vehicles
      // vDist is the PMF and vDistCumulative the CDF for the
      // random variable defining vehicle type
      var vDistCumulative:Array = new Array();
      var sum:Number = 0;
      for (var pInd:int = 0; pInd < vehicleDistribution.length;
pInd++){
            sum += vehicleDistribution[pInd];
            vDistCumulative.push(sum);
```

```
		}
		for (var i:int = 0; i < numVehicles; i++){
		// randomly choose vehicle type according to distribution
			var r:Number = Math.random();
			var t:int = 0;
			while (r > vDistCumulative[t]){
				t++;
			}
			var v:Vehicle = new vehicleTypes[t]();
			addChild(v);
			vehicleList.push(v);
			sourceList.push(v);
		}

		// generate MacGuffins
		for (var n:int = 0; n < numMacGuffins; n++){
			var mcg:MacGuffin = new MacGuffin();
			var rv:Vehicle = getRandomVehicle();
			rv.addMacguffin(mcg);
			macGuffins.push(mcg);
		}

		// create player character
		pc = new PlayerCharacter(pcMission);
		addChild(pc);
		sourceList.push(pc);
		vehicleList.push(pc);

		// text display for news updates
		textDisplay = new NewsDisplay(1060,290,1500,800,50);
		addChild(textDisplay);
		playerTextDisplay = new NewsDisplay(1060,30,1500,300,15);
		addChild(playerTextDisplay);

		dispatchEvent(new PlayerMessage(PlayerMessage.NAME,pc));

		// set mission targets
		for (var i3:int = 0; i3 < vehicleList.length; i3++){
			var ve:Vehicle = vehicleList[i3];
			ve.missionSetup();
		}
	}

	// update every frame
	public function onFrameEnter(e:Event):void{
		update();
	}

	// make sure all vehicles update
	public function update():void{
		for each (var ve:Vehicle in vehicleList)
		{
			ve.sense(sourceList);
			ve.update();
		}
	}
```

```
// remove vehicle from game
public function remove(v:Vehicle):void{
      var index:int = vehicleList.indexOf(v);
      if (index >= 0){
            vehicleList.splice(index,1);
      }
}

// print to newsfeed
protected static function print(se:StoryEvent):void{
      textDisplay.addLine(se.toString());
      // if event concerns player, print to player newsfeed
      if (se.playerEvent)
            playerTextDisplay.addLine(se.toString());
}

// print to player newsfeed
protected static function playerMessage(pm:PlayerMessage):void{
      playerTextDisplay.addLine(pm.toString());
}

// returns a random vehicle, of a certain type
// if type is specified
public function getRandomVehicle(type:Class = null):Vehicle{
      var vehiclesOfType:Vector.<Vehicle>;
// if vehicle type specified, list all vehicles of that type
// This feature isn't currently in use, though;
// none of the actual calls to getRandomVehicle specify a type.
      if (type){
            /* Could check vehicleTypes list to see if this type
               is on it, but decided against it, because this
               should still work with inherited superclasses.
               (E.g., if AVehicle inherits from BVehicle and
               vehicleTypes includes AVehicle, calling
               getRandomVehicle(BVehicle) shouldn't return null.)
            */
            vehiclesOfType = new Vector.<Vehicle>();
            for each (var ve:Vehicle in vehicleList)
            {
                  if (ve is type){
                        vehiclesOfType.push(ve);
                  }
            }
      }
      else
            vehiclesOfType = vehicleList;
      // return random vehicle
      if (vehiclesOfType.length > 0){
            var randomIndex:int = Math.floor(
                        Math.random()*vehiclesOfType.length);
            return vehiclesOfType[randomIndex];
      }
      // if none found, return null
      else
            return null;
```

```
        }

        // returns random MacGuffin
        public function getRandomMacGuffin():MacGuffin{
              if (macGuffins.length > 0){
                    var randomIndex:int = Math.floor(
                                      Math.random()*macGuffins.length);
                    return macGuffins[randomIndex];
              }
              return null;
        }

        // generates unique vehicle names by shuffling name bank
        // and adding a number whenever a name is repeated
        public function getName():String{
              var randomIndex:int = Math.floor(
                                      Math.random()*nameBank.length);
              var name:String = nameBank[randomIndex];
              nameBank.splice(randomIndex,1);
              namesUsed.push(name);
              if (nameCount > 0)
                    name += nameCount;
              if (nameBank.length == 0)
              {
                    nameCount++;
                    nameBank = namesUsed;
                    namesUsed = new Array();
              }
              return name;
        }
    }
}
```

```
package
{
      import controllers.VehicleSim;
      import flash.display.Sprite;
      import flash.geom.Point;
      import flash.utils.getQualifiedClassName;

      /**
       * Objects in gamespace that are detectable by Sensors.
       * Superclass of Vehicle.
       */
      public class Source extends Sprite
      {
            protected var id:int;                          // unique ID number
            private static var idCounter:int = 0;
                                                     // ID generation counter
            public var wSize:int;                                    // width
            public var hSize:int;                                    // height

            public function Source()
            {
                  super();
                  id = idCounter;
                  idCounter++;
                  wSize = 35;
                  hSize = 35;
            }

            public function get location():Point{
                  return new Point(x,y);
            }

            public override function toString():String{
                  return flash.utils.getQualifiedClassName(this) + " " + id;
            }

            // check for collisions using separating axis test
            protected function collidesWith(o:Source):Boolean{
                  // vertices of this Source
                  var vVertices:Array = [ new Point(x,y),
                        new Point(x + wSize * Math.cos(rotation*
Math.PI/180), y + wSize*Math.sin(rotation* Math.PI/180)),
                        new Point(x + wSize * Math.cos(rotation* Math.PI/180)
- hSize * Math.sin(rotation* Math.PI/180),y+wSize*Math.sin(rotation*
Math.PI/180) + hSize * Math.cos(rotation* Math.PI/180)),
                        new Point(x - hSize * Math.sin(rotation*
Math.PI/180), y + hSize * Math.cos(rotation* Math.PI/180))];
                  // vertices of other Source
                  var oVertices:Array = [ new Point(o.x,o.y),
                        new Point(o.x + o.width * Math.cos(o.rotation*
                                                      Math.PI/180),
                         o.y + o.width*Math.sin(o.rotation* Math.PI/180)),
                        new Point(o.x + o.width * Math.cos(o.rotation*
                        Math.PI/180) - o.height * Math.sin(o.rotation*
                        Math.PI/180),o.y+o.width*Math.sin(o.rotation*
```

```
                Math.PI/180) + o.height * Math.cos(o.rotation*
                Math.PI/180)),
                new Point(o.x - o.height * Math.sin(o.rotation*
                      Math.PI/180), o.y + o.height *
                      Math.cos(o.rotation* Math.PI/180))];
          var p1:Point;
          var p2:Point;
          // check each side of each shape for separating axis
          for (var i:int = 0; i < vVertices.length; i++){
                p1 = vVertices[i];
                if (i > 0)
                      p2 = vVertices[i-1];
                else
                      p2 = vVertices[vVertices.length-1];
                if (separatingEdge(p1,p2,vVertices,oVertices))
                      return false;
          }
          for (var j:int = 0; j < oVertices.length; j++){
                p1 = oVertices[j];
                if (j > 0)
                      p2 = oVertices[j-1];
                else
                      p2 = oVertices[oVertices.length-1];
                if (separatingEdge(p1,p2,vVertices,oVertices))
                      return false;
          }
          return true;
    }

    // check for collision against all obstacles
    protected function hasCollision():Obstacle{
          for each (var o:Obstacle in VehicleSim.world.obstacles){
                if (collidesWith(o)){
                      return o;
                }
          }
          return null;
    }

    // check if line between these two points is a separating axis
    private static function separatingEdge(edgePoint1:Point,
          edgePoint2:Point, verticesA:Array,verticesB:Array):Boolean{
          var aValues:Array = new Array();
          var bValues:Array = new Array();
          // get side of edge for each point in these two sets
          for each (var pa:Point in verticesA){
                aValues.push(getSideOfEdge(
                                  edgePoint1,edgePoint2,pa));
          }
          for each (var pb:Point in verticesB){
                bValues.push(getSideOfEdge(
                                        edgePoint1,edgePoint2,pb));
          }
          // points in one set should all be on the same side and
          // points from the other on the opposite side
          if (aValues.every(nonPositive) &&
```

```
                    bValues.every(nonNegative) ||
                    bValues.every(nonPositive) &&
                    aValues.every(nonNegative) )
                    return true;
              else
                    return false;
        }

        // Returns positive for points on one side of the edge, negative
        // for points on the other
        private static function getSideOfEdge(edgePoint1:Point,
                          edgePoint2:Point, thirdPoint:Point):Number{
              var edgeVector:Point = edgePoint1.subtract(edgePoint2);
              var rotatedVector:Point = new Point(-
edgeVector.y,edgeVector.x);
              return rotatedVector.x * (thirdPoint.x - edgePoint1.x) +
rotatedVector.y * (thirdPoint.y - edgePoint1.y);
        }

        private static function
nonPositive(element:Number,index:int,arr:Array):Boolean{
              return element <= 0;
        }

        private static function
nonNegative(element:Number,index:int,arr:Array):Boolean{
              return element >= 0;
        }

        // center point (assuming Source is a rectangle)
        public function center():Point{
              return new Point(x + wSize*.5
                              *Math.cos(rotation*Math.PI/180) -
                          hSize*.5*Math.sin(rotation*Math.PI/180),
                          y + wSize*.5*Math.sin(rotation*Math.PI/180) +
                          hSize*.5*Math.cos(rotation*Math.PI/180));
        }

        // place in a random spot that doesn't collide with any obstacles
        protected function placeRandomly():void{
              x = Math.random()*VehicleSim.world.dimensions.x;
              y = Math.random()*VehicleSim.world.dimensions.y;
              if (hasCollision()){
                    placeRandomly();
              }
        }

        // rotate Source about center
        protected function rotateThetaDegrees(theta:Number):void{
              var finalAngleRadians:Number = (rotation + theta)
                                                  * Math.PI / 180;
              x = center().x - .5*wSize*Math.cos(finalAngleRadians) +
                              .5*hSize*Math.sin(finalAngleRadians);
              y = center().y - .5*wSize*Math.sin(finalAngleRadians) -
              .5*hSize*Math.cos(finalAngleRadians);
              rotation += theta;
```

```
            }
        }
    }
```

```
package vehicles
{
	import controllers.VehicleSim;
	import events.StoryEvent;
	import events.VisionEvent;
	import flash.display.DisplayObject;
	import flash.display.Shape;
	import flash.display.Sprite;
	import flash.geom.Point;
	import flash.text.TextField;

	/**
	 * Base class for all Vehicles, which are used to
	 * represent both player and non-player characters.
	 * Intended as an abstract class;
	 * all instances of Vehicle will belong to
	 * some subclass and inherit Vehicle's functions.
	 */
	public class Vehicle extends Source
	{
		protected var color:uint;			// vehicle color
		protected var numTargets:int = 0;
						// number of targets being pursued
		protected var targets:Array;			// array of targets

		protected var leftSpeed:Number = 0;		// speed of left wheel
		protected var rightSpeed:Number = 0;		// speed of right wheel
		protected var leftSensor:Sensor;		// left sensor
		protected var rightSensor:Sensor;		// right sensor
		protected var sourcesInSight:Array = new Array();
						// all Sources currently visible
		protected var speedLimit:Number = 50;		// maximum wheel speed
		protected var inventory:Array = new Array();
					// all MacGuffins held by this Vehicle
		public var dead:Boolean = false;		// is character dead?
		private var label:TextField;			// character nametag
		private var characterName:String;		// character's name

		// Keeps track of steps for the wander() function
		protected var stepCounter:int = 0;
		protected var numSteps:int = 50;

		// Skill checks: probability of success at certain actions
		protected var pickpocketSkillCheck:Number = .8;
		protected var assassinSkillCheck:Number = .8;

		private var _missionComplete:Boolean = false;
							// is mission complete?
		protected var zoneOfControl:int = 30;
						// "arm's reach" for this character

		public function Vehicle()
		{
			super();
			// draw
			graphics.beginFill(color);
```

```
        graphics.lineStyle(1);
        graphics.drawRect(0, 0, wSize, hSize);
        graphics.endFill();
        placeRandomly();
        wSize = 35;
        hSize = 35;
        rotation = Math.random()*360;
        graphics.drawCircle(wSize/2,0,3);
        characterName = VehicleSim.world.getName();
        label = new TextField();
        label.text = characterName;
        addChild(label);
        // set up sensors
        leftSensor = new Sensor(new Point(x,y),
                                rotation* Math.PI/180,true);
        rightSensor = new Sensor(new Point(
                    x+wSize*Math.cos(rotation* Math.PI/180),
                    y+wSize*Math.sin(rotation* Math.PI/180)),
                    rotation* Math.PI/180,false);
}

// debugging tool: outlines Vehicle's visual field
public function drawVisionBounds():void{
        var leftPoints:Array = leftSensor.visionBoundPoints();
        var rightPoints:Array = rightSensor.visionBoundPoints();

        graphics.moveTo(0,0);
        graphics.lineTo(leftPoints[0].x,leftPoints[0].y);

        graphics.moveTo(wSize,0);
        graphics.lineTo(wSize+rightPoints[1].x,rightPoints[1].y);

        graphics.beginFill(0xffffcc,.1);
        graphics.drawCircle(0,0,leftSensor.sightRadius);
        graphics.drawCircle(wSize,0,rightSensor.sightRadius);
}

// By default, vehicle stands still.
// Mostly overridden by subclasses.
public function updateSpeed():void{
        leftSpeed = 0;
        rightSpeed = 0;
}

// Check what's visible
public function sense(sources:Vector.<Source>):void{
        // iterate through list of sources
        for (var s:int = 0; s < sources.length; s++)
        {
                // next source on list
                var nextSource:Source = sources[s];
                if (nextSource != this)
                {
                        // check if source was already visible
                        var ind:int = sourcesInSight.indexOf
                                                (nextSource);
```

```
                        var wasVisible:Boolean = ind >= 0;
                        if (sourceInSight(nextSource))
                        {
                              if (!wasVisible)
                              {
                                    sourcesInSight.push(nextSource);
                                    VehicleSim.world.dispatchEvent(new
VisionEvent(VisionEvent.WAS_SEEN,this,nextSource));
                                    VehicleSim.world.dispatchEvent(new
VisionEvent(VisionEvent.SAW,this,nextSource));
                              }
                        }
                        else{
                              if (wasVisible){
                                    sourcesInSight.splice(ind,1);
                                    VehicleSim.world.dispatchEvent(new
VisionEvent(VisionEvent.WAS_LOST,this,nextSource));
                                    VehicleSim.world.dispatchEvent(new
VisionEvent(VisionEvent.LOST,this,nextSource));
                              }
                        }
                  }
            }
            leftSensor.senseVehicles(sourcesInSight);
            rightSensor.senseVehicles(sourcesInSight);
      }

      // is source in my field of vision?
      public function sourceInSight(source:Source):Boolean{
            var leftEye:Boolean = leftSensor.sourceInSight(source);
            var rightEye:Boolean = rightSensor.sourceInSight(source);
            return leftEye || rightEye;
      }

      // updates position every frame
      public function update():void{
            if (dead){
                  leftSpeed = 0;
                  rightSpeed = 0;
            }
            else{
                  updateSpeed();

                  // update speed, position, and orientation
                  leftSpeed = Math.min(leftSpeed,speedLimit);
                  rightSpeed = Math.min(rightSpeed,speedLimit);

                  x += (leftSpeed + rightSpeed) * .5 *
Math.sin(rotation* Math.PI/180);
                  y -= (leftSpeed + rightSpeed) * .5 *
Math.cos(rotation* Math.PI/180);
                  rotation += Math.atan((leftSpeed - rightSpeed) /
wSize) * 180 / Math.PI;

                  var obs:Obstacle = hasCollision();
                  if (obs){
```

```
                        // if this is a door and bot wants to leave
                        if (obs.isDoor && missionComplete){
                              exit();
                        }
                        // collision
                        else {
                              // back up
                              x -= (leftSpeed + rightSpeed) * .5 *
Math.sin(rotation* Math.PI/180);
                              y += (leftSpeed + rightSpeed) * .5 *
Math.cos(rotation* Math.PI/180);
                              rotation -= Math.atan((leftSpeed -
rightSpeed) / wSize) * 180 / Math.PI;

                              if (!(this is PlayerCharacter)){
                              // make sure vehicle doesn't hit a wall

                                    if (x < 50)
                                          x = 55;
                                    if (y < 50)
                                          y = 55;
                                    if (x > 950)
                                          x = 950;
                                    if (y > 950)
                                          y = 950;

                                    // turn around
                                    x += wSize *
Math.cos(rotation*Math.PI/180) - hSize * Math.sin(rotation*Math.PI/180);
                                    y += wSize *
Math.sin(rotation*Math.PI/180) + hSize * Math.cos(rotation*Math.PI/180);
                                    rotation += 180;
                              }
                        }
                  }
                  // keep labels upright
                  label.rotation = -rotation;
                  for each (var mcg:MacGuffin in inventory){
                        mcg.label.rotation = -rotation;
                  }
                  // update sensor coordinates
                  leftSensor.location = new Point(x,y);
                  rightSensor.location = new Point(
                              x+wSize*Math.cos(rotation* Math.PI/180),
                              y+wSize*Math.sin(rotation* Math.PI/180));
                  leftSensor.direction = rotation* Math.PI/180;
                  rightSensor.direction = rotation* Math.PI/180;
            }

      }

      // is mission complete?
      protected function get missionComplete():Boolean{
            return _missionComplete;
      }
```

```
		protected function set missionComplete(isComplete:Boolean):void{
			_missionComplete = isComplete;
		}

		/**
		 * Dummy function overridden by subclasses.
		 */
		public function missionSetup():void{

		}

		// wander around
		protected function wander():void{
			if (stepCounter > 0){
				stepCounter--;
				leftSpeed = 5;
				rightSpeed = 5;
			}
			// every few seconds, randomly change direction
			else{
				stepCounter = numSteps;
				leftSpeed = 5 + 15 * Math.random();
				rightSpeed = 5 + 15 * Math.random();
			}
		}

		// aim for Source s
		protected function aimForSource(s:Source):void{
			aimForPoint(s.center());
		}

		// steer vehicle towards point p
		protected function aimForPoint(p:Point):void{
			/* Not a mistake! The x and y parameters are switched on
			   purpose, because Sprites measure rotations clockwise.
			   The order of the coordinates is switched on purpose,
			   too. This was just that way that made it all come out
			   positive.
			*/
			var targetAngle:Number = Math.atan2(p.x - center().x,
center().y - p.y) * 180 / Math.PI;
			var turnAngle:Number = targetAngle - rotation;
			var baseSpeed:Number = 8;
			var turnSpeed:Number = .08;
			leftSpeed = baseSpeed + turnAngle*turnSpeed;
			rightSpeed = baseSpeed - turnAngle*turnSpeed;
		}

		// returns true if within "zone of control" of this source,
		// i.e. if it would be possible for this character to reach out
		// and touch Source s
		protected function inZOC(s:Source):Boolean{
			var centerS:Point = s.center();
			var myCenter:Point = center();
			var distSq:Number = (myCenter.x - centerS.x)*(myCenter.x -
centerS.x) + (myCenter.y - centerS.y)*(myCenter.y - centerS.y);
```

```
		return distSq < zoneOfControl*zoneOfControl;
	}

	// does player have this MacGuffin? If mcg is not specified,
	// does player have any MacGuffin?
	public function hasMacguffin(mcg:MacGuffin=null):Boolean{
		if (mcg)
			return inventory.indexOf(mcg) >= 0;
		return inventory.length > 0;
	}

	// This is just to handle the graphics, to make sure all
	// MacGuffins are separately visible
	private function nextSlotOpen():int{
		var slots:Array = new Array();
		for each (var mcg:MacGuffin in inventory){
			slots[mcg.slot] = true;
		}
		for (var i:int = 0; i < slots.length; i++){
			if (!slots[i])
				return i;
		}
		return slots.length;
	}

	// add item to inventory
	public function addMacguffin(mcg:MacGuffin):void{
		if (!hasMacguffin(mcg)){
			inventory.push(mcg);
			mcg.setParent(this,nextSlotOpen());
		}
		else
			throw new Error("Tried to add an object to inventory
twice");
	}

	// do I see the vehicle holding this MacGuffin?
	public function seesMacguffinVehicle(mcg:MacGuffin):Vehicle{
		for each (var s:Source in sourcesInSight){
			if (s is Vehicle){
				var v:Vehicle = Vehicle(s);
				if (v.hasMacguffin(mcg)){
					return v;
				}
			}
		}
		return null;
	}

	// Adversary vehicle either steals a specific object
	// or, if mcg is null, cleans out inventory
	public function hasPocketPicked(mcg:MacGuffin=null):Array{
		if (mcg){
			if (hasMacguffin(mcg)){
				removeChild(mcg);
				return inventory.splice(
```

```
                        inventory.indexOf(mcg),1);
                }
                else
                    return new Array();
            }
            else{
                for each (var child:MacGuffin in inventory)
                    removeChild(child);
                return inventory.splice(0);
            }
        }

        // Pick someone else's pocket
        protected function
pickPocket(v:Vehicle,mcg:MacGuffin=null):Boolean{
            // do skill check
            if (Math.random() < pickpocketSkillCheck){
                // success!
                var loot:Array = v.hasPocketPicked(mcg);
                if (loot.length == 0){
                    VehicleSim.world.dispatchEvent(new
StoryEvent(StoryEvent.NULL_THEFT,this,v));
                }
                for each (var item:MacGuffin in loot){
                    addMacguffin(item);
                    VehicleSim.world.dispatchEvent(new
StoryEvent(StoryEvent.ITEM_THEFT,this,v,item));
                }
                return true;
            }
            else{
                // skill check fails; can't steal anything
                if (mcg)
                    VehicleSim.world.dispatchEvent(new
StoryEvent(StoryEvent.THEFT_FAILURE,this,v,mcg));
                else
                    VehicleSim.world.dispatchEvent(new
StoryEvent(StoryEvent.THEFT_FAILURE,this,v));
                return false;
            }
        }

        // leave the room (called once Vehicle is at the door)
        protected function exit():void{
            VehicleSim.world.dispatchEvent(new
StoryEvent(StoryEvent.EXIT,this));
            visible = false;
        // remove from space so that other vehicles can't accidentally
        // react to it
            x = -50;
            y = -50;
            VehicleSim.world.remove(this);
        }

        // kill another vehicle
        protected function kill(v:Vehicle):Boolean{
```

```
				// already dead
				if (v.dead)
				{
					VehicleSim.world.dispatchEvent(new
StoryEvent(StoryEvent.FOUND_BODY,this,v));
					return true;
				}
				// skill check
				else if (Math.random() < assassinSkillCheck){
					// success!
					v.die();
					VehicleSim.world.dispatchEvent(new
StoryEvent(StoryEvent.DEATH,this,v));
					return true;
				}
				else{
					// failure
					VehicleSim.world.dispatchEvent(new
StoryEvent(StoryEvent.ASSASSIN_FAILURE,this,v));
					return false;
				}
			}

			// this Vehicle is killed
			public function die():void{
				graphics.lineStyle(3,0,1);
				graphics.drawCircle(wSize/2,hSize/2,Math.max(wSize,hSize));
				dead = true;
			}

			// Returns string listing all targets on list not yet found
			public function hitList():String{
				var s:String = "";
				for each (var t:Vehicle in targets){
					s += " "+t.toString();
					if (targets.indexOf(t) < targets.length - 2)
						s += ","
					if (targets.indexOf(t) == targets.length - 2)
						s += " and";
				}
				return s;
			}

			public override function toString():String{
				return characterName;
			}
		}
}
```

```
package
{
	import flash.geom.Point;
	import vehicles.Vehicle;

	/**
	 * Capable of detecting Sources in game.
	 * Each Vehicle has two, as left and right "eyes".
	 */
	public class Sensor
	{
		// vision bounds
		private static var halfAngle:Number = Math.PI*.65;
		private var upperVisionBound:Number;
		private var lowerVisionBound:Number;
		public var sightRadius:Number = 300;

		public var proximity:Number;  // how strongly is the sensor
// triggered? In this case, measures closeness of other vehicles in sight
		private var _loc:Point;				// location
		private var _dirRadians:Number;			// orientation

		public function Sensor(loc:Point,dir:Number,left:Boolean)
		{
			_loc = loc;
			_dirRadians = dir;

			//left vs. right sensors
			if (left){
				upperVisionBound = halfAngle;
				lowerVisionBound = Math.PI*-.5;
			}
			else{
				upperVisionBound = Math.PI*.5;
				lowerVisionBound = halfAngle*-1;
			}

		}

		// used for debugging
		public function visionBoundPoints():Array{
			return [new Point(sightRadius * -1 *
Math.sin(upperVisionBound), sightRadius * -1 * Math.cos(upperVisionBound)),
				new Point(sightRadius * -1 *
Math.sin(lowerVisionBound), sightRadius * -1 * Math.cos(lowerVisionBound))];
		}

		public function set location(loc:Point):void{
			_loc = loc;
		}

		public function get location():Point{
			return _loc;
		}

		public function set direction(dir:Number):void{
```

```
			_dirRadians = dir;
		}

		// Uses combination of number of vehicles and closeness of each
		// to calculate "crowdedness" of each direction.
		public function senseVehicles(vehicles:Array):void{
			proximity = 0;
			for each (var s:Source in vehicles){
				if (s is Vehicle)
					proximity += calcProximity(s);
			}
		}

		// Linear (make inverse proportional instead?)
		public function calcProximity(s:Source):Number{
			var dist:Number = Point.distance(_loc, new Point(s.x,s.y));
			return Math.max(100 - (100*dist/sightRadius),0);
		}

		public function sourceInSight(source:Source):Boolean{
			if (Point.distance(location,source.location) > sightRadius)
				return false;

			var distToSource:Point =
						source.location.subtract(location);
			// Apparently the y-coordinate actually comes first,
			// whatever the tooltip says
			var angleToSource:Number = Math.atan2(-
1*distToSource.y,distToSource.x);
			var sightAngle:Number = angleToSource - _dirRadians;
			if (sightAngle > Math.PI){
				sightAngle = Math.PI*2 - sightAngle;
			}
			return lowerVisionBound < sightAngle && sightAngle <
upperVisionBound;
		}


	}
}
```

```
package vehicles
{
      import controllers.Frame;
      import controllers.VehicleSim;

      import events.PlayerMessage;
      import events.StateChangeEvent;

      import flash.events.Event;
      import flash.events.KeyboardEvent;
      import flash.geom.Point;
      import flash.ui.Keyboard;

      /**
       * Player-controlled vehicle.
       */
      public class PlayerCharacter extends Vehicle
      {
            private var speed:Number = 10;                  // baseline speed
            private var macGuffinTarget:MacGuffin;          // target
MacGuffin for theft mission
            private var assassinTarget:Vehicle;       // target Vehicle for
                                              // assassination mission
            private var missionID:String = null;      // indicates which
                                                      // mission is in play

            public static const THEFT_MISSION:String = "Theft mission";
            public static const ASSASSIN_MISSION:String = "Assassination
mission";

            public function PlayerCharacter(mission:String)
            {
                  super();
                  // draw
                  graphics.beginFill(0xffffff);
                  graphics.lineStyle(1,0,1);
                  graphics.drawRect(0, 0, 35, 35);
                  graphics.endFill();
                  graphics.drawCircle(17,0,3);
                  placeRandomly();

                  VehicleSim.world.stage.addEventListener(
                                    KeyboardEvent.KEY_DOWN,controls);
                  zoneOfControl = 60;
                  missionID = mission;
            }

            public override function missionSetup():void{
                  switch(missionID){
                        case ASSASSIN_MISSION:
                              // make sure you can't be sent to kill yourself
                              while (!assassinTarget ||
                                    assassinTarget == this)
                                    assassinTarget =
                                    VehicleSim.world.getRandomVehicle();
                              if (!assassinTarget)
```

```
                                        throw new Error("Insufficient vehicles of
target type");
                                else
                                        // print
                                        VehicleSim.world.dispatchEvent(new
PlayerMessage(PlayerMessage.ASSASSIN_MISSION,assassinTarget));
                                break;
                        case THEFT_MISSION:
                                // get theft target
                                macGuffinTarget =
VehicleSim.world.getRandomMacGuffin();
                                if (!macGuffinTarget)
                                        throw new Error("MacGuffin not found");
                                else
                                        // print
                                        VehicleSim.world.dispatchEvent(new
PlayerMessage(PlayerMessage.THEFT_MISSION,null,macGuffinTarget));
                                break;
                        default:
                                break;
                }
        }

        // check if mission is complete
        protected override function get missionComplete():Boolean{
                switch(missionID){
                        case ASSASSIN_MISSION:
                                return assassinTarget.dead;
                                break;
                        case THEFT_MISSION:
                                return hasMacguffin(macGuffinTarget);
                                break;
                        default:
                                return false;
                                break;
                }
        }

        // keyboard controls
        public function controls(ke:KeyboardEvent):void{
                if (!dead){
                        // movement: backwards/ forwards and rotating,
                        // using arrow keys
                        var keycode:uint = ke.keyCode;
                        var oldXY:Point = new Point(x,y);
                        var oldRotation:Number = rotation;
                        if (keycode == Keyboard.UP){
                                x += speed*Math.sin(rotation*Math.PI/180);
                                y -= speed*Math.cos(rotation*Math.PI/180);
                        }
                        if (keycode == Keyboard.DOWN){
                                x -= speed*Math.sin(rotation*Math.PI/180);
                                y += speed*Math.cos(rotation*Math.PI/180);

                        }
                        if (keycode == Keyboard.LEFT)
```

```
                        rotateThetaDegrees(-10);
                    if (keycode == Keyboard.RIGHT)
                        rotateThetaDegrees(10);
                    // check for collisions
                    var obs:Obstacle = hasCollision();
                    if (obs){
                        // if at door and ready to exit, do so
                        if (obs.isDoor && missionComplete)
                            exit();
                        // otherwise, back up
                        else{
                            x = oldXY.x;
                            y = oldXY.y;
                            rotation = oldRotation;
                        }
                    }

                    // theft: spacebar
                    if (keycode == Keyboard.SPACE)
                    {
                        for each (var s:Source in sourcesInSight){
                            if (inZOC(s) && s is Vehicle){
                                var v:Vehicle = Vehicle(s);
                                pickPocket(v);
                                return;
                            }
                        }
                        VehicleSim.world.dispatchEvent(new
PlayerMessage(PlayerMessage.INVALID_ACTION));
                    }

                    // assassination: shift key
                    if (keycode == Keyboard.SHIFT){
                        for each (var s2:Source in sourcesInSight){
                            if (inZOC(s2) && s2 is Vehicle){
                                var v2:Vehicle = Vehicle(s2);
                                if (!v2.dead){
                                    kill(v2);
                                    return;
                                }
                            }
                        }
                        VehicleSim.world.dispatchEvent(new
PlayerMessage(PlayerMessage.INVALID_ACTION));
                    }
                }
            }

            // PC death: switch to state END
            public override function die():void{
                super.die();
                Frame.frame.dispatchEvent(new
StateChangeEvent(StateChangeEvent.PLAYER_DEATH));
            }

            // leave the room: switch to state END
```

```
		protected override function exit():void{
			super.exit();
			Frame.frame.dispatchEvent(new
StateChangeEvent(StateChangeEvent.PLAYER_SUCCESS));
		}
	}
}
```

```
package vehicles
{
      import controllers.VehicleSim;
      import events.VisionEvent;

      /**
       * Vehicle that searches for and pursues some other vehicle(s).
       * Basically an abstract class; all actual instances of this
       * will belong to a subclass and inherit these functions.
       * Subclasses include MacguffinSeekVehicle,
       * AssassinVehicle, KleptoVehicle.
       */
      public class PursuitVehicle extends Vehicle
      {
            protected var currentTarget:Vehicle = null;
                                    // vehicle currently being pursued

            public function PursuitVehicle()
            {
                  super();
            }

            /**
             * Overridden by subclasses; returns an
             * identified target vehicle in this vehicle's visual range,
             * if any such exist.
             */
            protected function seesTarget():Vehicle{
                  return null;
            }

            public override function updateSpeed():void{
                  if (!missionComplete)
                  {
                        // if there is no target in sight, or we've lost
                        // sight of a previously spotted target,
                        // look for a new one
                        if (!currentTarget || !sourceInSight(currentTarget)){
                              currentTarget = seesTarget();
                              // found new target
                              if (currentTarget)
                                    VehicleSim.world.dispatchEvent(new
VisionEvent(VisionEvent.FOUND_TARGET,this,currentTarget));
                        }
                        // if some target is now in sight, follow it
                        if (currentTarget){
                              aimForSource(currentTarget);
                              // if target is within range, take some action
                              if (inZOC(currentTarget))
                                    targetCaught();
                        }
                        // no targets visible: wander gamespace
                        else
                              wander();
                  }
                  else{
```

```
                    // if mission is complete, exit room
                    aimForPoint(VehicleSim.world.exitLocation);
                }
            }

            /**
             * Dummy function overridden by subclasses;
             * called when this vehicle catches its
             * current target
             */
            protected function targetCaught():void{

            }
        }
    }
```

```
package vehicles
{
	import controllers.VehicleSim;
	import events.PlayerMessage;
	import events.StoryEvent;
	import events.VisionEvent;

	/**
	 * Vehicle given a hit list of other vehicles
	 * it wants to kill.
	 */
	public class AssassinVehicle extends PursuitVehicle
	{
		public function AssassinVehicle()
		{
			color = 0x888888;
			super();
			// Number of targets on hit list
			numTargets = VehicleSim.world.assassinTargets;
		}

		public override function missionSetup():void{
			targets = new Array();
			// Get targets from VehicleSim
			while (targets.length < numTargets){
				var v:Vehicle = VehicleSim.world.getRandomVehicle();
				if (!v)
					throw new Error("Insufficient vehicles of
target type");
		// Make sure targets are neither oneself nor already on the list
				else if (v != this && targets.indexOf(v) < 0)
					targets.push(v);
			}
			// Prints hit list to newsfeed
			// (if this is one of the events VehicleSim registers)
			if (targets.length > 0){
				VehicleSim.world.dispatchEvent(new
StoryEvent(StoryEvent.HITLIST_UPDATE,this));
				// notify the player if targeted
				if (targets.indexOf(VehicleSim.world.pc) >= 0)
					VehicleSim.world.dispatchEvent(new
PlayerMessage(PlayerMessage.TARGETED,this));
			}
		}

// Returns a target on the list in AssassinVehicle's visual field, if any
		protected override function seesTarget():Vehicle{
			for each (var v:Vehicle in targets){
				if (sourceInSight(v))
					return v;
			}
			return null;
		}

		public override function updateSpeed():void{
			super.updateSpeed();
```

```
		// If a target is found already dead, cross them off the list
			if (currentTarget && currentTarget.dead){
				VehicleSim.world.dispatchEvent(new
StoryEvent(StoryEvent.FOUND_BODY,this,currentTarget));
				crossOffTarget();
			}
		}

		// Once in range of a target, attempt assassination
		protected override function targetCaught():void{
			var success:Boolean = kill(currentTarget);
			if (success){
				crossOffTarget();
			}
		}

		// Remove current target from list
		private function crossOffTarget():void{
			if (!currentTarget)
				throw new Error("No current target");
			if (targets.indexOf(currentTarget) < 0)
				throw new Error("Target not on list");
			targets.splice(targets.indexOf(currentTarget),1);
			currentTarget = null;
			if (targets.length > 0)
				VehicleSim.world.dispatchEvent(new
StoryEvent(StoryEvent.HITLIST_UPDATE,this));
			else {
				missionComplete = true;
				VehicleSim.world.dispatchEvent(new
StoryEvent(StoryEvent.MISSION_COMPLETE,this));
			}
		}

	}
}
```

```
package vehicles
{
      import controllers.VehicleSim;
      import events.StoryEvent;
      import events.VisionEvent;
      import flash.geom.Point;

      /**
       * Vehicle that searches for a specific MacGuffin.
       * Its goal is to steal its target object and leave the room.
       */
      public class MacguffinSeekVehicle extends PursuitVehicle
      {
            private var macGuffinHolder:Vehicle = null;
                  // vehicle in possession of target MacGuffin, if known
            private var targetItem:MacGuffin;         // target MacGuffin

            public function MacguffinSeekVehicle()
            {
                  color = 0x0000ff;
                  super();
            }

            // Returns true if sees a vehicle carrying targetItem
            protected override function seesTarget():Vehicle{
                  return seesMacguffinVehicle(targetItem);
            }

            // Sets targetItem
            public override function missionSetup():void{
                  targetItem = VehicleSim.world.getRandomMacGuffin();
                  if (!targetItem)
                        throw new Error("MacGuffin not found");
            }

            // Checks if this vehicle has targetItem before running inherited
            // function, to account for case where MacguffinSeekVehicle had
            // item and lost it.
            public override function updateSpeed():void{
                  missionComplete = hasMacguffin(targetItem);
                  super.updateSpeed();
            }

            // If this vehicle has caught up with the vehicle carrying
            // targetItem, picks its pocket
            protected override function targetCaught():void{
                  var success:Boolean = pickPocket(currentTarget,targetItem);
                  if (success){
                        currentTarget = null;
                        /*    Check to make sure macguffin is in possession!
                              "Success" may mean successfully picking the
                              pocket of someone who no longer has the item.
                        */
                        if (hasMacguffin(targetItem)){
                              // If item obtained, proceed to exit
```

```
                                        VehicleSim.world.dispatchEvent(new
StoryEvent(StoryEvent.MISSION_COMPLETE,this));
                                        missionComplete = true;
                                }
                        }
                }
        }
}
```

```
package vehicles
{
	import controllers.VehicleSim;
	import events.VisionEvent;

	/**
	 * Vehicle that steals any MacGuffins it finds.
	 * Not to be confused with MacguffinSeekVehicle,
	 * which steals only one particular MacGuffin.
	 */
	public class KleptoVehicle extends PursuitVehicle
	{

		public function KleptoVehicle()
		{
			color = 0x00ffff;
			super();
		}

		/**
		 * Checks if a vehicle in possession of a MacGuffin
		 * is in sight.
		 */
		protected override function seesTarget():Vehicle{
			for each (var s:Source in sourcesInSight){
				if (s is Vehicle){
					var v:Vehicle = Vehicle(s);
					if (v.hasMacguffin())
						return v;
				}
			}
			return null;
		}

		/**
		 * Once a target is located, picks its pocket.
		 */
		protected override function targetCaught():void{
			var success:Boolean = pickPocket(currentTarget);
			if (success)
				currentTarget = null;
		}

	}
}
```

```
package vehicles
{
	/**
	 * Basic vehicle that wanders the scene randomly
	 * with no other goals
	 */
	public class WanderVehicle extends Vehicle
	{
		public function WanderVehicle()
		{
			color = 0xffff88;
			super();
		}

		public override function updateSpeed():void{
			wander();
		}
	}
}
```

```
package
{
	import flash.display.Sprite;
	import flash.text.TextField;
	import flash.text.TextFormat;

	import vehicles.Vehicle;

	/**
	 * In-game object, transferable between vehicles.
	 */
	public class MacGuffin extends Sprite
	{
		private var _itemName:String;
		public var label:TextField;
		public var slot:int = -1;
			// indicates where to render image relative to vehicle
		private static var nameBank:Array = ["Maltese Falcon",
					"diamonds", "briefcase", "missile plans"];
		private static var namesUsed:Array = new Array();
		private static var nameCount:int = 0;

		public function MacGuffin()
		{
			super();
			_itemName = getName();
			// draw
			graphics.beginFill(0x00ff00);
			graphics.lineStyle(1,0,1);
			graphics.drawCircle(0,0,3);
			// display name
			label = new TextField();
			label.text = _itemName;
			var format:TextFormat = new TextFormat(null,8);
			label.setTextFormat(format);
			addChild(label);
		}

		public function get itemName():String{
			return _itemName;
		}

		/**
		 * Set Vehicle v as the character holding this item;
		 * place item in slot slotNumber
		 */
		public function setParent(v:Vehicle,slotNumber:int):void{
			v.addChild(this);
			slot = slotNumber;
			x = ((slot%3) + 1)*(v.wSize/4);
			y = (Math.floor(slot/3) + 1)*(v.hSize/4);
		}

		public override function toString():String{
			return _itemName;
		}
```

```
        /**
         * This function is used to generate names to ensure that
         * each name is unique to a single MacGuffin.
         * To avoid confusion if there are more items than names,
         * any additional items are assigned a previously used
         * name with a number attached (Name1, Name2, etc.).
         */
        private static function getName():String{
              // choose random name from those used least so far
              var randomIndex:int =
                          Math.floor(Math.random()*nameBank.length);
              var name:String = nameBank[randomIndex];
              nameBank.splice(randomIndex,1);
              namesUsed.push(name);
              // count number of times this name has been used
              if (nameCount > 0)
                    name += nameCount;
              // if we've run through the entire list
              if (nameBank.length == 0)
              {
                    nameCount++;
                    nameBank = namesUsed;
                    namesUsed = new Array();
              }
              return name;
        }
    }
}
```

```
package ui
{
	import flash.text.TextField;

	/**
	 * Newsfeed that shows what's happening in the game.
	 * Displays the last numLinesVisible lines, but saves all
	 * (possibly useful for reviewing the sequence of events
	 * after the fact)
	 */
	public class NewsDisplay extends TextField
	{
		private var numLinesVisible:int;
					// number of lines displayed on the screen
		private var textLines:Array;
					// record of all text sent to display

		public function NewsDisplay(xcoord:Number,ycoord:Number,
							w:Number,h:Number,numLines:int)
		{
			super();
			text = "";
			x = xcoord;
			y = ycoord;
			height = h;
			width = w;
			numLinesVisible = numLines;
			textLines = new Array();
		}

		/**
		 * Adds a line of text to the display
		 */
		public function addLine(line:String):void{
			textLines.push(line);
		// Update display: only the last numLinesVisible lines are shown
			var subarray:Array = textLines.slice(-1*numLinesVisible);
			text = subarray.join("\n");
		}

	}
}
```

```
package events
{
      import flash.events.Event;
      import vehicles.PlayerCharacter;
      import vehicles.Vehicle;

      /**
       * Story-related events that happen during the game.
       */
      public class StoryEvent extends Event
      {
            // Event types
            public static const MISSION_COMPLETE:String = "Mission
completed";
            public static const MISSION_FAILED:String = "Mission failed";
            public static const ITEM_THEFT:String = "Item stolen";
            public static const DEATH:String = "Someone died";
            public static const ASSASSIN_FAILURE:String = "Failure to kill
someone";
            public static const THEFT_FAILURE:String = "Failure to steal
something";
            public static const NULL_THEFT:String = "Picked pockets but found
nothing";
            public static const EXIT:String = "Someone left the room";
            public static const ENTRY:String = "Someone entered the room";
            public static const FOUND_BODY:String = "Found a dead body";
            public static const HITLIST_UPDATE:String = "Hit list updated";

            private var _subject:Source;
            private var _object:Source;
            private var _item:MacGuffin;
            public var playerEvent:Boolean;
                  // Is the player character involved in this event?

            public function StoryEvent(type:String, subject:Source,
object:Source=null, item:MacGuffin=null)
            {
                  super(type);
                  _subject = subject;
                  _object = object;
                  _item = item;
                  // check if player is involved
                  playerEvent = _subject is PlayerCharacter || _object is
PlayerCharacter;
            }

      // Determines what is printed by NewsDisplay for each type of event
            public override function toString():String{
                  switch (type){
                        case MISSION_COMPLETE:
                              return _subject + " has completed its mission";
                              break;
                        case MISSION_FAILED:
                              return _subject + "has failed in its mission";
                              break;
                        case ITEM_THEFT:
```

```
                              return _subject + " stole " + _item + " from "
+ _object;
                              break;
                        case DEATH:
                              return _subject + " killed " + _object;
                              break;
                        case ASSASSIN_FAILURE:
                              return _subject + " tried and failed to kill "
+ _object;
                              break;
                        case THEFT_FAILURE:
                              if (_item)
                                    return _subject + " tried and failed to
steal " + _item + " from " + _object;
                              else
                                    return _subject + " tried and failed to
pick " + _object + "'s pocket";
                              break;
                        case EXIT:
                              return _subject + " has left the room";
                              break;
                        case ENTRY:
                              return _subject + " has entered the room";
                              break;
                        case NULL_THEFT:
                              return _subject + " picked " + _object + "'s
pocket but found nothing";
                              break;
                        case FOUND_BODY:
                              return _subject + " found the body of " +
_object;
                              break;
                        case HITLIST_UPDATE:
                              var v:Vehicle = Vehicle(_subject);
                              return _subject + " is now after " +
v.hitList();
                              break;
                        default:
                              return type;
                              break;
                  }
            }
      }
}
```

```
package events
{
      import flash.events.Event;
      import vehicles.Vehicle;

      /**
       * Events that occur when a vehicle sees or loses sight of
       * another vehicle. VisionEvents are generated every time, but
       * VehicleSim determines which ones are sent to the NewsDisplay.
       */
      public class VisionEvent extends StoryEvent
      {
            // Event types
            public static const WAS_SEEN:String = "was seen by";
            public static const WAS_LOST:String = "was lost by";
            public static const SAW:String = "saw";
            public static const LOST:String = "lost";
            public static const FOUND_TARGET:String = "found target";

            private var _viewer:Vehicle;  // object doing the viewing
            private var _seenObject:Source;     // object seen by viewer

            public function VisionEvent(type:String, looking:Vehicle,
                                        seen:Source)
            {
                  var subject:Source;
                  var object:Source;

                  if (type == WAS_SEEN || type == WAS_LOST)
                  {
                        subject = seen;
                        object = looking;
                  }
                  else
                  {
                        subject = looking;
                        object = seen;
                  }
                  super(type, subject, object);
                  _viewer = looking;
                  _seenObject = seen;
            }

            public function get seeingVehicle():Vehicle{
                  return _viewer;
            }

            public function get seenObject():Source{
                  return _seenObject;
            }

            public override function toString():String{
                  if (type == WAS_SEEN || type == WAS_LOST)
                        return _seenObject + " " + type + " " + _viewer;
                  else
                        return _viewer + " " + type + " " + _seenObject;
```

```
            }
        }
}
```

```
package events
{
	import controllers.VehicleSim;
	import flash.events.Event;
	import vehicles.Vehicle;

	/**
	 * Messages particularly relevant to the player:
	 * error warnings, results of player actions, etc.
	 * These get sent to their own NewsDisplay above the general one.
	 */
	public class PlayerMessage extends Event
	{
		// Event types
		public static const INVALID_ACTION:String =
						"Can't do that--no targets in range";
						// Error message for invalid actions
		public static const TARGETED:String = "Target!";
				// Player is being targeted by an AssassinVehicle
		public static const NAME:String = "Your name is ";
						// Announces player's character name
		public static const ASSASSIN_MISSION:String =
						"Your mission is to assassinate ";
					// Player's target Vehicle for assassin mission
		public static const THEFT_MISSION:String =
								"Your mission is to steal ";
					// Player's target MacGuffin for theft mission

		private var _other:Vehicle;
		private var _item:MacGuffin;

		public function PlayerMessage(type:String, other:Vehicle=null,
							item:MacGuffin=null)
		{
			super(type);
			_other = other;
			_item = item;
		}

		/**
		 * Defines message printed by NewsDisplay
		 */
		public override function toString():String{
			switch(type){
				case TARGETED:
					return "Warning: " + _other + " is after you!";
					break;
				case NAME:
					return type + _other;
					break;
				case ASSASSIN_MISSION:
					return type + _other;
					break;
				case THEFT_MISSION:
					return type + _item;
				default:
```

```
                        return type;
                    }
                }
            }
        }
```

```
package
{
	import controllers.VehicleSim;
	import flash.geom.Point;

	/**
	 * Walls and other scenery
	 */
	public class Obstacle extends Source
	{
		// Doors are selectively permeable; all other obstacles are solid
		public var isDoor:Boolean = false;

		public function Obstacle(w:Number=25,h:Number=25,locx:Number=-
1000,locy:Number=-1000,door:Boolean=false)
		{
			super();
			isDoor = door;
			// draw
			if (isDoor)
				graphics.beginFill(0xffffff);
			else
				graphics.beginFill(0x00);
			graphics.drawRect(0,0,w,h);
			graphics.endFill();
		// Either use specified coordinates or set location at random
			if (locx >= -50 && locy >= -50)
			{
				x = locx;
				y = locy;
			}
			else{
				placeRandomly();
			}
		}
	}
}
```

```
package controllers
{
	import events.StateChangeEvent;
	import flash.display.Sprite;
	import flash.events.MouseEvent;
	import flash.text.TextField;
	import ui.Button;

	/**
	 * State displayed after game ends, through player death
	 * or player success.
	 */
	public class EndGame extends Sprite
	{
		public function EndGame(won:Boolean)
		{
			super();

			// Message indicating reason for end of game
			var text:TextField = new TextField();
			if (won)
				text.text = "Player successfully completed mission!";
			else
				text.text = "Player death";
			text.x = 300;
			text.y = 200;
			text.width = 200;
			addChild(text);

			// Button to return to UserOptions state
			var button:Button = new Button("Play again?",
							restart,400,500);
			addChild(button);
		}

		// Return to UserOptions state
		private function restart(me:MouseEvent):void{
			parent.dispatchEvent(new StateChangeEvent(
					StateChangeEvent.NEW_GAME));
		}
	}
}
```