A Grammar-Based System for Game Playing with a Sensor Network

Jonathan Schwarz
jonathan.schwarz@yale.edu
Davenport 2006
Electrical Engineering and Computer Science
Advisor: Andreas Savvides

I. INTRODUCTION

Sensor networking is a rapidly expanding field, with many applications in research and everyday life [6] [4]. While research in sensor networking is growing, the ease of use of creating systems of sensors is still not an easy task. Programming for a sensor operating system such as SOS requires a thorough understanding of how the OS works, loads modules, passes messages, and is configured [4]. In addition, most sensor network systems are built expressly for one type of task, rather than allowing broad application. To use them in a method not originally imagined can take significant work. A method known as hierarchical sensory grammars would change this. Using generally designed rules, a system could generate code from the rule definitions that would then parse and handle input based on the rules. By allowing the rules to be generic, almost any type of behavior can be created by simply changing the rules. This has the effect of working in any application as well as abstracting away the underlying system, making it easier to use.

II. RELATED WORK

This project builds heavily on the work done at ENALAB on the sensory grammars project [3]. While it uses the approach of defining areas much like the cooking activity recognition project, the implementation and real-time feedback are more similar to the turning grammar developed there. Elements of both projects are used as the basis for the concept and implementation details of this project.

Using a sensor network for game playing has been explored by several research groups. The involvement of the network varies from the concept of the network controlling the game entirely to using the network to administer the game and control the setting [1] [2].

Novel concepts in this project include the use of a hierarchical sensory grammar as the basis for a game, and the specific design of the grammar, using pairs of tokens instead of single characters.

III. OVERVIEW

The basis of the underlying system is that SOS modules are automatically generated based on the definitions of a grammar provided by the user. These modules run on SOS nodes, whether XYZ nodes or PC hardware simulating them, and are capable of handling as input tokens following the rules of the grammar that was defined [5]. Creation of the SOS modules is done by using a desktop tool that takes in only grammar definitions and output mappings to define the flow of communication between modules. Bison and Flex, both free software released under the GPL, handle the generation of C programs. Bison is able to generate C code that can parse a grammar based on the grammar definition it is given. Similarly, Flex is used to create a C lexer that handles pattern matching on the text input to the grammar. This C code is then wrapped in custom code that creates SOS modules. Multiple modules are configured to communicate by creating an output table that specifies how the modules will pass messages between themselves. In this way, all the benefits of using SOS modules can be had without needing to know how to program modules for SOS.

The automated SOS module creation process.
IV. GAMEPLAY

The game is based on following traffic laws. A simplified street map, similar to what can be obtained through Google Earth or other similar products, is displayed on the floor by a projector hung from the ceiling. A simple USB camera, similar to webcams that are available commercially, is suspended near the projector. The player’s objective is to navigate the streets from a designated start to a finish point while losing as few points in the process. The player controls an inexpensive remote control car approximately two inches long. The car has simple controls, only forward and reverse, left and right. There is no proportional control for the car, the throttle and steering are either full speed or full turn or off.

Each player starts with 100 points, and loses one point for driving off the road, two points for driving the wrong way on a one-way street, and ten points for hitting any obstacles. There is no time element to the scoring, although that has been considered as a future addition. Scoring is handled on a per-incident basis, meaning that repeatedly driving off the road continues to deduct points until the player returns to the road. This scoring strategy discourages use of creative shortcuts.

V. DRIVING GRAMMAR

The grammar is defined to recognize the various rule violations in order to score those situations. Some positive situations (non-rule-breaking) are defined in order to allow for easy parsing of the grammar. Each rule sends a token to the scoring module using the function produce(), which writes a character to the output buffer of the module. The grammar is based off the concept that each input to the parser will consist of an (area, direction) pair. This is enough information to determine if the player is violating a rule at any given point. For some rules, the direction is inconsequential, but the parsing structure requires the direction to be sent to maintain the pair format. There are three types of areas defined in the grammar, and five unique areas overall. The types of areas include: roads, obstacles, and non-road areas. There are five areas total because there are three roads which must be defined individually. Part of the reason for this is one-way roads must have their own rule, which defines the direction of travel that is illegal. For example, if a one-way road heads north, only driving south is illegal. Area computation is handled by a simple if-then statement in one module. Areas are listed in order of priority, with the most important first. In this way, if a one-way road overlaps a normal road, the one-way road must be defined first if the intersection is to behave as expected. Otherwise, drivers could head in the illegal direction because the rule for the top road would take precedence. In addition to the four cardinal directions (W, N, E, & S), a token representing a null direction (X) was created. This was necessary because the module that handles directions can only send a direction after two updates from the camera have arrived. A null direction is considered legal on a one-way road.

In the grammar, rules can be defined as combinations of tokens, or of tokens and other rules. An example of this is the simplest rule, that of being off the road. This rule states that the player is off the road if they are in the non-road area (B), and are traveling in any direction (D). D is defined as the tokens ‘W’, ‘N’, ‘E’, or ‘S’. Tokens are differentiated from rules in definitions by the use of single quotes. Any token (a
direct input) must be quoted, while a rule does not use quotes. This makes it possible to have the ‘W’ token, representing traveling west, and the W rule, representing driving the wrong way.

The K rule, representing legal area/direction combinations, was necessary because of the decision to use pairs of tokens. In Bison, the parser generates an error if an input token is found that doesn’t match any grammar definition. This error can be handled, but causes the parser to skip the rest of the tokens in its buffer. The only way to handle this is to tell Bison what token to start processing on again, which is not possible with this grammar. A grammar that has a newline after every set of tokens could match “error ‘n’”, but we use a simple stream of characters without any separators. Instead, the K rule matches all two-way roads and all legal directions on one-way roads, allowing them to be parsed without errors. Separators could have been added to the grammar to allow the error matching to behave as expected, but this would have complicated the modules and grammar more than adding a single rule to the grammar. The full grammar follows below.

```
M : /* empty */
| M E /* Exception */
| M K { produce('K'); } /* oK */
| M error { produce('E'); } /* Error */;

E : O { produce('O'); }
| W { produce('W'); }
| H { produce('H'); }

K : /* define the oK states */
    'T' D | 'R' 'W' | 'R' 'E' | 'R' 'S' | 'R' 'X' | 'S' D;

O : 'B' D /* Offroad = Building + Direction */;

H : 'P' D /* Hit = Pedestrian + Direction */;

W : 'R' 'S' /*Wrongway = Road + South */;

D : /* Directions = West, North, East, South or undefined (first update) */
    'W'| 'N' | 'E'| 'S' | 'X';
```

VI. SOFTWARE INFRASTRUCTURE

There are four major components used in the game system. The first is the only one that’s not an SOS module, the camera level. The camera is connected to the AER Emu software, which is able to emulate an address-event imager with standard camera hardware [7]. The software is used to send centroid updates to the SOS modules. Rather than detecting motion, a static background is configured, and each frame captured is compared to that initial frame. Any difference is seen as motion, and the centroid of that motion is calculated. This information is then sent to a standard TCP socket. The software was configured to send updates as frequently as possible, which translated to a framerate of almost exactly 15fps, which allowed for very fine resolution of movement.

The next module received the centroid information from a socket, and used that information to calculate the area and direction of the car. Areas are calculated based on rectangular regions of the image that are mapped beforehand. As mentioned before, areas are listed in priority order so that the first area that matches is used. Directions are calculated by calculating the difference between two successive centroids. Since only the cardinal directions are used, the path of a car traveling northwest must be examined to decide if it is more north or more west. Theoretically, it would be possible to drive the wrong way down a one-way street by tacking (as in sailing) at extreme angles to the main direction of the road. However, this is highly unlikely, and next to impossible to do in practice, as the cars are not that precise. This module sends two SOS messages containing the characters representing the area and the direction, respectively, to the next module in the output mapping.

The third module contains the main driving grammar. It parses the pairs of the input tokens as described earlier, and outputs single tokens described what violation has occurred, if any, and a standard “valid” token otherwise. This token is passed on to the scoring module.
The scoring module is very simple. It also is dynamically generated from a grammar definition, but its rules are much simpler. It simply reads each incoming token, and either deducts the appropriate number of points from the score, or does nothing. In a more advanced version of the game, this module could have more complex functionality that would incorporate these valid driving tokens, such as giving bonuses for staying on the road for a certain amount of time.

VII. EVALUATION

The game worked well, proving to be surprisingly fun to play despite its simplicity. The scoring was highly accurate, which was how most participants in the game judged the system. After making a few preliminary runs, they would often drive the car right up to the edge of an area on the floor, watching the display to see if it would be scored correctly. Almost all of the time, the resolution of the camera and the calibration of the system was satisfactory. The projected map was approximately six feet wide, so a resolution of 320x240 pixels gave roughly quarter inch accuracy. Since the car was about an inch wide, this was small enough to allow the game to be realistic. This accuracy required a good deal of calibration, however, which was one of the most time-consuming stages of the project. The camera had to be carefully aligned with the projected image, and the mount was not completely stable. Once aligned, an image from the camera was used to map the corners of each area into pixel values. Automatic calibration based on sensing the edges of the image or specially placed corner indicators would be a huge boon, but are not necessary for the system to be considered a success.

A weakness of the game, rather than the system itself, was how the game was started and stopped. Start and end areas were marked on the map, and moved for different levels of differing complexities. Defining these spots as areas within the game would have allowed the game to be timed and stopped and started by the system. Instead, these tasks were performed manually, and the program had to be restarted each run to reset the scoring. Performing these tasks automatically would have been relatively simple, and would have made the gameplay smoother.

Finally, one issue with the system was discovered when using the error handling mechanism in Bison. When passing in tokens that were not defined in the grammar, the module parsing them would experience a memory leak and crash soon after starting. It was discovered that the token input buffer was not being cleared when Bison encountered an error, and it would quickly fill up with unprocessed tokens. Previous grammars that were tested on the system didn’t have a need for error handling, and so this situation was never encountered. Fortunately, it was easily fixed by clearing the buffer every time the parser returned, whether or not it had parsed successfully.

VIII. CONCLUSIONS

The sensory grammars system worked well for this project. It does perform as advertised, allowing complex behaviors to be described using only a few lines of grammar. The benefits of using the system are twofold: first, grammars allow powerful rules to be created with minimal effort, and secondly, users are able to create SOS modules without having to program them manually. The combination of these two effects means that anyone who can write a set of rules can create their own sensor
networking system without being able to program. While the audience of people interested in doing such a thing without possessing any technical skills is small, it is beneficial to almost anyone working with SOS. Rather than dealing with the tedium of writing modules, the user’s energy can be devoted entirely to creating a set of rules, a much higher-level expression of the operation of the system. The sensory grammar system provides an abstraction layer for SOS that handles rule-based systems with ease, while removing the hassle of writing directly to the SOS layer. Using SOS as a base allows for very scalable systems to be created, as modules can be run completely on one node, or spread across processors as needed. More modules can easily be added in series or parallel with any existing chain of modules, allowing complex functionality to be built quickly and easily.

IX. REFERENCES


