INTERIM REPORT

ECE 496Y

Train Set Simulation Software

For a Web-Enabled Automated Train System

Professor Paul Chow

Group 2

Tuesday, January 5, 1999.

David Tam

EXECUTIVE SUMMARY

The train group is building a web-enabled automated train system. When it is complete a user will be able to control a physical train set from anywhere in the world via the internet. The necessary software and hardware are being developed by the group. Several teams have been formed to handle the various aspects of the design. System development began in September 1998, and will be completed by April 9, 1999. This interim report covers the time period between October 27, 1998 (the day after the final proposal due date) and December 31, 1998.

The purpose of this project is to develop the train set simulation software. This software has the role of providing the location of trains to the user interface (UI) and preventing collisions. This report is intended for managers as well as technically-oriented readers.

The simulation software consists of the following high-level components: (1) interface to UI, (2) control manager, (3) track manager, (4) simulation engine, and (5) interface to command interpreter. Development of the simulation engine and the interface to command interpreter were my assigned tasks. Consequently, this report will focus on the progress of these two components.

The simulation software is approximately 3 weeks behind schedule. Approximately 75 % of the components have been developed. At this point in the project, software integration should have been be completed and extensive software testing should have begun. The team is currently concluding the development of remaining components, integrating these components and testing the software as a whole. These tasks will be completed by January 10th.

Specific to my assigned tasks, development of the interface to command interpreter is behind schedule. Currently, investigation of interprocess communication (IPC) mechanisms is being conducted to develop this interface. Development of this component will be completed by January 6th.

INTRODUCTION

Goal and Purpose

The goal of the train group is to build a train set that is controlled from a Java applet. A user, who may be located anywhere in the world, will be able to control the train set via the internet and Java applet. Both the software and hardware are being developed by the group. The group is divided into several teams which are responsible for various components of either software or hardware. On the software side, there are teams responsible for the user interface (UI), simulation software, command interpreter, and low-level drivers. The hardware teams include network protocol and physical implementation, network controller, radio link and control, engine control, switches and light interfaces, and crossings and sensor interfaces.

System development began in September 1998, and will be completed by April 9, 1999. This interim report covers the time period between October 27, 1998 (the day after the final proposal due date) and December 31, 1998. It will focus on the progress of the simulation engine and interface to command interpreter since these were my assigned tasks.

The purpose of this project is to develop the train set simulation software. This software acts as the link between the UI and the command interpreter. When a user issues commands through the UI, the request is given to the simulation software for validation. If the simulation software determines the command will not cause a collision, the command is passed to the command interpreter. Data also flows in the opposite direction. When the condition of the track changes, such as when a train triggers a sensor, the track sends this update to the simulation software via the command interpreter.

According to the timeline established in the proposal document, the simulation software, as a whole, is approximately 3 weeks behind schedule. At this point in the project, the software should have been integrated, and extensive software testing should have begun. In reality, approximately 75 % of the components have been completed. A preliminary integration test was completed successfully and indicate that the components are working together in harmony. As a team, we are currently concluding the development of components, integrating these components and testing the software as a whole. These tasks will be completed by January 10th.

Specific to my assigned tasks, development of the interface to command interpreter is the only task behind schedule. Currently, investigation of interprocess communication (IPC) mechanisms is being conducted to develop this interface. In the original timeline, the use of IPC was not anticipated and extra time was not allocated for investigation. Development of this component will be completed by January 6th.

This report has been written for both managers and technically-oriented readers. Detailed technical descriptions and discussions are located primarily in the "Process" and "Obstacles" subsections and are suitable for the technically inclined. Other such suitable sections include literature review and technical issues in productization. The remaining sections of the report are suitable for all readers.

Software Overview

Internally, the simulation software consists of five main components: (1) interface to UI, (2) control manager, (3) track manager, (4) simulation engine, and (5) interface to command interpreter. Figure 1 presents a diagram of these components and their interconnections.

The interface to UI is the communication liaison between the UI and the rest of the simulation software. This interface implements a client server architecture. The interface acts as the server and accepts concurrent internet TCP/IP connections from multiple client UIs. Through this interface, user commands are passed to the rest of the simulation software. Track state information is also passed through the interface, but in the opposite direction. In this case, data flows from the simulation software to the UI. This flow serves to update the UI on the location of the trains.

The control manager is responsible for preventing collisions. It continuously inspects the future state of the track and ensures collisions will not occur in the near future. As well, all user commands are inspected and filtered by the control manager to ensure they do not cause any problems to the track. Once past the inspection, the command is issued to the actual track.

The track manager handles the databases that store the track layout and track state conditions. Mutual exclusion is enforced by this component to allow various other components to query and update data concurrently.

The simulation engine is responsible for continuously predicting the target time in which a train will occupy its neighboring track piece. Once the target time is reached, the track manager is called to update the location. These predictions are eventually use by the UI to depict train movement. The engine also has the task of forecasting the location of trains in the near future. The forecast information is subsequently used by the control manager to prevent collisions.

The interface to command interpreter is the communication liaison between the simulation software and the command interpreter. It also hides the fact that the train tracks may be real or virtual from the rest of the simulation software. If the track is real, commands are passed to the command interpreter. If the track is virtual, commands are short-circuited and the track manager is called directly to update the desired values. The interface uses interprocess communication (IPC) to communicate with the command interpreter. Any updates from the real track are sent from the command interpreter to the interface. The interface then calls the appropriate track manager function to perform the update.

Road Map

The introduction section of the report describes the purpose of the project in detail and gives an overview of the software. An outline of the report is given to create a clear picture of the discussion.

The literature review section describes the research conducted to understand and implement the IPC mechanisms behind the interface to command interpreter. This section is more suitable for technically-oriented readers.

The program review section presents the status of each stage relevant to the development of the simulation engine and interface to command interpreter. Completed stages are described first, followed by stages in progress, and incomplete stages at the end. For completed stages, what was done, why, and how it contributes to related stages is explained. Significant obstacles are described along with solutions. For stages currently in progress, current obstacles and estimated completion times are discussed. For incomplete stages, revised forecasts of the completion dates along with predicted obstacles are discussed. Subsections entitled "Process" and "Obstacles" are more suitable for technically-oriented readers.

The technical issues in productization section deals with technical issues involved with creating a high-calibre product that meets the expectations of the current market place. This section is more suitable for technically-oriented readers.

The summary of changes to program section summarizes major deviations from the proposal, along with justification for such changes. The effects on the outcome of project are also explained.

LITERATURE REVIEW

The following reference describes the concept of process communication. It presents useful background information in designing the interface to command interpreter. It describes the high-level concepts behind interprocess communication but without any specific details.

Abraham Silberschatz and Peter Baer Galvin
Operating System Concepts 5th Edition
Addison-Wesley, Don Mills, Ontario (1998),
pages 108-120, 687-693.

The following document will be useful for developing code in the interface to command interpreter. A copy of this document can be found on the internet at
http://www.eda.bg/docs/programming/sockets/sockets.html

Stuart Sechrest
Sockets Tutorial: An Introductory 4.4BSD Interprocess Communication Tutorial
Computer Science Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley

This document provides both a general background on IPC as well as specific details and examples on the various methods of achieving interprocess communication. It led to the decision of using the socketpairs method of IPC. The examples were very helpful and will accelerate the development of the interface to command interpreter. The section on unix pipes provides background information the use of one particular IPC mechanism. The section on socketpairs builds upon the unix pipe mechanism and is the particular information that was desired. A skeleton IPC example, using socketpairs, is presented and will serve as the basis for the interface to command interpreter.

PROGRAM REVIEW

Summary

The following stages have been completed.
1. Develop basic simulation engine.
2. Develop forecast component of simulation engine.
3. Develop short-circuit block.
4. Integrate simulation engine subcomponents.

The following integration stages are currently in progress.
5. Develop interface to command interpreter.
6. Integrate simulation software components.

The following productization stages are targeted for future dates.
7. Extensive simulation software testing.
8. UI and command interpreter integration and testing.
9. Total software and hardware integration testing.
10. Fine-tuning of simulation engine.
11. Marketing.

Figure 2A and Figure 2B present selections from the original time line and a revised version which incorporates the productization phase.

COMPLETED STAGES

1. Development of Basic Simulation Engine

Introduction

The simulation engine development stage produces a working component that is responsible for predicting the target time of when a train will occupy its adjacent section of track. It acquires current track information from the track manager, produces a prediction of when the train will occupy its next section of track, and sends the new location to the track manager when the targeted time is reached.

This stage has been completed. The engine was written in C. Rather than attempting to write the most accurate simulation engine on the first try, a basic simulation engine was written and fine-tuning will be conducted during total system integration. This plan enables a more natural, iterative approach to solving the problem at hand. Also, fine-tuning is made easier when the actual functional track is available. Since this component communicates with the track manager, an interface was also defined for acquiring necessary data and sending location updates.

The Process

Before development could occur, an understanding of the requirements, limitations, and capabilities of the train system was developed with team members. These parameters defined the scope of the simulation engine's responsibilities and capabilities.

An algorithm outline was developed to gain an understanding of the magnitude of code required. This outline can be found in Appendix A. It is written in pseudo-code and shows a high-level view of the steps executed by the simulation engine. An inspection and walk-through of this algorithm was performed to determine if it was adequate for the requirements. Once validated, the outline was transformed into C source code. The inspection and walk-through process was applied to the source code to ensured the engine would perform its role adequately. Developing the code also involved defining the interface to the track manager. This definition process occurred simultaneously with code development. When data was required from the track manager, a new function interface was defined to access the data. These new interface definitions were sent to the team member responsible for track manager development.

The transformation of pseudo-code into source code required the use of unix timing facilities. These facilities used a timeval data structure that was not quite compatible with the SI units of time (seconds). Due to this situation, a set of timing operations and transformations were developed to handle basic operations under the unix time data structure. By writing this set of functions, the development of the source code for the simulation engine became slightly easier because a layer of services was available. As another benefit, the simulation engine code became more comprehensible due to these functions.

A number of agreed upon conventions were created to ensure uniformity across the entire train project. For instance, the semantics behind switch positions was defined by the low-level drivers group as LEFT equals the straight portion of a forking track piece, and RIGHT equals the curved portion of a forking track piece. Restrictions on sensor placements were discussed informally and briefly within the simulator team. These restrictions made the simulation engine slightly easier to program. Agreed upon internally, track sensors should be placed exactly in the middle of track pieces to eliminate problems with asymmetrical track lengths due to a train's direction of travel. This problem is illustrated in Figure 3A and Figure 3B. In figure 3A, the remaining track length is dependent upon the direction of travel. If a train is approaching from the left, the remaining length after tripping the sensor is 0.02 m. On the other hand, the remaining length is 0.07 m if the train is approaching from the right. This asymmetry is eliminated in Figure 3B. Track switching restrictions were also discussed briefly as this affected the simulation engine algorithm as well as the control manager. Switches are not permitted to activate when any train is occupying that specific track. Another restriction realized is that to uniquely identify the train which tripped a specific sensor, switches must be placed at the entrances and exits of every forking track piece. The above restrictions were realized and discussed as the simulation engine was being developed.

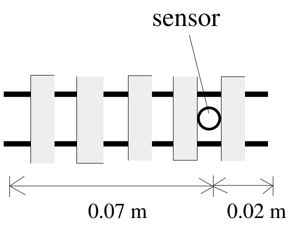

Figure 3A -- Asymmetrical Placement

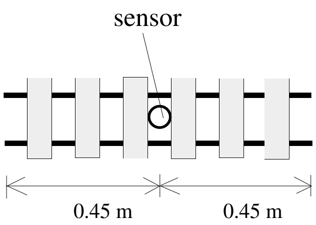

Figure 3B -- Symmetrical Placement

A major software architectural concern arose during the development of the simulation engine component. This came about due to the concern of how this particular component fit into the overall simulation software. Choices such as whether this component should run as a separate process, a separate thread, or a plain function call, required some thought. After careful consideration, it was decided that this component would run as a POSIX thread. This choice was made because (1) threads do not consume a lot of operating system overhead, (2) the simulation engine did not make any system calls that would otherwise cause the entire encapsulating process to be blocked, and (3) communication with the track manager would be much simpler because it would also reside in the same encapsulating process.

Once the code was written for the simulation engine and the supporting timing operations, an inspection and walk-through of the code was performed to ensure it was correct and contained no logical or syntax errors. Once the code compiled successfully, test driver code was written to simulate the creation of the simulation engine thread. As well, stubs were written for the various track manager function calls that were made. A simple test track was created in the stub functions. It consisted of an oval track, with two trains located at opposite ends of the track, travelling in the clockwise direction. All track pieces were of equal length. One train travelled at a speed which was twice the other. Figure 4 shows the test track. Again, these pieces of code were inspected and followed a walk-through to eliminate errors. Each of these pieces were compiled separately into object code to ensure all syntax errors had been eliminated. Once this was completed, the object code files were linked together to form a simple unit test case. The results of this simple test case can be found in Appendix B.

Figure 4 -- Test track

Obstacles

Difficulties arose during unit testing of this component. The test output revealed that only the first train was moving along the test track. The second train did not produce any output to indicate it was moving. Also, due to the fact that an infinite 'while' loop was used in the algorithm, the test program refused to terminate under the normal interrupt signal (in unix, control-C). It appeared as if the first POSIX thread (train #1) was never relinquishing control to allow the second thread (train #2) to execute. Further debugging revealed this hypothesis to be correct. Investigation into the POSIX thread library indicated there was no yield function available for allowing other threads a chance to execute. Round-robin scheduling, a plausible solution, was mentioned in the manual pages but was not implemented by the operating system. Use of the sleep() function or usleep() function (micro sleep) seemed to be the answer. Use of either of these functions at the end of 'for' and 'while' loops produced the desired results. The usleep() function was used with a parameter value of 1 microsecond. Since the sleep() function had a granularity of seconds, it was deemed inappropriate for this application.

Changes to Program

There were a number of deviations from the original plans described in the proposal document. Initially, before this component was to be developed, speed ratio differences were to be obtained. These ratios could account for speed variations due to the type of track, the type of train engine, the amount of load the train was carrying, and various other real world anomalies. It was decided that this task was only performing fine-tuning operations and should really be performed once the entire train system had been integrated. It would be difficult to take such measurements early in the project because the train engine control and sensors had not been developed at the time. These elements would be needed to acquire precise measurement values. Also unknown were the train loads. Therefore, this task will be performed at a later date since it is not essential to basic simulation engine development.

The task of determining simulation engine granularity will also be delayed until after integration. Although this issue directly affects the simulation engine algorithm, the granularity can not be judged until a working system is in place.

2. Development of Forecast Component of Simulation Engine

Introduction

The simulation engine includes a forecast component that is used in collision avoidance. This forecast component acquires similar information from the track manager as the basic simulation engine. It produces a forecast of train locations based on either the current or desired track conditions and returns the result to a global data structure. The control manager reads the data in this global structure to perform collision avoidance.

The forecast component has two main modes of operation. It can predict the future locations of trains based on actual track conditions or based on "mock" track conditions. Mock track conditions are conditions that the UI interface has set. They reflect the desired changes the user wishes to make. Forecasts made using mock conditions are used by the control manager to determine whether to pass the user command to the actual track. Forecasts based on the actual conditions have a different reaction. If a collision is forecasted, the control manager will take necessary action to prevent it. If no collision is forecasted, no action is taken.

This stage has been completed. The component was written in C. Since this component communicates with the track manager, an interface was defined to allow for the acquisition of necessary data. For the output of calculated data, a global data structure used for communicating with the control manager was defined. The data structure is a three dimensional array with the following dimensions: (1) trainid, (2) track occupation, (3) forecast step number. This data structure contains the forecast calculated by the simulation engine. Also, a global variable specifying the time interval size of a step was defined.

The Process

An algorithm outline for this component was developed as an intermediate step to developing the source code. This outline can be found in Appendix C. Once an inspection and walk-through of the algorithm was performed, it was transformed into C source code. The interface to the track manager was defined during the development of the code. When an access to the track manager was required, and a previously defined function did not fulfill the need, a new interface definition was defined. These new interface definitions were sent to the team member responsible for track manager development.

Following normal routines, an inspection and walk-through of the source code eliminated the majority of logical and syntax errors. Once compiled into object code, all syntax errors were removed. A test driver was written to simulate a function call to the forecast component. As well, stubs were written to simulate the various track manager function calls made. The simple circular test track used in the basic simulation engine development was used as the test case as well (see Figure 4). Again, these pieces of code were inspected and followed a walk-through to eliminate errors. Each of these pieces were compiled separately into object code to ensure all syntax errors had been eliminated. Once this was completed, the object code files were linked together to form a simple unit test case. The results of this simple test can be found in Appendix D.

Obstacles

A number of problems occurred during the testing phase of this component. The test output revealed that the forecasted locations were not quite correct. After some time was spent debugging, the problem was found to be due to the inherent nature of floating point numbers which follow the IEEE 754 single-precision floating point number convention. Equality comparisons of floating point numbers created problems. The solution involved artificially converting these numbers to integers before performing comparisons. This solution however, created another problem, although it was a minor one. Rounding errors on the last digit of these artificial integers created a few inaccuracies in the forecast. After some thinking and several attempts at debugging, it was concluded that this problem was very minor. It causes a forecasted location to deviate by one track at most. Fortunately, this deviation did not affect forecasts for subsequent intervals. Due to these conditions, this minor problem can be safely ignored without detrimental effects to the train system.

Changes to Program

This component allows for advanced collision avoidance and detection. In the original proposal, this component provided only simple forecast data. Thus, the complexity of this component has increased to allow for advanced forecasting. This extra complexity did not significantly increase the predicted development time of the component. Although the complexity of the design increased, the actual development of code remained fairly simple. This was perhaps due to careful attention during the design.

3. Development of Short-Circuit Block

Introduction

The short-circuit block development stage produces a working component that hides the fact that the track may be a real or virtual. When a track is real, the only job of this component is to pass the command to the physical track. When a track is virtual, the short-circuit block must emulate the physical track by sending track updates to the track manager. For instance, if a user requests that a train speed should be increased on a virtual track, the short-circuit block would call the track manager update functions to perform the update.

The short-circuit component has been developed and unit tested successfully. Results of the unit testing can be found in Appendix E.

Changes to Program

The development of this component has been incorporated into the development of the interface to command interpreter rather than the simulation engine. Since the simulation engine does not need access to the short-circuit block while the interface to command interpreter requires it, the change increased cohesion within each respective component. The interface to command interpreter determines whether to call the command interpreter or to update the track manager directly, effectively acting as the short-circuit block.

4. Integration of Simulation Engine Subcomponents

The simulation engine subcomponents such as the forecast component and the time operations components have been integrated and tested during the previously mentioned component development stages. Since the various tests have proven to be successful, the subcomponents have been integrated automatically. This stage is complete.

STAGES CURRENTLY IN PROGRESS

5. Development of Interface to Command Interpreter

Introduction

The interface to command interpreter development stage produces a working component responsible for communicating to the command interpreter. This interface component acts as the sole gateway to the hardware. All simulation software outputs to the physical train track must pass through this component. As well, all inputs from the physical train track must also pass through this component. This component uses the interprocess communication (IPC) mechanism to perform communication between the simulation software process and the command interpreter process.

Currently, research is being conducted on the IPC mechanism. A number of useful sources are described in the literature review section. These documents contain a good overview of IPC as well as example source code that will form the basis of the IPC mechanism. A number of simple programs are currently being written to test out the capabilities and performance of the IPC mechanism. Communication with the command interpreter group will be required shortly to discuss the IPC mechanism as well as the protocol to be used between the two components.

Once the research is complete and a few tests conducted to gain confidence in the IPC mechanism, the interface to the command interpreter can be developed. Development of this component will be completed by January 6th.

Potential Obstacles

A number of potential problems with IPC may occur. The simulation software and the command interpreter were designed to execute in separate processes because of performance concerns. The simulation software performs a large amount of computations at a fairly high rate, with multiple threads handling parallel tasks. On the other hand, the command interpreter and low-level drivers perform reads and writes to the serial port. Such a call to the serial port and the actual transfer of any data is relatively slow compared to the rate of processor execution. The time consumed by transferring one character through the serial port may be equivalent to hundreds of computations performed by the simulation software. Normally, a process is blocked during any transfer of data to the serial port. If the command interpreter and low-level driver were to exist in the same process as the simulation software, the entire process would be blocked. Considering the amount of work that could have been accomplished by the simulation software during such a block, performance issues naturally arise. To solve this problem, the command interpreter and low-level drivers will exist in one process while the simulation software will exist in another. Figure 5 depicts this configuration. The two processes will communicate using IPC. Communication using IPC should not cause the simulation software process to block but this situation remains to be tested. This configuration should only cause the command interpreter and low-level driver process to block during serial port accesses, while the simulation software process may continue to perform computations. If these conditions do not hold, another solution must be found quickly. Perhaps at that point, consultation with Professor Stumm would be required.

6. Integration of Simulation Software Components

Introduction

A preliminary integration trial was performed earlier with partially completed components. Integration of all components is currently being conducted, with new components integrated once completed and unit tested. This stage will be completed by January 10th.

The one day preliminary integration trial was conducted on December 29, 1998. The end results were very positive. All developed components worked together in harmony. A very primitive prototype of the software was put together and was working correctly. The interface to UI, control manager, simulation engine, and short-circuit block worked together on a simple test track. The test track used is depicted in Figure 4. The track manager component was not available for testing at the time. The preliminary integration trial required approximately five hours to complete. It was a very positive and encouraging experience because it proved that the pieces work together indeed, and that the simulation software development was progressing. It was the first time we could see a significant portion of the software in operation. It was a sign that the software was designed correctly and was headed in the right direction. Integration of all components should be fairly similar to this "acid" test case.

The Process

The individual components were unit tested separately before the integration date and led to a fairly smooth preliminary integration attempt. A number of minor integration problems occurred, however they were solved fairly quickly. The stubs written for unit testing proved to be useful in testing the integrated software. Initially, the basic simulation engine and forecast component were integrated with the control manager. Next, the short-circuit block was integrated, followed by the interface to UI at the end.

A number of quick tests were performed once the forecast component of the simulation engine was integrated with the control manager. A few simple test cases were created where (1) trains would not collide during the forecast period, (2) trains would collide during the forecast period, and (3) trains would never ever collide. Test cases (2) and (3) were repeated successfully for the interface to UI integration.

Obstacles

A few problems were found in the interface to the forecast component during the preliminary integration. A number of constant definitions were duplicated. It created minor problems in the test cases. For instance, the speed of the trains were defined as constants. When these constants were changed, the forecasted results did not change. This anomally was due to constants that were defined twice in separate header files. This interface problem was quickly and easily solved. Conditional definitions were used along with standard C comments to remind readers of the duplication.

Declaring global variables across multiple source files had to be done quite carefully, otherwise strange results occurred. In particular, mutual exclusion locks appeared to be disfunctional when global variables were declared incorrectly across multiple files. This problem was due to the inherent nature of the C programming language.

During the design of the collision avoidance feature, measurement units for train length were assume to based on the number of tracks. For instance, a train may be five track pieces long. However, the length of the track pieces themselves also vary. This leads to inconsistent units of measurement. To solve this problem, the shortest piece of track was chosen as the unit of measurement. This design decision involved a few changes to the forecast component as well as the control manager. The email message in Appendix F explains this problem and explores a possible solution.

To Do

When all the components have been integrated, a few simple tests will be conducted to ensure the integration is complete. A simple client such as a plain telnet client will be used to perform simple testing of the front end of the completed software. A stub will used to perform simple testing of the back end IPC communication channel.

Issues concerning crossings and lights were raised during discussions at the preliminary integration. We were uncertain as to whether the crossings were activated by software or by hardware. If it should be activated by software, a simple extension to the control manager could easily provide the functionality. The purpose, function, and operation of lights were also questioned. These issues will require discussion with the appropriate hardware teams.

FUTURE STAGES

7. Extensive Simulation Software Testing

This stage encompasses testing the simulation software from all aspects of the train system. Thus, this stage includes unit testing of the simulation software, complete testing of the software side of the train system, and complete system testing. It is incorporated in the productization stage. As faults are found in the simulation software, code fixes will be applied quickly to allow testing to resume. This stage will commence on January 10th and be completed by the end of February.

Once the integration of simulation software components is completed, extensive testing of the simulation software will occur with the development of test scripts and the use of test drivers and test stubs.

8. UI and Command Interpreter Integration and Testing

UI and command interpreter integration and testing will commence on January 10th. A new productization stage has been created to encompass and replace this stage. The new stage will have an new software integration team as well as a new testing team to address this stage. White box testing will be used to ensure 100 % code coverage. This stage is scheduled for completion by the end of January. To be effective at integration, when problems are pinpointed to a specific component, the author of the component can be consulted for a solution.

9. Total Software and Hardware Integration and Testing

Total software and hardware integration and testing is expected to commence on January 15th and complete by the end of February. This stage has also been replaced by the productization phase. Consequently, the newly formed integration team and testing team will handle this task. White box testing will be used to ensure 100 % code coverage. As in the previous stage, to be effective at integration, when problems are pinpointed to a specific component, the author of the component can be consulted for a solution.

10. Fine-Tuning of Simulation Engine

Currently, when a train is slower than predicted, the simulation engine pauses the movement of the train in the computer model. However, when the train is faster than predicted, the computer model is adjusted instantaneously and the train is placed directly in its correct location. These two situations lead to possibly very blocky and jittery movements of the train in the computer model, which is the basis of the UI view of the model. It is unknown at this time whether a smooth algorithm is needed. If deemed necessary, a smooth version of the simulation engine algorithm can be developed by modifying the existing code.

Once the hardware and software is integrated, fine-tuning of the simulation engine may then occur. At that point in time, feedback from the physical track can be observed and adjustments can be made. The task can involve using various ratios to adjust for real world phenomenon. To further improve the simulation engine, speed values and ratios may be adjusted using feedback from the actual track. In such an approach, the longer a train is allowed to run on the track, the more accurate the interpolations will become.

Fine-tuning of the simulation engine may be a concern because of the heuristic nature of the procedure. Possibly due to erratic behavior of physical phenomenon, the simulation engine may only be accurate within a certain degree despite fine-tuning attempts.

11. Marketing

The marketing stage is also encompassed by the productization stage. Productization teams will be formed during this stage to package the entire software system into a marketable product. Besides the integration and test teams mentioned in the previous sections, a documentation team and a productization team will be formed. The documentation team will be responsible for creating a user manual and ensuring internal code documentation is standardized. The productization team will be responsible for creating a web page, an install shield wizard, and handling external packaging. The marketing stage will commence on January 10th and will be complete by the end of February.

TECHNICAL ISSUES IN PRODUCTIZATION

Introduction

This section will examine the underlying technical issues behind productization and is suitable for technically-oriented readers. Internal technical features, such as the limits on the number of train sets, tracks, trains, lights, crossings, and switches are discussed in this section. Accuracy and performance are also discussed. Finally, installation methods are explored.

Underlying Technical Issues

Transforming the simulation software into a marketable product requires the following features to be supported ideally. It must be capable of handling one real physical set and an infinite number of virtual train sets limited only by computer memory space. The number of tracks per train set, the number of trains, switches, lights, and crossings should only be limited by the available memory.

In the C programming language, the use of functions such as calloc(), malloc(), and realloc() allow for this requirement to be met. Large arrays may be allocated dynamically using these functions, once the size of the array is known. Large arrays were the chosen data structure because they allow for very fast access to items. Since performance is a primary concern, this structure was chosen. Other possibilities include using purely dynamic data structures such as linked lists. However, such an organization would not allow for quick indexed accessed. This structure would reduce system performance but make efficient use of memory. However, memory usage is not a concern because the software is relatively small.

Accuracy of the simulation engine is important to productization. The interpolation of train locations should be accurate within a 10 % error, with a confidence level of 80 %. To meet this goal, several parameters will be factored into the predictions. These factors include (1) track type ratios, (2) locomotive type ratios, (3) train load ratios, (4) previously occupied track ratios, and (5) number of trains on track ratios. These parameters provide a heuristic method of bridging the computer simulation model with the actual physical track. From observation of the initial train set in December 1998, it was observed that the train speeds fluctuated due to track types occupied. Also, the type of locomotive influenced the speed as well. Some locomotives ran faster than others during the test runs. Train loads also led to different train speeds as well. From a brief discussion with a fellow electrical engineer, it was realized that the number of trains occupying the tracks and their speeds affected the speed of the each of the other trains. This leads to the interesting possibility that a system of mathematical equations exist for the track, which can be solved to determine the correct speed of all trains. Exploration of this mathematical system will not be conducted due to time constraints.

To be a marketable product, the software must exhibit fast performance. It must not become out of date with the actual track conditions. Although part of this performance may be achieved with fast hardware, the software should also be efficient. Since performance issues have always been a concern, the architecture of the software has been designed with performance in mind. The software architecture includes a mixture of processes and threads to increase execution efficiency. Figure 4 depicts part of the architecture involved with communicating with the command interpreter. The software simulation process contains multiple threads that execute the control manager, track manager, and simulation engine in parallel. As mentioned in the section on the interface to command interpreter, the command interpreter and low-level driver will exist in a separate process to increase performance. Interprocess communication (IPC) will be used by the simulation software to communicate with the command interpreter. This strategic use of processes and threads allow computing resources to be used efficiently and effectively, leading to a general increase in performance.

The use of arrays for fast indexing by-passes the need to use linear searching and increases overall software performance. Although a small amount of memory is sacrificed for such a configuration, the increase in performance justifies the trade off. As mentioned previously, the amount of memory consumed is relatively small.

To allow for easy software installation, the simulation software should ideally consist of one binary executable file, which also incorporates the command interpreter and low level driver. This requirement can be accomplished quite easily with standard features of any C compiler. Installation would simply involve copying the single binary file to a desired directory. No other support binaries should be needed.

Another possibility for installation is to distribute all of the necessary source files in tarred directory. A user would simply need to untar the directory onto their computer and type "make" to cause compilation of the binary executable. This second possibility can be an additional method and targeted towards those who are technically inclined and wish to fully understand the inner workings of the software. Another benefit of this method is that the software would be platform independent to a small degree. The software could be compiled on any Unix platform and possible the Windows platform as well.

The forecast component of the simulation engine should be flexible to allow for a variety of forecasts to be calculated. This has been accomplished by creating parameters for the time interval and the number of time steps desired. With no practical limits placed on these parameters, forecasts can be made at any desired level of granularity and as far ahead into the future as desired. As well, forecasts can be made on actual track conditions or user established mock conditions. These capabilities have resulted in a very flexible forecast engine.

SUMMARY OF CHANGES TO THE PROGRAM

The first major change was the creation of the productization stage which encompassed the previous stages of (1) extensive simulation software testing, (2) UI and command interpreter integration and testing, and (3) total software and hardware integration and testing. Productization teams formed will transform the software system into a marketable product.

The second major change to the simulation software is the addition of advanced collision detection. This feature was added due to the extra help that was available from members of the command interpreter team. This change altered the development of the forecast component.

The short-circuit block was moved from the simulation engine to the interface to command interpreter. Since the simulation engine does not need access to the short-circuit block while the interface to command interpreter requires it, the change made increased cohesion within each respective component.

The determination of various ratios used to ensure accuracy of the simulation engine was moved to a date after the integration of software and hardware has commenced. This task, called the fine-tuning stage, was moved because the ratios are simply fine-tuning adjustments and accurate feedback is required for the adjustments to be useful. Determining simulation engine granularity was also moved for similar reasons.

Appendix A

Simulation Engine Algorithm Outline

Appendix B

Simulation Engine Test Results

Appendix C

Forecast Algorithm Outline

Appendix D

Forecast Algorithm Test Results

Appendix E

Short-Circuit Block Test Results

Appendix F

Train Length Problem