Performance Considerations of Embedded Scripting Languages in Real-Time Training

John Aughey
The Boeing Company
St. Louis, MO
john.h.aughey@boeing.com

ABSTRACT

Because scripting languages provide great flexibility, programmers have begun to use them more frequently within software programs. In the context of training systems, the ability to tailor the software to the needs of the user rather than relying on a static implementation allows for creation of software that facilitates a very agile training curriculum that is easily adaptable to meet the needs of students. As these scripting languages are used more frequently in time-critical applications, such as real-time training devices, it is important to assess their effects on the overall speed and performance of the software. In this paper, I will highlight areas in which scripting languages can assist in providing software that easily adapts to a dynamically changing training environment. I will also discuss strategies for embedding these scripting languages while avoiding negative impacts on real-time performance. Finally, I will analyze and report on the performance of these scripting languages in existing training environments.

ABOUT THE AUTHOR

John Aughey has worked as a software engineer at the Boeing Company for 10 years. He has developed software for Integrated Defense Systems to include visual systems, F/A-18 and F-15 simulator projects, and Internal Research and Development projects. John is a co-inventor on two visual-systems related patents. John received his Bachelors degree in Computer Science from Purdue University in 1996 and a Masters degree in Computer Science from Washington University in St. Louis in 2001.

Performance Considerations of Embedded Scripting Languages in Real-Time Training

John Aughey
The Boeing Company
St. Louis, MO
john.h.aughey@boeing.com

INTRODUCTION

As the performance and capabilities of scripting languages have improved, the gap between them and complied languages has narrowed. Improvements in computational technology, such as faster clock speeds and multiple processing cores enhance the usefulness of scripting languages within larger applications. Real-time training applications in particular can benefit immensely from the use of embedding scripting languages, not only for the initial development, but also throughout the entire lifecycle of the product. This paper will discuss how scripting languages can benefit real-time training environments taking into consideration the real-time deadlines and other requirements of these environments.

DEFINITION OF A SCRIPTING LANGUAGE

Scripting languages are computer programming languages that are interpreted at the time the program is executed. In contrast to a compiled language where the application is written once and "compiled" to a machine readable form, a scripting language remains in a form that can be read, and more importantly, modified to meet the immediate needs of the user. This has the benefit of allowing the end user to change the behavior of the program, and customize how the application behaves in different situations. Repetitive tasks can be automated with a script, and tedious calculations can be written as modules and reused throughout the product.

Scripting languages are used in a wide range of applications. Web browsers use the JavaScript scripting language to embed logic in web pages which enable highly interactive web pages such as Google(tm) Maps. Excel allows VBScript or JavaScript scripts to be written within spreadsheets to perform automation tasks that considerably reduce the user's workload. A growing number of commercial applications embed some type of scripting language in their products.

Benefits of Scripting Languages

In the domain of real-time training systems, scripting becomes even more important. As the complexity of the training devices increase due to intra-force and inter-service interaction, the level of control required by instructors in order to effectively train also increases. The setup of a large scenario in these environments requires an attention to detail that is difficult to obtain without the help of automation. Scripting languages can aid in that automation. During the actual training exercise, many forces need to be monitored and controlled, often at a level of detail that could overwhelm an instructor. A monitoring scripting language in this case could reduce the workload of instructors allowing them to concentrate less on managing the simulation, and more on training the war fighter.

In addition, much of the cost of a training system is the development and testing efforts. Scripting languages in these types of environments can be invaluable to an engineer. Scripting languages can be used as an embedded debugging tool that interacts dynamically with the running simulation giving the engineer much finer control that can be given using traditional debugging tools. In a testing environment, scripting languages can be used as an automation tool enabling tests to be executed much faster and more repeatable than could be obtainable if the tests were run manually.

Embedded scripting languages are designed to be helpful for solving problems in specific domains. Quite often, general purpose scripting languages can be used and extended, eliminating the need to design a special-purpose language. Languages such as Ruby, Python, LUA, and Perl all can be embedded into a host environment and extended to meet the needs of your particular problem domain.

Reasons to use Scripting Languages

The primary reason to have a scripting language as an integral part of a product is to allow the details to be

determined at the last possible moment and by the most appropriate person. The details of an application change and evolve as time goes on. It's extremely difficult to know how a particular application is going to be used until it has been put into service. During development, the design team can make educated guesses as to the level of control and interactivity that will be needed, but it is not until end users actually interact with the system as a whole that anyone can understand what is really needed. There is an 80-20 guideline that says details will change 80% of the time while the implementation, that is, the basic device functionality, e.g. the flight dynamics of the simulated aircraft, changes 20% of the time. The goal with using an embedded scripting language is to push as much of that 80% that changes out into the scripting language where change is easier and allow the 20% that remains to be the core compiled system.

A very large percentage of development costs are dedicated to maintenance. The initial implementation of a product is relatively inexpensive compared to the ongoing maintenance costs. This again points to the 80-20 guideline where the maintenance is focused on the details of the product. If the maintenance costs of an application can be reduced through the use of scripting languages, then the overall cost of the device can be reduced.

Robert Glass wrote in “Facts and Fallacies of Software Engineering” that maintenance typically consumes 40 to 80 percent (average, 60 percent) of software costs. This means that maintenance is a very costly part of the life cycle of a product. Similarly, Andrew Glover wrote, "If the lion’s share of money is spent after the initial application is built, it seems to me that the criteria for success then isn’t how quickly a high quality application was delivered to a customer. A more logical standard for success, it seems, would be that an application was indeed delivered quickly, but that it also is relative inexpensive to enhance and therefore will prove to be an economically friendly long-lived application."

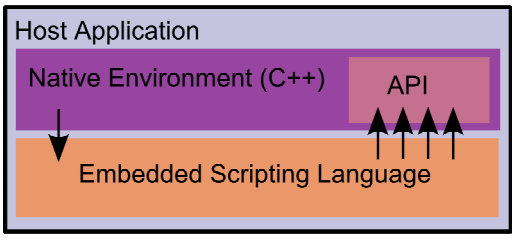

Figure 1. Scripting Environment

How a scripting language is used

Figure 1 illustrates a scripting language embedded in a native C++ environment. In this situation, the host application exposes an Application Programming Interface (API) that allows the scripting language to interact with the simulation. This API could be as simple as "get" and "set" methods for key variables, or more complex and interactive such as methods that allow objects to be created and modified dynamically.

There are tools such as SWIG (Simplified Wrapper and Interface Generator) that aid in exposing a C or C++ API to a scripting language. SWIG can interface a C or C++ API with many different scripting languages such as Perl, Python, Ruby, and Tcl. This both reduces the engineering time it takes to embed a scripting language into an existing product, and allows flexibility in the choice of a scripting language.

APPLICATION DOMAIN

A simulation application typically operates in a frame-based mode. In a frame-based application, the simulation runs a set of calculations to determine what state the simulation should be after a given time difference. For example, a system that simulates a projectile will know the current velocity and acceleration, along with the forces acting on the projectile. At time t, the projectile is at a point in space, and the simulation will perform the necessary calculations to determine where the projectile should be at time t+dt where dt is the time between frames.

For a simulation to be real-time means the calculations are performed in sync with the wall clock. In other words, when ten seconds pass in the real world, ten seconds has also passed in the simulation. A frame-based simulation does not necessarily have to be real-time. A simulation of a nuclear blast, for example, requires the evaluation of equations so complex that it takes longer to compute than current computational systems can do in real-time. Frame-based simulations can also be run in a faster-than-real-time mode similar to fast-forwarding a movie. For the purposes of this paper, I will use applications that run in real-time with respect to the wall clock as a baseline, but the concepts also can be applied directly to slower and faster-than-real-time applications or even event-driven user interface applications.

Quality of Real-timeness

Real-time applications generally fall into two catego-

ries; hard and soft real-time systems. A hard real-time system is one that has strict deadlines that must be met within a specified period of time. The traditional examples of hard real-time systems are pacemakers and engine control systems. These systems perform controls that absolutely must happen. If a hard real-time system does not complete its calculations in time, the system will fail. Soft real-time systems, on the other hand, have an allowable tolerance in how long it takes to complete its calculations. The system may have a preferred time when the calculations should complete, but it can compensate if the system is late (or early) with some degradation of quality. An example of a degraded soft real-time system is an Internet video player that stutters due to network delays or other computer activity.

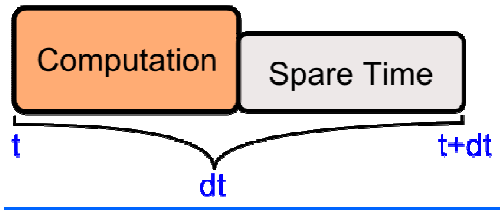

Figure 2. Hard real-time system

Hard real-time systems almost always use a fixed delta time. This delta time is chosen by the level of fidelity required by the simulation. In our projectile simulation example, a very small delta time will result in a much smoother projectile curve than if a coarser delta time was chosen. In hard real-time systems, the amount of computation that is required every frame is bounded. The computation is expected to complete before the end of the frame, so there is a period of spare time until the next frame begins (figure 2). So given a bounded computational frame and a required delta time, the computational system can be selected such that the computation can be completed within the required time.

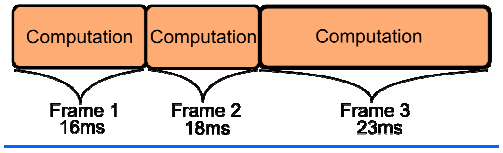

Figure 3. Soft real-time system

Soft real-time systems, on the other hand, often use a variable delta time. The frame time is allowed to change depending on the available computational resources. In the example of the Internet video player, during normal operation the video will play smoothly, but if there are additional demands on the computer such as loading another application or saving to the disk, the video player will skip frames or adjust the frame time to compensate for the lack of necessary computer resources. In a training application, a visualization computer may have to degrade performance in areas that have extremely high scene content; in this case, the perceivable degradation is similar to the video player example. Soft real-time systems may also eliminate the spare time after computation and allow the simulation to run "as fast as possible." In this mode, the real-time computation may be variable based on the current CPU demands and the program will adapt its computation to provide an acceptable simulation. Many commercial games operate in this mode. In summary, a soft real-time system able to adapt to a variable operating environment.

THE PROBLEM

The problem with scripting languages, in real-time environments, is twofold. First, scripting languages are inherently slow. Compared to a compiled language, a scripting language can be many orders of magnitude slower when performing the same set of

Language	Time	Relative Speed
C gcc-4.0.1	0.05 seconds	1.00 x
ocaml compiled 3.09.2	0.05 seconds	1.00 x
SBCL 1.0.2	0.13 seconds	2.55 x
Java 1.4.2	0.40 seconds	8.00 x
Lua 5.1	1.50 seconds	30.00 x
ocaml bytecode 3.09.2	3.76 seconds	75.15 x
Python 2.5.1	9.99 seconds	199.80 x
Perl 5.8.6	12.37 seconds	247.34 x
TCL 8.4	16.00 seconds	320.00 x
Perl 5.8.6	21.75 seconds	435.00 x
PHP 5.1.4	23.12 seconds	462.40 x
Javascript v1.6	31.06 seconds	621.27 x
Ruby 1.8.4	34.31 seconds	686.18 x
Emacs Lisp	47.25 seconds	945.00 x
Applescript	71.75 seconds	1435.00 x
Io 20070410	85.26 seconds	1705.13 x

tasks. Table 1 shows how several popular scripting languages compare against a compiled language.

This particular stacking was a fractal calculation test done by Erik Wrenholt. The point of this table is not to demonstrate absolute speeds of different languages, but to demonstrate how much slower an interpreted scripting language is compared to a compiled languages. Techniques such as just-in-time compilers can bring the performance of scripting languages closer to that of compiled languages, but in general they are going to be slower than their compiled counterparts.

The second problem with scripting languages is that they can add a level of non-determinism to the application. In a real-time environment, we noted that the computational time is fixed. All of the computation for a given frame must fit within this time window. If the work done inside a script takes an indeterminate amount of time, then the overall time may exceed the allowable frame time. This problem could be generalized to any block of work that is non-deterministic that needs to be run in a real-time environment.

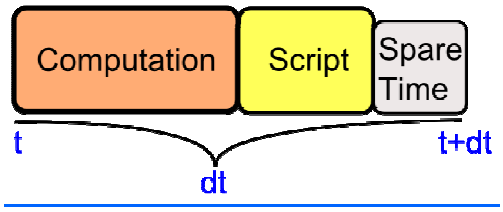

Figure 4. Embedded Script

For the rest of this paper I will address these two core problems, provide some solutions for how these problems can be overcome, and introduce additional ways that scripting languages can be used within real-time training applications without adversely impacting real-time performance.

APPROACH TO THE PROBLEM

To illustrate the problem, I am going to use a flight simulation application that is typical of a real-time training system. For the purposes of this illustration the training system is written initially in a compiled language such as C++. I'll demonstrate how large portions of the software can be refactored into a scripting language, and as a result, can provide a level of flexibility and control that is difficult to obtain in a statically compiled language.

Aircraft Simulator Analysis

I'll focus on one of the more critical components - the aircraft simulator. The aircraft simulator would fit into the category of a hard real-time system. It is not a hard real-time system in the sense that the system will fail if it does not meet the real-time constraints, but rather that the time between computation frames is fixed. This fixed frame computational time is often a side-effect of interacting with external physical devices such as motion seats or video displays, since these external devices require data to be sent at fixed intervals. The simulation of the aircraft is also a core component of a training device, so we want that simulation to be as precise and smooth as possible.

In general, the aircraft simulator will operate in the phases shown in figure 5.

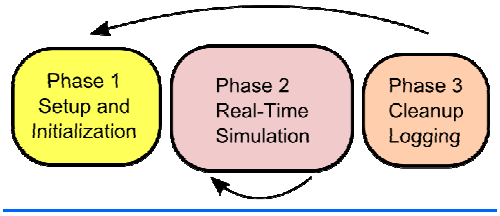

Figure 5. Simulation Phases

The first phase sets up the initial conditions of the training scenario. In this phase the stores are loaded on the aircraft, fuel level established, weather conditions set, ground-truth tables loaded, and so on. These initial conditions were created by the operator or instructor prior to the exercise. This phase may take a relatively long time to complete. Even though it is part of the real-time software system, this phase does not have real-time time constraints. This is akin to booting up a computer or starting a complex application that takes considerable time to load.

The second phase is the main processing loop. It is during this phase that the system is running a fixed frame time simulation as in figure 2. During each frame time, the system computes equations of motion along with calculating additional aircraft avionics needed to simulate flight. The computational requirements have been determined ahead of time to ensure that they will complete before the frame is over. Results of these calculations are then sent to the various external components to provide a visual representation of the environment and provide motion cueing in the seat.

There is, however, a degree of interactivity required during this critical real-time phase of the simulation. The instructor/operator will need to interact with the simulation in order to provide the necessary training to the student. This may be as simple as inducing a failure on the aircraft, or it could be more complex such as creating additional threats dynamically or reconfiguring the aircraft to fly in new locations or environments. These dynamic requirements throw a proverbial monkey wrench into the well-established real-time system. Additionally, it adds a great deal of complexity to the application and this complexity makes up a large portion of the 80% details guideline.

The final phase is the tear-down and cleanup phase. This phase is very similar to the setup phase in that it is outside the real-time processing loop. It is during this phase that the simulation is reset into a condition where a new training scenario can be loaded and executed.

SCRIPTING OPPORTUNITIES IN FIRST PHASE

Typically, the setup of a simulation is done by reading a set of initial conditions from some sort of data source. Often this is a set of configuration files, though it could be transmitted over the network via some established protocol. An offline tool often creates the scenario and writes out this set of configuration files that can be parsed by the simulator. Sometimes an established network protocol between the operator computer and the simulator is used to pass the initial conditions to the simulator. No matter what mechanism is used, some configuration system is defined and written into the application. Depending on complexity the system, the parsing of this configuration data and the setup of the simulation can be a very difficult task. Even if a standard format such as XML is used, the amount of code dedicated to configuration and setup can be large.

A portion of an example configuration source code along with its corresponding configuration file is shown in figure 6.

This pattern would be repeated for all of the various components of the aircraft that can be configured.

There are several concerns with parsing a configuration file and setting up the initial state in this way. First, even if a utility library such as TinyXML is used to parse the configuration file, the source code required to interpret the data inside the file can become overwhelming. When a new configurable attribute is created, the configuration source must be modified to add this new parameter. This can be a relatively small change for simple cases like in this example, however, there are often more significant interactions between other components in the system, making this a much more difficult task.

The solution to this complexity is to expose the configurable parameters as an API to the scripting language and allow the scripting language itself to setup the system. In the example above, the simulation would expose the API function **setFuelLevel** to the scripting language. Instead of parsing a configuration file, the simulation would simply execute a script. To set up the simulation as we had done in the previous example, the script would look like:

```
aircraft.setFuelLevel(LEFT,75.0)
aircraft.setFuelLevel(RIGHT,85.0)
```

As you can see, we've completely eliminated the need

```
if(config->hasAttribute("left_fuel_level")) {
  aircraft->setFuelLevel(LEFT,config->getAttributeAsDouble
("left_fuel_level"));
}
if(config->hasAttribute("right_fuel_level")) {
  aircraft->setFuelLevel(LEFT,config->getAttributeAsDouble
("right_fuel_level"));
}

<aircraft>
  <left_fuel_level>75.0</left_fuel_level>
  <right_fuel_level>85.0</right_fuel_level>
</aircraft>
```

Figure 6

for configuration file parsing code. We can simply set the attributes directly from the executable scripting language. This example is very simple, but suppose we wanted to set the right fuel level to be 1/2 the level of the left. It could be written like:

```
aircraft.setFuelLevel(LEFT,75.0)
aircraft.setFuelLevel(RIGHT,37.5)
```

But that is not very expressive. It is functionally accurate, but it doesn't convey any meaning. To convey additional meaning, the configuration script could be written as shown in figure 7 example 1.

This script now conveys more meaning. You can tell by reading this that the right fuel level is supposed to be half the left. An even more expressive configuration is seen in figure 7 example 2.

Now we're getting somewhere. It is very clear that the fuel levels are specifically being set to some off-balance ratio. And this ratio is 1 to 2. Though this can read like a traditional configuration file, the configuration has the leverage of an executable language. If we use features of the scripting language that allow us to extend the API, we can create additional methods to make the configuration easier to read. After adding some helper methods, the configuration reads as figure 7 example 3.

Now, not only is the configuration very expressive, but it is easier to read (and write). The argument could be made that if the right tools exist, you never have to read or write a configuration file. But these *tools* would still have to be written and these tools would be complex themselves. This expressive configuration language is available for developing those tools too, and it allows for the system to be extended to meet future requirements. For example, if the requirements change where the attributes are now stored in an Oracle relational database, the system can still configure itself by writing it as seen in figure 7 example 4.

Scripting languages themselves can be extended through the use of add-on library modules. The database module for the above scripting language example came from a freely available database management binding. As you can see from the example, we were able to create a very expressive configuration system that evolved and was extended as the needs and requirements changed, without modifying the original simulation program.

SCRIPTING OPPORTUNITIES IN SECOND PHASE

The second phase of the application, the actual real-time simulation, can be more complicated because this is the part of the program that has real-time deadlines. For purposes of this discussion, we'll assume the real-time portion runs on a 50Hz clock where each computation frame lasts 20ms. A typical computation frame may look like figure 8.

```
1)    aircraft.setFuelLevel(LEFT,75.0)
      aircraft.setFuelLevel(RIGHT,aircraft.getFuelLevel(LEFT) / 2.0)

2)    right_to_left_ratio = 1.0/2.0
      aircraft.setFuelLevel(LEFT,75.0)
      aircraft.setFuelLevel(RIGHT,aircraft.getFuelLevel(LEFT) *
      right_to_left_ratio)

3)    right_to_left_ratio = 1.0/2.0
      aircraft.left_fuel_level = 75.0
      aircraft.right_fuel_level = aircraft.left_fuel_level *
      right_to_left_ratio

4)    aircraft.left_fuel_level = database.execute("select left_fuel_level
   from prameters where exercise_id=3").value
      aircraft.right_fuel_level = database.execute("select right_fuel_level
   from parameters where exercise_id=3").value
```

Figure 7. Code Examples

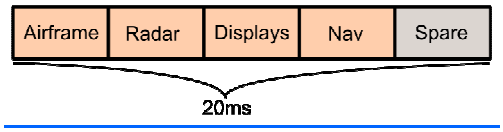

Figure 8. Real-time Simulation

In this example, each component will take some time to complete. The system is designed so that in any given frame, if you sum up the amount of time each component took to execute, it will not exceed the available frame time; in this case, that frame time is 20ms.

There are several possibilities for embedding scripting languages in this critical environment. The first possibility is to include a scripting component directly in the critical path. In this case, the scripting language is executed in the same computational pipeline along with all of the other components. In figure 9, the scripting language would consume part of the spare time at the end of the computation cycle.

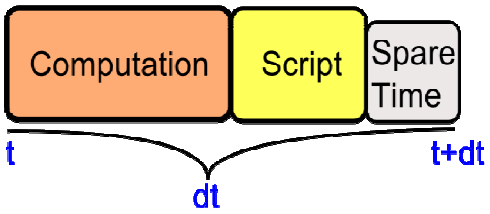

Figure 9. Embedded Script

When the embedded scripting language is part of the critical path of the real-time program, the execution time of that script must be considered carefully. If the execution time of the scripting component were to exceed the available time, it would violate the real-time performance restrictions and degrade the system performance. For scripts that operate in this critical path, we need to consider both the performance of the scripting language, and the determinism of the code running inside the script.

Speed of scripting languages

We have already seen that scripting languages are many order of magnitude slower than compiled languages. So it is clear that the the bulk of the real-time portion of a simulation would still be written in a compiled language such as C++. In many cases it is beneficial, however, to have a scripting language available in order to perform some task-specific job. So when looking at the speed of scripting language, we must put into perspective the amount of work that is actually going to be done inside the scripting language. You should expect that the tasks done inside a scripting language are not going to be very computationally expensive. Tasks that are good candidates to be handled by a scripting language are simple logic checks, data collection, performance validation, etc. An example of such a task might be:

```
if aircraft.altitude > 10000.feet
  aircraft.right_engine.flame_out
end
```

This example task might be written as part of an instructional program that will induce an engine failure when some conditions are met. In this example, the script will cause the right engine to flame out when the aircraft passes 10,000 feet.

To test the speed of a very simple task such as this, I wrote a test environment that includes an embedded scripting language inside a host C++ application. The host C++ application executes the routine in the example by calling into the scripting language 4 million times. The test computer is an 2.8GHz Intel Xeon running a Linux 2.4 SMP kernel. The gcc compiler is version 3.2.2 and the scripting language chosen for this test is Ruby 1.8.5.

The test application was able to call the scripting language 4 million times in 3.05 seconds. This means each call into the script lasted only .000000762 seconds or 762 nanoseconds. When the logic task above was written in a compiled language, each call to the example routine only took 5 nanoseconds or (was 152 times faster). But when you put the total time into perspective, the scripting language still only took up 4/1000ths of 1 percent of the available frame time.

In fact, the overhead of simply calling into the Ruby scripting language was over 60 percent of the total time. This time was measured by calling into a function that did no work. In this test, the overhead measured was 505 nanoseconds to just switch between the C++ environment to the Ruby scripting language environment and back again. So if you subtract out the overhead costs of switching environments, the simple logic statement only took 257 nanoseconds to run. In a 20ms frame, over 77,000 logic statements similar to this example could run.

These series of tests could be run against a wide range of available scripting languages such as Python, Lua, ECMAScript, and others, but the point of this is not to quantify and compare the speeds of different languages, but to put into perspective the type of work that could be delegated to a scripting language and to show that in spite of the inherent slowness of scripting languages, a lot of work can happen inside of scripting languages in a very short period of time.

Determinism of scripting language tasks

The example above was very simple and the time it takes to run is deterministic, that is, it takes the same amount of time to execute each time it is called. Even a more complicated example with many decision branches can still execute in a deterministic way. The problem lies in what happens when the scripting task needs to perform work that may take an unknown amount of time to complete. Non-deterministic tasks could be as simple as reading and writing data to a disk, or could be more complicated such as accessing a remote database. How can we allow these non-deterministic tasks to be run inside of a deterministic real-time system? The answer is, it depends.

First, let's look at what is going on when this non-deterministic task is run and what the consequences of this task slowing down the real-time system and causing over-frame conditions might be. There are many times when some degradation of system performance is acceptable. Consider the case where an embedded scripting language is used for automated system testing. In this environment, the script may have total control over the simulation and may validate the behavior of the simulation against a known set of conditions. After each test completes, the script may log the results to a database. This logging activity may take a relatively long time to complete and degrade the real-time system during that period. But in this environment, it doesn't matter if the system degrades during logging. At that point the test has either passed or failed and what happens while it is logging is of little concern.

Another condition where simulation degradation due to scripting overhead is acceptable is during time periods where the simulation is changing state in an unnatural way. This may happen as a result of the instructor reloading a mission that places new threats or weapons stores into the environment. For situations like this, it is the exceptional case where new aircraft appear in space or new weapons are loaded on the wings, so a recoverable degradation in the real-time simulation is perfectly acceptable.

But there are often cases where the task being performed by the scripting language will take longer than is acceptable to be done in the critical path. This could be due to interaction with non-deterministic external systems, or it could just be a very complex computation task. For these situations, an acceptable solution is to run the scripting environment in a separate thread.

Processor clock speeds have increased dramatically over time. Today, the emphasis is not on faster clock speeds, but on including multiple processing cores into a single machine. Most computer systems built today have at least two processing cores on a single CPU and are designated as dual-core or multi-core. Computers with eight processors and 16 cores are available commercially and are very affordable. To take advantage of these multiple processing units, applications can employ techniques such as threading to improve performance.

In multi-threaded applications, multiple execution "threads" are running at the same time. On a single core machine, the threads are time sliced, but on multi-core machines the threads can execute simultaneously. This introduces the problem of synchronizing access to shared resources, but it also opens up the possibility of doing more than one task at the same time.

If the scripting language were split out into a separate thread, then on a multi-core machine the real-time task and the scripting task could execute at the same time. If the scripting task needed a long time to complete, it would not impact the real-time requirements of the real-time task. There might still be some interaction between the two threads for accessing shared resources and general synchronization, but the real-time impacts would be minimized.

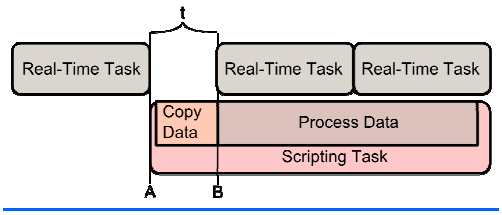

Figure 10. Synchronized Access

In the figure above, the scripting thread requires some data of the real-time task, but the operations on that

data will take a long time to complete. It's critical that the data the scripting task gathers is coherent, so the two threads must synchronize themselves. At point A, the two tasks synchronize. During time t, the scripting task extracts data from the simulation while the real-time task waits. At point B, the scripting task has gathered all the data it needs and signals the real-time task. After point B, the real-time task can continue on with its processing while the scripting task performs its own tasks in parallel. The scripting task may take a long time to run, but that is not a problem because the real-time task is allowed to run without needing to wait for the scripting task to complete. It is not until the scripting task needs additional data that it asks to re-synchronize with the real-time task.

There are a lot of situations where running in this type of mode is acceptable. Consider the original example we used where the script was monitoring the altitude in order to induce a failure. While this specific example does not take a long time to run, assume that the task were more complex that required additional time to complete. The fact that new data is not gathered every single frame does not seriously impact the goals of this script. What matters for this script is that a failure is induced at 10,000 feet and if the failure happens a few milliseconds after the aircraft reaches that altitude the end result is the same.

SCRIPTING OPPORTUNITIES IN THIRD PHASE

The third phase is very similar to the first phase in that it does not have any real-time constraints. This phase is another prime candidate for scripting because of the nature of the tasks that typically need be done. It can be difficult to know what needs to be done after a simulation exercise until after the simulator has been put into operation. The system might just shut down and prepare for another exercise. Or it may do some more complicated tasks such as logging the exercise results to a student training database. The need for additional logging or data stores may come long after the initial deployment, so a flexible scripting environment can enable just-in-time enhancements to be created with little or no impact on the original system.

Conclusion

Scripting languages have clear benefits for development of real-time training applications. Through careful programming, these scripting languages can be embedded in a training application, not only for setup and teardown phases, but for the real-time processing phase too. As training systems become more and more complicated, the need to integrate systems built by multiple companies factors significantly into the cost of building a device. When scripting languages are integrated into training environments, the training curriculum becomes more adaptable and customizable to the needs of the crew. Embedding scripting languages not only improves the development of the device, but, more importantly, benefits our men and women who rely on the best training we can offer.

REFERENCES

John K. Ousterhout. 1998. Scripting: Higher Level Programming for the 21st Century. IEEE Computer Magazine, March 1998.

Andrew Glover. The inconvenient truth of software development. http://www.testearly.com/2007/04/30/the-inconvenient-truth-of-software-development/

Erik Wrenholt. 2007. Ruby, Io, PHP, Python, Lua, Java, Perl, Applescript, TCL, ELisp, Javascript, OCaml, Ghostscript, and C Fractal Benchmark. http://www.timestretch.com/FractalBenchmark.html

John Aughey. 2006. Extending Training Capability Through the Use of Embedded Domain-Specific Languages. I/ITSEC Proceedings 2006.