Toward a Rapid Prototyping Environment for Character Behavior

Ian Horswill

Northwestern University
2133 Sheridan Road, Evanston, IL 60208
ian@northwestern.edu

Abstract

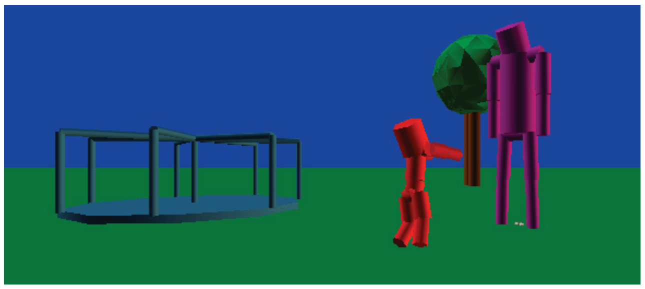
I describe work in progress on a system for interactive prototyping of AI-based characters. A sort of "Sims construction set", the system combines a simple physics simulation with a set of domain-specific languages to allow programmers to quickly build and test character AI. It allows iterative, incremental development in which behaviors can be compactly authored, tested, monitored, and hot-swapped for new behaviors, using in-game editing and debugging facilities.

Motivation

Building AI-based virtual characters is hard. Building a system like *Façade* (Mateas and Stern 2005) from scratch requires enormous amounts of both knowledge and programmer time. In principle, one should be able to leverage existing tools such as commercial game engines, or research animation systems. However, in practice these systems, while very useful for certain genres and applications, are often difficult for non-specialists to adapt to other uses.

A number of libraries have been developed for interfacing research AI systems to commercial game engines, ranging from early work with Quake Soar (Laird and van Lent 1999) to more recent work such as Microsoft's .NET interface for UT (Sterland, Lissiak et al. 2007) and Pogamut (Gemrot, Kadlec et al. 2009). These systems typically implement some kind of remote procedure call interface in which action requests by the AI system are serialized over a TCP connection and responses and environment data are returned by serializing over the same socket. This approach can be extremely valuable; however it has important limitations that reduce its flexibility.

First, the user is limited to the character actions provided by the game engine, which tend to center on running, jumping shooting. Pogamut, for example, is recommended by its authors only for first-person shooter genres (Gemrot et al, p. 13). Actions like hugging another character,



carrying a child, or lying on a sofa are typically not supported. They could be added by importing new model and animation assets, however this requires considerable time and money, as well as access to facilities (e.g. mocap studios) and people (skilled animators, actors), which often aren't available at research institutions.

One could also attempt to incorporate research tools for procedural animation (aka motion synthesis) into existing game engines. Although this would be a daunting task, many procedural animation systems have been developed, ranging from early kinematic systems such as (Badler, Phillips et al. 1993) and (Perlin and Goldberg 1996), to more recent physically-based methods such as (Treuille, Lee et al. 2007; Levine, Lee et al. 2011; Muico, Popović et al. 2011). Although there has been steady progress, most work has focused on building high-quality controllers for isolated actions (usually locomotion), often assuming the availability of preexisting mocap data, and generally assuming characters don't interact physically with other objects other than as disturbances to be recovered from. There is simply a dearth of systems that allow easy implementation of simple human actions such as picking up a glass, writing on a clipboard, hugging, or shaking hands (Hecker 2010).

So the animation subsystems of contemporary systems severely constrain the set of actions an AI researcher can ask their characters perform. I believe we need to stop thinking of this as an animation problem and start thinking of it as an AI problem. Character actions are just that: *actions*. That they happen to be implemented in contemporary game engines by sequencing and blending fixed animation clips is simply an implementation technology, and a suboptimal one at that. As we seek develop more flexible characters that need to be able to perform larger repertoires of actions, and more expressive characters in whom bodily movements and postures are continually modulated by the character's motivational and affective state, we will inevitably need to consider the control of limbs, posture, and the face to be AI problems, or at least robotics problems.

The second problem with interfacing AI systems to game engines via RPC is speed. Control rates typically range from 4 to 10 Hz, which means the round-trip latency between the engine and the AI system (i.e. the time between something happening and the user seeing some sign of a response) will be a minimum of one RPC cycle (100-250ms) and more likely two cycles (200-500ms). This is more than sufficient for a high-level strategic AI operating over large time scales; however, processes operating on faster time scales, such as a character flinching when touched by someone they don't like, do not have time to run over the RPC connection and so have to be coded on the engine side.

This brings us to the last issue with RPC-style systems, which is that they impose a hard division between the "AI part" and the "engine part," where the latter is relatively fixed. Adding new sensory or motor capabilities, a new camera controller, or anything else that needs to run fast, requires the user to bring the engine down, edit and recompile the relevant parts of the engine or the interface library, and restart the whole system. We want these things to be scriptable and debuggable in-game.

In this paper, I will discuss work in progress on *Twig*, an open-source prototyping environment for research in character AI. It allows researchers (or students) with little or no knowledge of graphics or game programming to rapidly construct a wide range of character behavior, from high-level reasoning down to low-level sensory-motor control. Due to space constraints, this paper will focus on the overall design and goals of the system, and on the authoring system for bodily behaviors, since these are the most unusual aspects of the system.

Design Goals

The system was designed with a number of technical goals in mind:

- Cartoon **believability**; true realism is beyond the abilities of the system.
- Ease of **interface to AI systems.**
- Ease of **adding new behaviors** and objects.
- Support for behaviors involving **sustained contact between characters**.
- Continuous **tuning and blending** of behaviors.

In addition, there were a set of *social goals*:

- Ability to use and extend the system **without access to professional animators**
- Usable **without having to learn** graphics, game programming, quaternions, etc.

To support these goals, the system adopts the following design principles:

- **Everything is scriptable**. As much as possible is interactively scriptable using an in-game IDE.
- **Everything is debuggable**. Scripting is fully interactive. Behaviors can be hot-swapped in running characters, and debugged using in-game debugging displays.
- **Everything is a robot**. Characters are powered ragdolls; low-level motion is driven by classical behavior-based robot controllers (Arkin 1998) rather than animation clips.
- **Everything is IK**. Programmers need never think about joint angles or revolute coordinate systems;

High Level Design

The system is written in C# and built on the Microsoft XNA platform. We will focus here on its scripting, and to a lesser extent, simulation facilities, since they are most relevant.

Overview of Scripting Facilities

The primary scripting language for the system is Scheme, a derivative of Lisp. Scheme was chosen in part because it's simple to implement, and in part because it makes it easy to construct embedded languages (domain-specific languages implemented within Scheme itself). It is these DSLs that do most of the run-time work for the Scripts. Scheme is largely used as a front-end for implementing and accessing the DSLs. We have also recently added a relatively simple Prolog implementation, since it allows a fair amount of existing code, e.g. for planning and natural language to be downloaded and run directly. The use of Scheme and Prolog is not particularly important to the discussion here; any language that provides good support for metaprogramming (e.g. a Turning-complete macro system) could be used instead. See Norvig (1991) for an excellent discussion of the implementation of Scheme and Prolog, as well as their use in AI programming; see also Thompson's Yield Prolog (2010) for a particularly elegant implementation of Prolog within languages such as C# and Python.

Each character or other simulation object has its own namespace to prevent cross-talk between scripts for different characters. Code can be run within a given character's namespace using the Scheme form `(within` *object …code…*`)`. Each character also has a blackboard structure (Isla and Blumberg 2002) used principally to implement optional adverbs for actions, such as telling a ball which direction a character would like it to move in when kicked.

The Scheme and Prolog implementations emphasize power over speed; although they are fast enough for event handlers, they are have not been extensively optimized and should not be used for code that runs frequently (e.g. on every frame). High performance code is supported through special-purpose DSLs embedded in Scheme, but which bottom-out in custom interpreters written in C#.

The Signal Language (GRL)

The most important DSL is a language for describing real-time signals based on the GRL robotics language (Horswill 2000). GRL programs define a graph of signals whose current values are updated automatically. A GRL expression such as:

```
(define-signal a
  (+ b (low-pass-filter c 1.5)))
```

tells the system that it is responsible for ensuring that whenever one asks for the value of a, it should hold the sum of the values of b and a low-pass filtered version of c. How the system does this is up to the implementation; the original GRL compiler did elaborate analysis and optimization. However the current version simply does type inference and constructs a graph of signal objects that are then topologically sorted and updated in round-robot fashion on each clock tick; this has been more than fast enough for the time being.

GRL has the standard advantages of strongly-typed functional languages; it allows higher-order, polymorphic procedures to be used, but these are resolved at compile/load time, and transformed into a form that can be run without any dynamic storage allocation or run-time type checks. It was originally introduced to allow easy scripting of sensory and motor systems. However, its use has extended to other parts of the system where script code needs to be run at video rate, for example for camera control, or in condition checks for state machines or behavior trees that cannot be handled through event-driven messaging.

Other Domain-Specific Languages

The system also contains DSLs for common game- and research-AI tasks. A **state machine language** based on Rabin's work (2002) and a **behavior tree** language based on Chris Hecker's implementation from *Spore* (2008) are available as embedded languages within Scheme. Since they are straight reimplementations of Rabin and Hecker's work, I will not discuss them here, but instead refer the reader to the original authors. State machines and behavior trees can call code in the other languages; for example, they can use signals as event triggers and can explicitly start, stop, and pass arguments to motor behaviors written using the signal language. They can also specify arbitrary Scheme code to run upon entry to or exit from a given state or behavior tree node.

The Prolog system also supports the standard Prolog notation for **definite-clause grammars** for natural language parsing and generation. Again, this is a textbook DCG implementation, so the reader is directed to Norvig (1991) for a good introduction both to their implementation and their use.

Finally, we are working on a simple rendering DSL, **renderkun**, for scripting passes and material/effect mapping in the renderer. The intent is that this will make it easy to programmatically switch between, different styles of rendering, such as cel versus Gouraud shading, or even deferred shading, and experiment with different renderers interactively.

Physics and Motion Control

Twig began as a procedural animation system (Horswill 2009) that has been repeated extended to make it easier for others to use, until it's now effectively a full-fledged game engine. It allows programmers to control character bodies by applying forces directly to push and pull individual body parts, as if it were a puppet, rather than having to deal with joint torques. Unpowered body parts are moved passively by the Verlet physics solver (Jakobsen 2001), providing a kind of passive inverse-kinematics. This allows body parts to be controlled independently, with minimal risk of a change in one controller destabilizing another.

Development Environment

The system runs as a subprocess of GNU Emacs (Stallman 1981), providing a simple IDE supporting editing, syntax highlighting, and interactive debugging of running character behaviors. Behaviors can be interactively added and removed, monitored (using watch points and visualization tools), and hot-swapped with new versions, all while the character moves about the world and interacts with other characters.

Body Control

Twig character bodies are controlled through low-level interfaces called **effectors**. An effector takes a control signal as input and drives the body part(s) appropriately. For example, the effectors for limbs take a signal specifying forces to apply to the endpoint (hand/foot) of the limb and/or the joint (elbow/knee) and its drives them with those forces.

Control signals are generated by collections of **behaviors**. A behavior consists of the actual control signal together with an activation signal that provides a dynamic estimate of the utility of allowing that behavior to drive the effector rather than some other behavior. **Arbiters** produce aggregate control and activation signals from a set of input behaviors, either by weighted averaging, as with motor schemas (Arkin 1998), or by choosing the highest priority active behavior. Arbiters can use either fixed priorities, as with the Subsumption architecture (Brooks 1986) or dynamic priorities as in behavior nets (Maes 1989).

Current Effector Interfaces

The system is a work in progress. One of the open research questions is what kinds of interfaces would be most congenial for controlling the body in different ways.

The system currently provides support for controlling individual limbs, pairs of limbs, the overall posture of the character (pelvis and shoulder height, relative positioning, and twist), locomotion (what target velocity to send to the gait control system), and object approach (a built-in behavior that drives the locomotion system toward a specified object).

Examples

A scripting system is difficult to evaluate in any meaningful quantitative manner. So at the risk of making the paper look like a tutorial, I will go through two examples of simple behavior systems, both to illustrate the general operation of the system and also to show that relatively simple code, of the sort that an undergraduate might be able to write, can implement useful functionality. Both examples are from the course notes used in an undergraduate course on virtual characters.

Space considerations prohibit a complete explanation of the code, but the following should be sufficient to at least give a sense of what authoring behaviors is like.

Implementing Manual Control

The simplest possible behavior system implements the manual piloting of the character from the keyboard. It consists of three behaviors, each defined by a name, activation (utility) signal, and a control signal.

We start with two posture behaviors that apply torques to the body to turn it left and right, respectively:

```
(define-posture-behavior steer-left
  (posture-force shoulder-yaw: 20
                 pelvis-yaw: 20)
 activation: (trigger-on-key Keys.Left))

(define-posture-behavior steer-right
  (posture-force shoulder-yaw: -20
                 pelvis-yaw: -20))
 activation: (trigger-on-key Keys.Right))
```

Each behavior's control signal rotates the body one way or the other. The `activation:` clauses give the numeric activation/utility of the behavior. `Trigger-on-key` is a signal function whose value is 1 when the key is pressed and 0 otherwise:

```
(define-signal-procedure (trigger-on-key k)
  (if (key-down? k) 1 0))
```

thus posture behaviors are active when their respective keys are pressed.

Forward motion is controlled by sending a vector in the forward direction to the gait control system. This is accomplished with a locomotion behavior that, again, is triggered by the press of the appropriate arrow key:

```
(define-locomotion-behavior walk
  this.FacingDirection
  activation: (trigger-on-key Keys.Up)))
```

The four expressions above are the complete code for manually piloting the character.

Stealth Humor

The next example involves two characters and a simple simulation of the startle reflex. One character, Fred, sneaks up behind another, Lefty, and taps him on the shoulder, startling Lefty and causing him to jump. Lefty's arousal level (wakefulness) is determined by how much noise Fred has produced while sneaking up on Lefty (as with a stealth mechanic). Lefty's startle response, which involves jumping into the air and flailing his arms with deliberate cartoon exaggeration, is inversely proportional to his arousal level.

We start with the code to make Fred sneak up on Lefty. Fred needs to approach Lefty until he is within range and then stop and tap Lefty on the shoulder. This is implemented by the following fragment:

```
(within fred
  (define-signal touched-lefty?
    (flip-flop (not (null? fred.ContactObject))
               false))

  (define-locomotion-behavior approach-lefty
    (saturate (- lefty.position fred.position)
             0.03)
   activation: (if touched-lefty? 0 1))

  (define-locomotion-behavior tap-lefty
    @(0 0 0)
    activation:
    (if (< (distance fred.position
                     lefty.position)
           0.4)
        2
      0)
    exclaim: "boo!"
    reach: (not touched-lefty?)))
```

The `touched-lefty?` signal is false until Fred touches Lefty, after which, the flip-flop keeps it true even if Fred stops touching Lefty. `Approach-lefty` walks toward Lefty at 3cm/sec until `tap-lefty` overrides it when Fred comes within 40cm. The `exclaim:` and `reach:` clauses send ancillary signals through blackboard channels to forcibly trigger the reaching behavior, and to generate a speech bubble ("boo!").

Now, for Lefty. We start with the sensory code that implements the stealth mechanic; the faster and closer Fred is, the noisier he is:

```
(within lefty
  (define-signal fred-distance
    (distance fred.Position
              lefty.Position))

  (define-signal fred-noise
    (/ (* 1 (magnitude fred.Velocity))
       (max 0.1 (square fred-distance))))
```

To simulate Lefty's arousal, we say that the noisier Fred is (and the longer he's noisy) the more Lefty wakes up:

```
(define-signal arousal
  (leaky-integrator fred-noise 10)))
```

The leaky integrator insures that sustained noise over a period of time will wake Lefty, but that if Fred is quiet for a time, then Lefty will go back to sleep.

Next, we define an ambient behavior for Lefty that produces a rough simulation of sleeping standing up. As his arousal level decreases, Lefty gradually leans forward and sags (by applying a force to the shoulders pushing them forward and down). However, the sagging force is modulated by a sine wave, producing a cartoon approximation to breathing during sleep:

```
(within lefty
  (define-posture-behavior stoop
    (posture-force
       shoulder-force:
       (* (abs (sin (* 0.5 time)))
          (saturate (/ (- lefty.TorsoForward
                          lefty.TorsoUp)
                       (max arousal 0.5))
                    100)))
    activation: (if (= 0 (count touched?))
                    1 0)))
```

Finally, the actual startle behavior. Its strength is inversely proportional to both Fred's distance and Lefty's arousal. The physical reaction is produced by stretching the character along his spine for 0.4 seconds:

```
(within lefty
  (define-signal touched?
    (not (null? lefty.ContactObject)))

  (define-signal startle-level
     (if touched?
         (/ 10 (* fred-distance arousal))
         0))

  (define-posture-behavior startle-jump
    (posture-force spine-extension-delta: 10)
    activation:
    (if (one-shot (> startle-level 1)
                  0.4)
        startle-level
        0)
    exclaim: "#$%!"))
```

Evaluation

Evaluating a system like *Twig* is difficult. It does not introduce any stunning new algorithms, and in general does not let the user do anything that couldn't be done, at least in principle, with some other system. Its claim instead is that it makes it easier to produce rough-and-ready prototypes of a wider range of character AIs than could be done with other systems. The examples given in the previous section are an attempt to establish the plausibility of this claim.

Twig was developed research both in my own lab and at other universities. In research, it has been used for a simulation of a biological model of extroversion and neuroticism in humans (Fua, Horswill et al. 2009), as a behavioral back-end for the ICARUS agent architecture (Langley, Trivedi et al. 2010), and for cinematic camera control (Jhala, unpublished). These projects used the older version of the system, whose scripting was very limited. The design of the current system was driven in large part by feature requests from these other projects.

An early version of the scripting system described here was used last year as the basis for a course on simulation-based virtual characters. The course consisted of 7 juniors, 16 seniors, and 3 graduate students, most of whom were CS majors. The course had no AI or graphics prerequisites, and most students had little knowledge of these areas.

Although they complained about using Emacs, they were able to learn to use *Twig* relatively quickly. The embedded DSLs had both pedagogical advantages and disadvantages, however. On the one hand, they allowed code to be written more concisely and made large classes of bugs impossible. However, the embedding of the DSLs within Scheme meant they all "looked like Scheme" to the students, so students would often mix the languages in nonsensical ways. Many students also said they would prefer a more familiar language like C#. However, those students who later took our Game Engine Design course, where they were forced write character behavior code in C# said that *Twig* had been much easier to use and debug.

Students implemented a wide range projects, including dancing, soccer, fighting games, simple narratives, and reimplementations of existing games, such as Super Mario Bros. and Pac-Man.

Since *Twig* allows users to write script code that runs at frame rate (60Hz) performance is an important concern. Thus far, the system has been more than fast enough for our needs; Idiotball, a simple soccer simulation used in class, uses approximately 2% of a 2.4GHz Core 2 Duo laptop to simulate 7 characters with 6 behaviors and their associated (simple) sensory systems. In the class, performance was only a problem for one project, which involved 53 characters and 230 objects in a relatively densely-packed environment. This swamped Twig's collision detection system, which does no broad-phase pruning. It could be improved easily by implementing a broad-phase algorithm such as sweep-and-prune.

Conclusions

The system described here represents an early attempt to allow researchers to build interactive characters with flexible sensory and motor systems and to drive those characters from research AI systems, without substantial background in graphics or game programming. Although not suitable for all applications, it allows AI researchers to control characters that move and interact in engaging manners, including behaviors that involve sustained contact between characters, such as hugging, holding hands, or walking together, without having access to the infrastructure of an animation studio. While still in its early stages, initial results with the authoring system are encouraging. The *Twig* system is open source and currently hosted on SourceForge at twig.sourceforge.net.

Acknowledgements

I would like to thank the reviewers for their helpful and clarifying comments and suggestions. I would also like to thank Karl Fua, Pat Langley, Arnav Jhala, Sam Fansler, Mohini Tellakat, and the students of EECS-395 for being remarkably patient alpha testers of various incarnations of the system. Finally, I would like to thank Michael Mateas, Chuck Rich, and Richard Evans for their kind words and encouragement.

References

Arkin, R. (1998). *Behavior-Based Robotics*. Cambridge, MIT Press.

Badler, N. I., C. B. Phillips, et al. (1993). *Simulating Humans: Computer Graphics Animation and Control*, Oxford University Press.

Brooks, R. A. (1986). "A Robust Layered Control System For A Mobile Robot." *IEEE Journal of Robotics and Automation* **RA-2**(1): 14-23.

Fua, K., I. Horswill, et al. (2009). Reinforcement Sensitivity Theory and Cognitive Architecture. *AAAI Fall Symposium on Biologically Inspired Cognitive Architectures (AAAI Technical Report FS-09-01)*. Arlington, VA, AAAI Press**:** 52-54.

Gemrot, J., R. Kadlec, et al. (2009). Pogamut 3 Can Assist Developers in Building AI (Not Only) for Their Videogame Agents. *Agents for Games and Simulations: Trends in Techniques, Concepts and Design* (Proceedings of AGS 2009). F. Dignum, J. Bradshaw, B. G. Silverman and W. v. Doesburg, Springer**:** 1-15.

Hecker, C. (2008). "Spore Behavior Tree Docs." Retrieved November 5, 2010, from http://chrishecker.com/My_Liner_Notes_for_Spore/Spore_Behavior_Tree_Docs.

Hecker, C. (2010). *Game AI is Game Desgin*. Invited talk, AIIDE 2010, Stanford, CA.

Horswill, I. (2009). "Lightweight Procedural Animation with Believable Physical Interactions." *IEEE Transactions on Computational Intelligence, AI and Computer Games* **1**(1): 39-49.

Horswill, I. D. (2000). "Functional Programming of Behavior-Based Systems." *Auton. Robots* **9**(1): 83-93.

Isla, D. and B. Blumberg (2002). Blackboard Techniques *AI Game Programming Wisdom*, Charles River Media.

Jakobsen, T. (2001). Advanced Character Physics. *Game Developer's Conference*. San Jose, CMP Inc.

Laird, J. E. and M. van Lent (1999). Developing an Artificial Intelligence Engine. *Game Developers' Conference*. San Jose, CA**:** 577-588.

Langley, P., N. Trivedi, et al. (2010). A command language for taskable virtual agents. *Sixth Conference Artificial Intelligence and Interactive Digital Entertainment*. Stanford, CA, AAAI Press.

Levine, S., Y. Lee, et al. (2011). "Space-Time Planning with Parameterized Locomotion Controllers." *ACM Transactions on Graphics* **30**(3).

Maes, P. (1989). "How to do the right thing." *Connection Science Journal* **1**: 291-323.

Mateas, M. and A. Stern (2005). Façade.

Muico, U., J. Popović, et al. (2011). "Composite Control of Physically Simulated Characters." *ACM Transactions on Graphics* **30**(3).

Norvig, P. (1991). *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*, Morgan Kaufmann.

Perlin, K. and A. Goldberg (1996). "Improv: A System for Scripting Interactive Actors in Virtual Worlds." *Computer Graphics* **30**: 205-216.

Rabin, S. (2002). Implementing a State Machine Language. *AI Game Programming Wisdom*. S. Rabin, Charles River Media**:** 314-320.

Stallman, R. M. (1981). "EMACS the extensible, customizable self-documenting display editor." *SIGPLAN Not.* **16**(6): 147-156.

Sterland, A., J. Lissiak, et al. (2007). Unreal Tournament .NET bots, Microsoft Corporation.

Thompson, J. (2010). "Yield Prolog." Retrieved March 15, 2011, from http://yieldprolog.sourceforge.net/.

Treuille, A., Y. Lee, et al. (2007). "Near-optimal character animation with continuous control." *ACM Trans. Graph.* **26**(3): 7.