Not for Citation. Submitted to IEEE Transactions on Computational Intelligence and AI in Games; Comments welcome

Lightweight Procedural Animation with Believable Physical Interactions

Ian Douglas Horswill[1]

***Abstract*—I describe a procedural animation system that uses techniques from behavior-based robot control, combined with a minimalist physical simulation, to produce believable character motions in a dynamic world. Although less realistic than motion capture or full biomechanical simulation, the system produces compelling, responsive character behavior. It is also fast, supports believable physical interactions between characters such as hugging, and makes it easy to author new behaviors.**

***Index Terms*—Virtual characters, interactive narrative, procedural animation.**

I. Motivation

Interactive narrative and similar AI-intensive applications require characters to perform a wide range of actions and gestures at run-time, the details of which may be difficult to anticipate at authoring-time. In the game industry, character animation is generally done either through motion-capture or hand-authored key-framing. Run-time animation is then a problem of selecting and blending pre-authored animation clips from a large library based on the behavior desired for the character and the geometric configuration in the character's immediate vicinity. These techniques can produce very realistic motions, but do so at a tremendous authoring cost. Not only do separate motions need to be captured for each character action, but they may need to be separately captured for different characters, for different variations on the character's motion, and for different objects being used. For example, sitting in a chair can require different animation clips depending on the type of chair, its height, and the character's height. Although some of this may be automated (see [1-4]), generating large libraries of character animations is still extremely labor-intensive, making it expensive for the game industry and prohibitive for universities, independent developers, and solo artists.

II. Motion synthesis

The natural alternative would be to compute motions algorithmically from first principles given some specification of the desired character behavior. These are sometimes divided into *procedural animation* systems, in which the animation algorithm is able to specify joint angles directly, and *dynamic simulation* systems, in which the output of the algorithm are forces or torques that control a separate physical simulation of the body.

Some of the earliest motion synthesis systems addressed animal behavior. Reynolds [5] described a system for computing the group behavior of flocking birds. Sims [6] used genetic algorithms to evolve bodies and locomotion controllers for synthetic agents. Tu and Terzopolis [7] implemented a dynamic simulation system that controlled a swimming fish that responded to environmental cues.

There has also been a great deal of work on animating bipedal motion [8]. Common approaches include kinematic solutions that compute joint angles without consideration of dynamics [9, 10], dynamic solutions that compute torques to be fed through physics [11-13], and hybrid approaches [14].

Many motion synthesis systems structure character control in terms of discrete behaviors that can be triggered by specific environmental or endogenous stimuli; such systems are sometimes referred to as *behavioral animation*. A number of architectures and frameworks have been proposed for controlling behavioral animation systems. Devilliers et al. developed a programming environment for developing animation behaviors based on hierarchical, parallel state machines [15]. Blumberg and Galyean [16] described a general behavior-based architecture for controlling character bodies and negotiating which behaviors had access to which of a body's degrees of freedom. Regelous' *Massive* system [17], used for large-scale crowed simulation in film and television, allows animators to specify character behavior using fuzzy logic [18].

There are also a few general-purpose systems that perform motion synthesis. Goldberg and Perlin's *Improv* procedural animation system [19] provides a number of scriptable behaviors for use in interactive narrative and other entertainment applications. Badler et al.'s *Jack* system [20-22] performs low-level control of humanoid bodies using a combination of parallel state machines and constrained inverse-kinematics, and provides a higher-level control interface based on natural language and AI planning techniques. Although not a procedural animation system *per se*, *SmartBody* [23] provides a set of scheduling and synchronization mechanisms for blending and controlling animations, including procedurally generated ones. Natural Motion's *Euphoria* system [24], which provides a set of controllers for humanoid motor behaviors that can be connected to a game engine's physics system to control character behavior. *Euphoria* has been used in a number of recent titles, most notably *Grand Theft Auto IV* [25] and *Star*

[1] Manuscript received November 25, 2008. Ian Horswill is with Northwestern University, Evanston, IL 60208 USA (847-467-1256; fax: 847-491-5258; e-mail: ian@northwestern.edu).

Wars: The Force Unleashed [26]. Although the exact capabilities of *Euphoria* have not been published, the *Eurphoria:core* system is available as part of a pre-packaged end-user application called *Endorphin* [27], which supports 9 arm behaviors ranging from "Hands Reach and Look At" to "Hands Protecting Groin," 3 leg behaviors, and 14 whole-body behaviors, such as "Catch Fall" and "Writhe in Mid-Air".

Despite this extensive work, versatile, extensible systems for motion synthesis that support complex physical interactions between character and the environment are still largely unavailable. As a result, most interactive narrative systems are built using commercial game engines such as those of *Half-Life 2* [28] or *Unreal Tournament 3* [29], although Mateas and Stern's *Façade* being a notable exception [30]. Because these game engines are not designed for interactive narrative applications, they often require authors either to develop extensive animation assets or limit themselves to the behavioral repertoire of typical first-person shooter characters.

III. Twig

In this paper, I describe *Twig*, a fast, AI-friendly procedural animation system that supports easy authoring of new behaviors. *Twig* provides behaviors for locomotion, object manipulation, and gesturing, and allows characters to interact physically with each other and with their environments in a believable manner. It also allows programmers to define new behaviors by composing simple control loops. Character joints are controlled directly in Cartesian space (as opposed to joint coordinates), using whatever combination of kinematic, dynamic, and constraint-based control modes are appropriate.

The system is structured in roughly four layers (Fig. 1. *Twig* software stack). First, a minimalist physics simulation provides the back-end to all motion control. It provides both dynamics simulation and resolution of collisions and kinematic constraints. Above this layer is a basic motion control system that implements functions such as limb control, posture, and walking. This layer is then controlled by a behavior-based system, similar to those used in robotics [31] and virtual creature systems [32]. These higher-level behaviors are driven in part by a simple attention simulation. Characters can be run either autonomously, controlled by a separate system using and RPC interface, or scripted directly.

It's interesting to note that the dynamic simulation actually simplifies control, allowing the use of relatively crude control signals, which are then smoothed by the passive dynamics of the character body and body-environment interaction; similar results have been found in both human and robot motor control [33]. Indeed, *Twig* shows that surprisingly simple techniques can generate believable motion and interaction. Much of the focus of this paper will be on ways in which *Twig* is able to cheat to avoid doing complicated modeling or control, while still maintaining believability. This paper is indebted to the work of Jakobsen [34] and Perlin [19, 35, 36], both for their general approaches of using simple techniques to generate believable motion, and for specific techniques noted below.

Twig is built on the Microsoft *XNA* platform [37] and is very efficient, running easily at 60Hz on a single core of a low-end machine. It is free, open-source software distributed under the Lesser Gnu Public License (LGPL) [38].

IV. Limitations

Twig is intended as a research tool. Its current repertoire of character behaviors still falls well short of what real actors can do.[2] However, it demonstrates that its approach to simulation and control is effective for the class of applications for which it's designed. Further behaviors can be easily added.

On the other hand, *Twig* is designed for versatility and "believability" [39] rather than physical realism.[3] While it generates surprisingly compelling character motion, modifying it for true physical realism would require major changes. A more accurate physics engine such as *Havok* [40] or *ODE* [41], and a more biologically-correct gait simulation [13, 42] may be more appropriate for works and genres requiring greater realism.

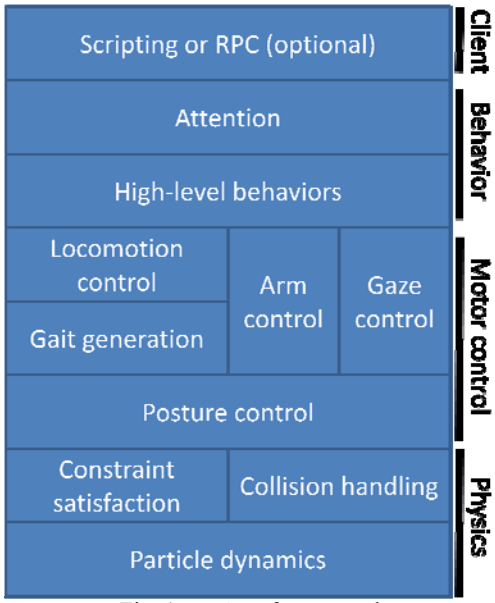

Fig. 1. *Twig* software stack

V. Geometric and Kinematic Modeling

Twig objects are represented internally as a set of point-particles called *nodes*, together with a set of collision volumes attached to the nodes. Collision volumes may be capsules (rounded cylinders), spheres, or boxes. Nodes are the only containers of kinematic and dynamic state in the system, so positions and orientations of objects and their collision volumes are determined entirely by the positions of their

[2] As of this writing, the repertoire is limited to navigation (walking/running), sitting/standing, gesturing, reaching for, holding and dropping objects, writing with/on objects, fighting and hugging, and withdrawal from pain

[3] I use the term "believable" in the technical sense used by the animation and believable agents communities. A character is believable if it appears sufficiently life-like to an audience that they are willing to suspend disbelief and relate to it as a living creature. Most cartoon characters are designed more for believability than literal physical realism.

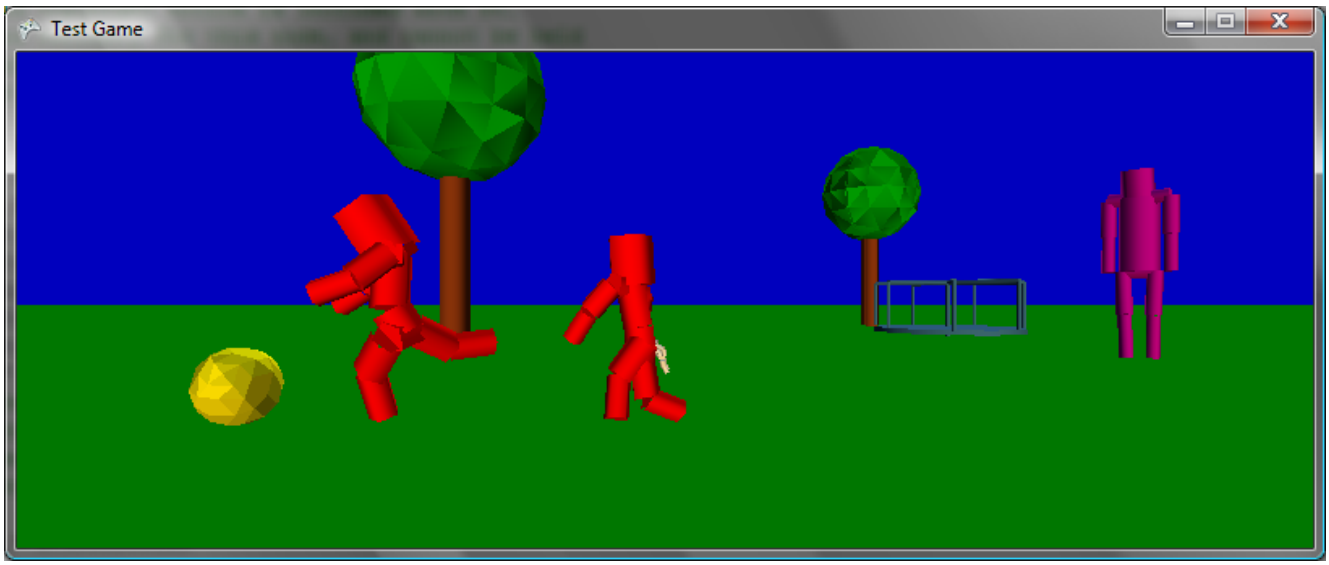

Fig. 2. Playground scene produced with Twig

associated nodes. Meshes for rendering are stored separately, with mesh transforms being computed from node positions at render time.

Most Twig objects are composed of a set of nodes connected by rigid rods called *links* (see Fig. 3. Kinematic model of a character in *Twig*. Circles represent point particles (nodes) that form the joints and endpoints of the limbs and trunk. Lines represent rigid distance constraints (links) between nodes.). Links function both as collision volumes and as kinematic constraints that force the nodes they connect to be a specific distance from one another. Links can be joined in kinematic chains by sharing nodes. The shared node then acts as a spherical joint (i.e. it can bend in any direction).

In the schoolyard scene shown in Fig. 2. Playground scene produced with Twig, the characters are modeled as 13 links (2 each for the spine and each arm and leg, and one each for the head, shoulders, and pelvis), connecting 16 nodes. The ball is represented as a single node. The merry-go-round, which is functional, is modeled as 18 nodes and 41 links; 25 of the links are visible and have collision volumes (the bars), and the rest are invisible links used only to hold the structure rigid.

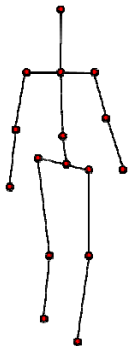

Fig. 3. Kinematic model of a character in *Twig*. Circles represent point particles (nodes) that form the joints and endpoints of the limbs and trunk. Lines represent rigid distance constraints (links) between nodes.

VI. Dynamics Simulation

Twig uses a mass-aggregate physics system [43] based on Jakobsen's work on the *Hitman* engine [34, 44], in which objects are modeled as point masses (the nodes) connected by massless rods (the links), and motions are computed using Verlet integration [45]. The AGEIA PhysX engine [46] also uses a related "position-based" approach to build a much more general dynamics engine. However, the simpler system discussed here is sufficient for our purposes.

In Verlet integration, the dynamic state of a particle is represented in terms of its position in the current frame and previous frame, rather than its position and velocity. Given a fixed inter-frame interval, Δt, we can describe the position $\mathbf{p}$ of a node at time $t + \Delta t$ in terms of its position in the previous frames as:

$$\begin{aligned}\mathbf{p}(t+\Delta t) &\cong \mathbf{p}(t) + (\mathbf{v}(t) + \mathbf{a}(t)\Delta t)\Delta t \\ &\cong \mathbf{p}(t) + \frac{\mathbf{p}(t) - \mathbf{p}(t-\Delta t)}{\Delta t}\Delta t + \mathbf{a}(t)\Delta t \Delta t \\ &= \mathbf{p}(t) + \big(\mathbf{p}(t) - \mathbf{p}(t-\Delta t)\big) + \mathbf{a}(t)\Delta t^2 \\ &= 2\mathbf{p}(t) - \mathbf{p}(t-\Delta t) + \mathbf{a}(t)\Delta t^2\end{aligned} \quad (1)$$

where $\mathbf{v}(t)$ is the node's instantaneous velocity and $\mathbf{a}(t)$ its acceleration at time t. If we want to model viscous damping, this can be done by modifying the relative weighting of the position in the two frames:

$$\begin{aligned}\mathbf{p}(t+\Delta t) = (2-d)\mathbf{p}(t) - (1-d)\mathbf{p}(t-\Delta t) \\ +\mathbf{a}(t)\Delta t^2\end{aligned} \quad (2)$$

where d is the damping factor.

This scheme has a number of advantages. First, the complete kinematic and dynamic state of an object is contained in the positions of its nodes, together with their stored positions from the previous frame (links function only as collision volumes and distance constraints on node position). The lack of explicit representation of momentum,

angular momentum, or even orientation, significantly simplifies the dynamics calculations. Second, it makes constraint satisfaction much easier, since node positions can be directly modified to enforce constraints, without having to compute their effects on orientation, angular momentum, etc. Finally, it allows the behavior system to control the characters and their nodes entirely in Cartesian space, without having to deal with joint angles or nested coordinate frames. The cost of the design is that in the few cases where momentum, orientation, or joint angles are needed to make control decisions, these need to be computed from position data.

A. *Friction, drag and damping*

The current system does not support accurate models of friction or drag. Instead, it provides a damping term (see equation above), whose coefficient is large when a node is in contact with a supporting surface, and small when in the air. While this is technically inaccurate (damping is linear in velocity, whereas drag is quadratic and friction includes a step function), the inaccuracies generally aren't apparent to a viewer.

Nodes can be damped relative to the environment frame (modeling air friction) and/or relative to another node in the object (a crude model of the biomechanical damping of muscles and tendons). Nodes can also be locked in place to model large static friction forces.

B. *Kinematic constraints*

Kinematic constraints (joint limits, rigid distance constraints, etc.) are implemented by projection, i.e. by moving a node that violates a constraint to a nearby position that does not violate it. To locally enforce the distance constraints imposed by a link, we measure the actual distance $\|\mathbf{p}_i(t) - \mathbf{p}_j(t)\|$ between its endpoint nodes and compare it to the desired distance, d. If the nodes are not the desired distance apart, we move each node half the difference between the desired and actual distances, weighted by their respective masses, m_i and m_j:

$$\begin{aligned}\mathbf{p}_i(t) &= \mathbf{p}_i(t) - m_i^{-1}\frac{\|\mathbf{o}\| - d}{2\|\mathbf{o}\|(m_i^{-1} + m_j^{-1})}\mathbf{o} \\ \mathbf{p}_j(t) &= \mathbf{p}_j(t) + m_j^{-1}\frac{\|\mathbf{o}\| - d}{2\|\mathbf{o}\|(m_i^{-1} + m_j^{-1})}\mathbf{o}\end{aligned} \quad (3)$$

where $\mathbf{o} = \mathbf{p}_i(t) - \mathbf{p}_j(t)$ is the offset between the nodes.

Constraint satisfaction is also used to enforce joint limits. For example, the knee is a revolute joint, meaning it's constrained to rotate about a specific axis. However, the underlying simulation simulates spherical joints, meaning they can rotate about any axis. To prevent the knee from bending sideways, it's necessary to bring it into alignment by constraining it to lie in the plane formed by the hip, foot, and forward direction of the pelvis. The normal to this plane is given by:

$$\mathbf{n} = (\mathbf{r} - \mathbf{e}) \times \mathbf{f} \quad (4)$$

where $\mathbf{r}$ and $\mathbf{e}$ are the positions of the root and end nodes (hip and foot) of the leg, respectively, and $\mathbf{f}$ is the forward direction of the pelvis. If the knee is in alignment, then the positions of the foot and knee will project equally along this axis, so $\mathbf{e} \cdot \mathbf{n} = \mathbf{j} \cdot \mathbf{n}$, where $\mathbf{j}$ is the position of the leg joint (knee). The error in the knee's position is therefore:

$$\Delta\mathbf{j} = ((\mathbf{j} - \mathbf{e}) \cdot \mathbf{n})\,\mathbf{n} \quad (5)$$

The constraint is then enforced by shifting the foot position by $\Delta\mathbf{j}/2$ and the knee position by $-\Delta\mathbf{j}/2$.

On each update cycle, each object tests its nodes against its constraints and adjusts node positions as necessary to locally satisfy the constraint being evaluated. This has the potential to violate constraints, but such violations are generally not detectable by the user, especially if the object is moving. Moreover, if the object stops moving, it quickly relaxes into a configuration that satisfies the constraints.

Projection is computationally efficient, but not especially accurate since it does not necessarily conserve energy or in all cases, even momentum. However, in practice, it generates motions that look real enough. Again, the goal is believability, not numerical accuracy.

C. *Constraint satisfaction in kinematic chains*

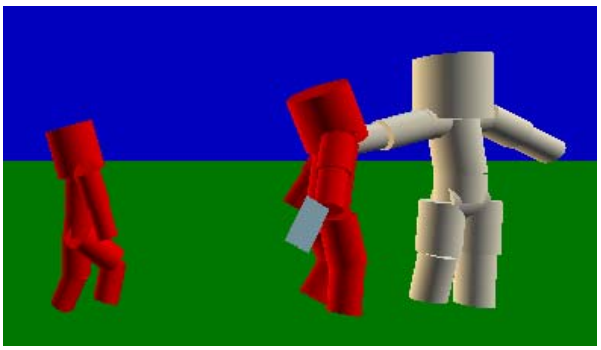

Fig. 4. A character drags off stage another character, who in turn drags another object (a clipboard), while a third character approaches.

As mentioned above, constraint satisfaction provides a mechanism for handling kinematic chains and provides a kind of simplified inverse kinematics, allowing control systems to apply forces to endpoints of kinematic chains, or even directly position them. If a character's hand is moved to a new location, its link to the elbow will drag the elbow node along with it, and that may, in turn, drag the shoulder, or even the whole rest of the character.

Grasping is implemented by creating temporary zero-length links between the hand node of a character and one of the nodes of the object being grasped. If the character moves their arm, it will then drag the object along with it. However, it is often easier to implement object manipulation by allowing the object to drag the character rather than the other way around (see section VIII, below).

Fig. 4 shows an example in which one character drags another entire character who drags another object (a clipboard), in turn. Here, the hand of the character on the right (the dragger) is linked to the shoulder node of the middle character, whose hand is linked to one of the corners of the clipboard. The flow of forces in this example starts with the shoulders of the character on the right, which are trying to center themselves above the character's pelvis. This drags the

arm, which drags the other character's shoulders, and along with them, the arm and clipboard.

D. Collision handling

Collisions are handled as a special case of constraint satisfaction. After each dynamic object is updated, its collision volumes are tested against the collision volumes of other objects and their nodes moved so as to separate their collision volumes.

We will discuss the link/link collision case, since most collision volumes in *Twig* are attached to links. Other cases are handled analogously. Let the endpoints of one link be nodes i and j, and the endpoints of the other be nodes k and l, with positions $\mathbf{p}_i$, $\mathbf{p}_j$, etc. Since links are modeled as cylindrical collision volumes, this can be reduced to testing the distance between the line segments $\overline{\mathbf{p}_i\mathbf{p}_j}$ and $\overline{\mathbf{p}_k\mathbf{p}_l}$. If the distance between them is less than the sum of the radii of the two cylinders, then they interpenetrate and need to be separated. To be physically accurate, we should determine the precise points of contact on the two cylinders, compute the relevant torques and moments of inertia, and update the positions of the endpoints accordingly. However, in practice, the links are almost always chained with other links that constrain their allowable motion. Since these inter-link constraints dominate the dynamics of the collision, we can obtain realistic looking collisions by translating the colliding cylinders apart, ignoring torques, and allowing the inter-link constraints to produce a realistic-looking motion.

In particular, let $\mathbf{n} = \frac{(\mathbf{p}_i - \mathbf{p}_j)\times(\mathbf{p}_k - \mathbf{p}_l)}{\|(\mathbf{p}_i - \mathbf{p}_j)\times(\mathbf{p}_k - \mathbf{p}_l)\|}$ be the contact normal along which the cylinders intersect. The distance between the spines of the cylinders is then $r = (\mathbf{p}_i - \mathbf{p}_k)\cdot\mathbf{n}$. If the radii of the two cylinders are a and b, then the penetration depth of the cylinders is $d = a + b - \|r\|$. We then translate both nodes i and j by $(d/2)\mathbf{n}$ (half the penetration), and we translate both nodes k and l by $-(d/2)\,\mathbf{n}$.

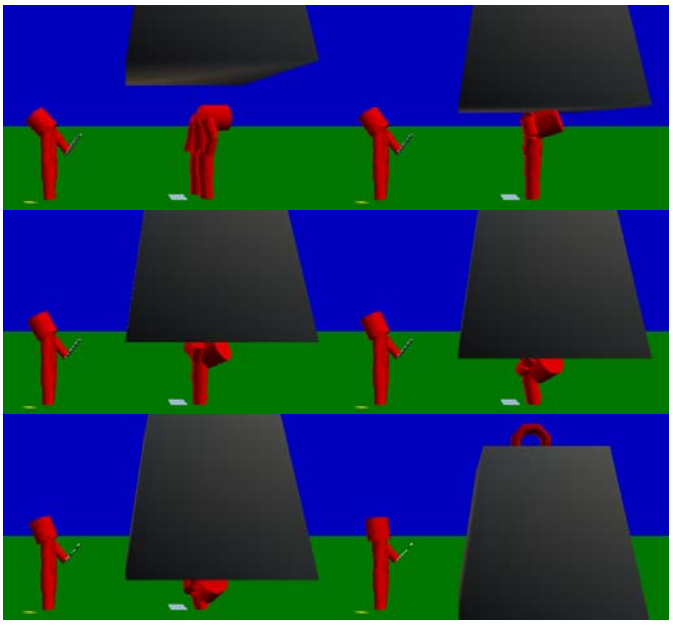

Fig. 5. Object collision in an homage to Cleese *et al*. [47].

A collision impulse could also be added to the links to simulate elastic collision. However, since humans don't bounce well, this would be counter-productive for links representing body parts.

Overall collision detection works in two phases. First, broad-phase detection is performed by projecting each object's position into the ground plane, and testing the distance of each object's projected position against a collision radius (this is equivalent to approximating objects as axis-aligned cylinders). If two objects are close enough, then their respective collision volumes are enumerated and testing against one another exhaustively. When collision volumes intersect, their contact points, contact normals, and penetration depths are computed and their corresponding nodes are translated apart, weighted by their respective masses.

Fig. 5 shows an example of a collision between a character and a non-cylindrical object (a 16 ton weight). The object's collision volume is modeled as an oriented bounding box.Fig. **5**

E. Tactile sensing

When a body part detects a collision, it stores a pointer to the object that hit it. This allows characters to detect when they are touching objects, and approximately where the contact is occurring. The system also computes the kinetic energy of the impact. If the kinetic energy is over threshold, the system registers it as pain. Characters also maintain an overall pain level, which decays exponentially over time.

VII. Low-level character control

All character behavior is ultimately implemented by moving nodes around. One of the advantages of the style of kinematic and dynamic modeling in *Twig* is that this control can be done directly in Cartesian coordinates, without having to deal with joint angles or perform explicit inverse kinematics.

A. Node control

Nodes are controlled principally by setting their velocity or acceleration. However, their positions can also be set directly, or they can be directed to perform a linear motion to a set-point. In the latter case, the node automatically moves along a straight line to arrive at the target in a specified amount of time without further need for control. This mode is used principally for limb motions. Nodes can also be locked in position or told to lock themselves when they come into contact with the ground plane.

B. Posture Control

Posture is controlled by applying forces directly to the nodes of the torso and pelvis, rather than by balancing the body as an inverted pendulum using simulated muscular forces. This makes control simple and stable at the cost of sometimes violating physical realism (for example, the current version of the system applies postural forces even when the legs aren't touching the ground). Again, this is adequate for the tasks we're considering, but a more complicated scheme would be necessary for applications in which it was necessary to accurately model balance, tripping, falling, etc.

Posture control consists of a set of simple control loops:

- Standing is implemented by two control loops
  - A force is applied along the *Y* (up) axis to the center of the pelvis to hold it at standing height.
  - Forces are applied along the *X* and *Z* axes to horizontally align the center node of the pelvis with the midpoint of the feet.
- Sitting up works essentially like standing, except that the center node of the shoulders is controlled so as to place the character's center of mass directly over the midpoint of the two feet. The shoulders are also tilted slightly in the direction of motion when the character is running.
- Orientations are controlled by twisting the pelvis and shoulders. Since the dynamics engine doesn't explicitly support torques, the torque is produced by applying opposite forces to opposite sides of the character.
  - The pelvis rotates to align with the direction of walking
  - The shoulders rotate to align with the gaze direction, subject to the constraint that they not rotate more than 90 degrees relative to the pelvis.

Note that these control loops are simple proportional controllers rather than proportional-derivative controllers (i.e. they have no damping term). They rely on the damping of the nodes themselves to prevent oscillation.

C. Limb control

The head controller points the "front" of the face toward the current gaze target, or the direction of motion, if there is no gaze target. In the current version of the system, this is an instantaneous motion. This will undoubtedly need to be changed to a smooth motion in the future, but since the current characters have no faces, this kind of exaggerated motion is actually useful for cuing the viewer that the character's gaze is shifting.

The arm controller currently supports five actions: swing (used when walking), reach, grapple, hug, and grab. Swinging is implemented by applying impulses to an arm when the opposite foot begins a step. At the level of the limb controller, reaching, grappling and hugging are all implemented by moving the hands directly in front of the shoulders at near-maximum extension. The rest of the reach, hug, and grapple actions are then controlled by higher-level controllers. Again, grasping is implemented by creating an invisible, zero-length link between the character's hand and one of the nodes of the object to be grasped.

Hugging. Hugging is implemented by reaching and approaching the target, then joining the hands when the target object makes contact with the character's torso.

"Grappling". Grappling is a kluge that is implemented by waiting until the character closes to within less than an arm length of the target and then engaging reaching, causing the arms to bash into the other character. This looks like shoving, punching, or wrestling to the viewer. It also tends to cause pain in the other character, triggering its pain withdrawal reflex, thus making it step back. While insufficient for a fighting game, it's sufficiently realistic for depictions of children fighting.

Legs are controlled by the gait controller (see below).

The system also supports simulated respiration by moving the shoulders up and down in a sinusoid, similar to [36]. Respiration increases with increased walking speed. In the current system, respiration is largely invisible to the viewer because the shoulders are modeled as a single cylinder, however they could be split to make it more apparent.

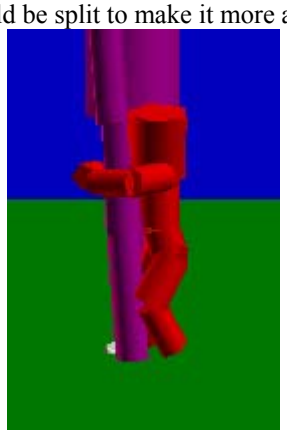

Fig. 6. A child character hugs its parent.

D. Gait Control

The gait generator drives the character to walk with a direction and speed chosen by one of the higher-level behaviors. Gait generation is largely kinematic and is closest to the work of Perlin [35]. The gait generator sets the ground-plane velocity of the pelvis to the walk vector, then monitors the extension of the legs. When a leg is sufficiently far behind the pelvis, the gait generator moves the foot node on a ballistic trajectory to a point in front of the pelvis, but in the direction of the walk vector. The constraint handling system moves the knee appropriately and insures that it doesn't bend backward or sideways.

E. Gesturing

Twig also provides support for playing back fixed gestures. Gestures are defined by specifying hand positions in a series of key frames stored in an XML file in the XNA Content Pipeline. In order to allow gestures to be easily ported from one body to another, hand positions are represented in a torso-centered coordinate system that can be normalized to the size of the character's arm, torso, or head. For example, to represent holding the arm fully extended from the body, one would use a coordinate system normalized to arm length, while to represent holding an arm aligned with the middle of the body, one would use coordinates normalized to torso size.

Since this approach only works for gestures that either don't reference other objects, or that reference one's own body, it will be necessary in future to allow key frames to be specified in the coordinate system of a target object. For example to represent patting another character on the back, one would want to represent the motion in a coordinate system centered on the other character's torso. However, this has not yet been implemented.

VIII. Object Manipulation

Twig currently supports two modes of holding an object. Hold holds the object loosely at the character's side. Hold is

implemented simply by creating a link between the character's hand and a specified node of the object.

In contrast to plain Holding, HoldForUse places the object in an object-specific pose appropriate for manipulation. HoldForUse is implemented by having the object compute its own desired pose and hover there, dragging the character's arm along with it. This simplifies design and makes control more stable.

A. *Task-specific coordinate systems*

Manipulable objects are allowed to specify task-specific coordinate systems called *charts*. A chart defines an object centered-coordinate system intended for use in a given type of manipulation. Charts are generally tied to the surface of the object, so that within the chart's coordinate system a point whose *Y* coordinate is 0 will lie on the object's surface, and whose *Y* coordinate is greater than zero will lie above the surface. Charts also specify the surface normal at a given point. Each object provides an *atlas*, which is a dictionary of named charts.

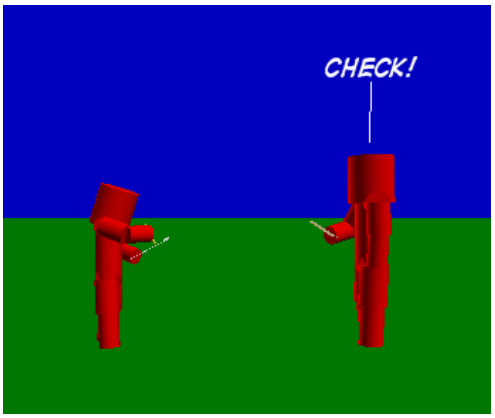

Fig. 7. Two characters hold papers, while one checks off items using a pen.

For example, when the character on the left in Fig. 7 uses a pen to write on the clipboard, the pen retrieves the chart named "front" from the clipboard's atlas and positions its endpoint at a specified location on the clipboard's surface. When the pen is not writing it lifts itself above the clipboard by moving a specified distance along the local *Y* axis in the front chart. Again, as the pen moves, it drags the character's hand with it.

The technique of allowing the manipulandum to drag the character's arm, rather than embedded a control loop in the arm works well in general. However a problem occurs if the character's facing direction is left unconstrained. The problem occurs because on each frame the object computes its target pose in terms of the character's current pose and facing direction. However, in dragging the character's arm, the object may rotate the character's shoulders slightly, resulting in a new facing direction, and hence a new target pose for the object in the next frame. This can cause the character to slowly spin in place if the character's orientation system isn't given a specific object or direction to lock onto.

IX. HIGH-LEVEL CONTROL

Characters can either be controlled through a separate sequencer (e.g. through scripting or a remote procedure-call interface), or they can run autonomously. Although this is the least well-developed part of the system, the current high-level behavior system consists of two main components. First, an attention system scans and appraises the objects in view to determine a focus object. Objects are reappraised on each clock tick, but focus switching is inhibited for a refractory period (1s) after each switch to prevent thrashing. The attention system runs autonomously, and usually has control of the gaze system. In addition to the attention system, a set of hierarchically structured high-level behaviors compete to send commands to the motor system. Each behavior computes an activation level (a rough measure of how useful it would be to fire the behavior at the moment) as well as a set of motor commands to send to the level below. Siblings in the hierarchy compete with one another; the behavior with the highest activation level is chosen to send its commands to the lower levels. Again, switching is inhibited for a short refractory period (0.2-0.75 seconds) each time a new behavior is selected to prevent thrashing. This forms a hierarchical behavior selection system similar to Blumberg's work on ethologically inspired control [32].

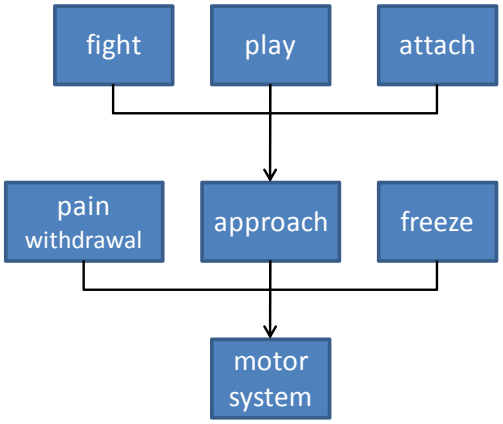

Fig. 8. Behavior network for the attachment simulation.

Fig. 8 shows the high-level behavior network for the safe home base simulation described below. The motor system is controlled by three main behaviors. Freeze is the default behavior, which does nothing. Pain-withdrawal triggers automatically when the character experiences pain and moves the character away from whatever object caused the pain. Approach is the main motor behavior. It steers the character toward a designated object, while avoiding obstacles (see section IX.A, below).

While sufficient for the simulation for which it was originally designed, the current high-level control system is quite limited. In particular, the lower-level motor system supports a number of behaviors, such as sitting and object manipulation, that aren't used by the higher-level system. At present, these can only be used by the scripting and RPC interfaces.

A. Object approach

The approach behavior is worth some discussion on its own. It takes as input a target object, a distance d from the object to stop at, and a direction $\mathbf{d}$ from which to approach it. Approach also takes as input settings for the hug, reach, and grapple controls, which it forwards to the arm control behaviors. It generates a walk vector, $\mathbf{w}$ (a velocity vector for the gait controller), based on an artificial potential-field/motor-schema [31] which is the sum of an attraction component, $\mathbf{a}$, in the direction of the target object, and a repulsion component, $\mathbf{r}$, pushing away from any intervening obstacles:

$$\begin{aligned} \mathbf{w} &= \mathbf{a} + \mathbf{r}, \text{ where:} \\ \mathbf{a} &= \mathbf{p}_{\text{target}} - \mathbf{p}_{\text{character}} + d\mathbf{d} \\ \mathbf{r} &= \sum_{o \neq \text{target}} c \frac{\mathbf{p}_{\text{character}} - \mathbf{p}_o + (\mathbf{p}_{\text{character}} - \mathbf{p}_o) \times \mathbf{e}_{UP}}{\max(d_{min}, \|\mathbf{p}_{\text{character}} - \mathbf{p}_o\|^2)} \end{aligned} \quad (6)$$

where $\mathbf{p}_x$ here denotes the position of object x, c and d_{min} are constants tuned to taste, and $\mathbf{e}_{UP}$ is a unit vector pointing upward. The cross product term produces a curl component to the field that pushes the character around obstacles, making it less likely that they will encounter local minima in the field. To avoid asking the walk system to move too fast, the $\mathbf{a}$ component is also saturated to prevent its magnitude from going over a threshold. To reduce computation time, objects over a threshold distance are ignored when computing $\mathbf{r}$.

The approach behavior works well for relatively uncluttered environments. For maze-like environments, a more complicated path-planning system would be necessary. However, such a system could be easily incorporated.

X. Rendering

Currently, most objects in Twig are either built out of spheres and cylinders, or are modeled using an external 3D modeling tool such as Google SketchUp. The former are represented as a set of cylindrical and spherical collision volumes, each with its own separate mesh for rendering. The latter are generally approximated as a single OBB collision volume, mostly out of laziness.

Thus, objects typically have a separate rendering mesh for each of their collision volumes. The renderer computes a transformation matrix for each of these meshes from the positions of the nodes defining its collision volume and draws it. Thus, characters are currently drawn as collections of cylinders, which fits with the overall cartoon aesthetic of the system. However, it would be straightforward to compute bone transformations for a rigged mesh from the character's node positions, if greater visual realism was desired.

Rendering is currently the bottleneck of the system in spite the relatively low polygon count of the meshes. This is because each cylinder is drawn as a separate batch using the default XNA shader. The system could be sped up considerably by using a different shader that supported instanced meshes.

XI. Authoring tools

Twig supports the XNA Content Pipeline, a set of extensions to the build system of Microsoft Visual Studio designed to help manage media assets and convert them from external formats to internal binary formats.

A. Prop authoring

The content pipeline is most commonly used for importing models for props. Depending on the model, this may require the user to make a new C# class that understands how to render the model, what its collision volumes should be, and how to implement any special behavior of the object. Many props are passive, however, and can be approximated as boxes for purposes of collision detection. In these cases, the user can use the BoxModel class and specify the name of a mesh from the content pipeline. The BoxModel class will automatically load the mesh and compute its bounding box. The object can then be placed in the world.

B. Gestures

Users can author gestures through the content pipeline by adding XML files containing the necessary key frames for the gestures. The gestures can then be played back on demand by specifying the name of the gesture file and the hand(s) to play it through.

C. Scripting

Finally, the user can script the behavior of characters using a simple scripting language of the form:

*name***:** *method args* …

where *name* is the name of an object in the *Twig* world and *method* and *args* define an arbitrary C# method to call on the object. The script interpreter uses the .NET reflection interface to invoke the method.

The script interpreter normally ignores the return value of the method. However, in cases where one wants to script an action that takes time to complete, the method can return an Action object, which the script interpreter will poll on each clock tick until the Action reports the operation is complete. Parallel execution of durative actions can be forced by adding an "&" to the end of the command, which causes the script interpreter to continue to the next command without waiting for the Action object to report completion.

Scripts can be managed as assets in the content pipeline or loaded from text files at runtime.

D. RPC interface

Most research users will want to drive Twig from an existing system, most likely not written under .NET. For such users, the script interpreter can be run over a TCP socket, providing a remote procedure-call interface that should be comparatively simple to implement on the client side. The user can also TELNET to the socket and drive the characters manually by typing script commands on the keyboard, although this is useful mainly for testing purposes.

XII. Example applications

To date, *Twig* has been used for two main applications. It was originally developed as a back-end for a behavior simulation system, but then developed a life of its own. Since then, it has also been used in scripted mode to do a series of short episodic pieces, a sort of moving-image version of a web comic.

Fig. 1. *Twig* software stack shows a scene from the original system, a crude simulation of the "safe home base" phenomenon from Attachment Theory [48]. Here, a child makes excursions from its parent to explore the environment, and in particular, to play with a ball, but periodically returns to the caregiver to be soothed.

The demonstration involves three approach behaviors (see Fig. **8.** Behavior network for the attachment simulation.Fig. 8): playing with the ball, fighting, and running to hug the parent (attachment). The characters appraise each object in view or in short-term memory for its salience (interest level), valence (good/bad), and monitoring priority (how much to pay attention to it). The maximal salience object becomes the focus of attention for that update cycle. The different approach behaviors react to the focus of attention and change their activation levels depending on the type of object and its appraisal.

In parallel, the gaze control system shifts visual attention between the current focus of attention, the target of the approach system (if different), and other objects that have high monitoring priority (the parent and any potential threats).

The result is that the children run after the ball because it's highly valenced. As the small child gets farther from the parent, however, it becomes anxious and the monitoring priority of the parent increases, causing the child to periodically stop and look back to the parent. Eventually, the child's anxiety becomes sufficient for it to abandon the ball and return to hug the parent, which reduces the child's anxiety. Eventually, the child's attention returns to the ball, the child returns to play, and the cycle repeats.

XIII. Implementation and performance

Twig is written in C# and runs under XNA Game Studio 3.0 [37]. It consists of three separate libraries. The main Twig library implements the basic physics, animation, and behavior systems, as well as the script interpreter and built-in object types such as characters and the BoxModel. The TwigServer library is a separate library that can be linked in to support control over a TCP connection. The final library, TwigContentProcessors, provides *Twig*-specific extensions for the XNA Content Pipeline.

The physics system runs at a fixed update rate of 60Hz, since Verlet integration is unstable with variable step times. XNA allows the renderer to skip frames if it can't sustain 60fps, but this isn't an issue in practice unless there are a large number of characters on screen at once.

A debug build of the scene in figure 1 takes approximately 5ms per frame on a single core of a 1.6MHz notebook machine. Physics and behavior generally take 1-1.5ms when the characters are interacting and less than 0.5ms when the characters are widely spaced; this is because broad-phase collision detection is able to prune all intersection tests. Actual rendering is slower, generally around 3.8ms; however, there is considerable room for optimization here (see section X).

A. *Failure modes*

The simplified physics and control in *Twig* do cause occasional problems. For example, the walking system applies an external force directly to a character's torso, which then pulls the (largely passive) legs along, rather than by simulating muscular forces within the legs and torso. This can potentially allow a character to violate conservation by pushing the merry-go-round while standing on it, although this has yet to happen in practice.

The system's kinematic simplifications are also sometimes noticeable. Since characters are modeled internally in terms of node positions rather than joint angles, kinematic constraints must be added to simulate joint limits. While this is straightforward to do for the knees, it's harder to do for the elbows because of the wider range of motion at the shoulder than the hip. In the current system, the elbows sometimes seem to wiggle unrealistically because they fail to capture the true dynamics of a human arm, even though each individual arm position is kinematically possible for a human.

A final class of issues stems from conflicts within the behavior system itself. For example, if the child runs too fast when trying to hug the parent, it can impact the parent with enough force to cause pain. That triggers a pain withdrawal reflex during the docking phase of hugging. Although this behavior is realistic in the sense that real human children do it from time to time, it has the potential to turn a sentimental scene into slapstick.

XIV. Future work

Twig provides a useful back-end simulation and animation system for interactive narrative research. That said, it has a number of limitations. To make the system more useful, the RPC system will need to be extended. It provides good control of the simulated world, but currently has only minimal facilities for reporting back to client about the state of the world.

Another major deficiency is the lack of faces for characters. Although this is less bothersome than one might expect, it nevertheless is a significant limitation.

The current system also has little or no facilities for lighting and camera control. Incorporating intelligent camera [49] and lighting [50] control would be very useful.

Finally, the existing high-level AI system is quite limited. Extending it to handle more actions and perform better means-ends analysis, will be important. Real path and reach-planning would also be helpful for a number of applications.

XV. Conclusion

Twig is a simple, extensible, AI-friendly, procedural animation system. Although still under development, it provides a range of capabilities, including goal-directed character locomotion, object manipulation, and complicated physical interactions between characters, such as hugging and dragging. Because *Twig* is intended principally for interactive

narrative applications, its design emphasizes believability in the technical sense of making characters seem alive to an audience, rather than realism in the sense of precise duplication of human motion.

While dynamic control is generally more difficult than kinematic control, in *Twig* the use of a minimalist physics simulation actually simplifies the problem of authoring behaviors. The constraint satisfaction system in the simulator allows programmers to directly specify the constraints and forces on nodes in Cartesian coordinates, without having to program in terms of joint angles or use an explicit inverse kinematics system. In many ways, it allows the programmer to think kinematically when writing individual controllers, while allowing those controllers to interact through the constraint and dynamics system. This makes it relatively easy to combine the actions of different controllers without worrying, for example, that the torques introduced by the swinging of the arm when manipulating an object will drive the posture system into oscillation.

The cost of this design is physical realism. While it is convenient to implement walking by allowing the pelvis to drag the legs, or to implement object manipulation by levitating the object and allowing it to drag the arm, this does not produce the same forces and viscous damping that a true biomechanical simulation would produce. Thus even if the character's hand has the same trajectory as a human's would (which it may well not), the motion of the elbow and the concomitant postural changes of the shoulder, spine, and legs, will not identical to those of a human. They do, however, generally look lifelike. For our applications, this is quite sufficient. Accurate biomechanical simulation is left as an exercise for the reader.

ACKNOWLEDGEMENTS

I would like to thank Michael Mateas, Andrew Stern, Rob Zubek, Andrew Ortony, Magy Seif El-Nasr, and Chuck Rich, for suggestions and encouragement. I would also like to thank Bill Manegold for being a helpful and forgiving alpha tester.

REFERENCES

[1] H. J. Shin, J. Lee, S. Y. Shin *et al.*, "Computer Puppetry: An Importance-Based Approach," *ACM Transactions on Graphics,* vol. 20, no. 2, pp. 67-94, April 2001, 2001.

[2] C. Rose, B. Bodenheimer, and M. F. Cohen, "Verbs and Adverbs: Multidimensional Motion Interpolation Using Radial Basis Functions," *IEEE Computer Graphics and Applications,* vol. 18, pp. 32--40, 1998.

[3] A. Bruderlin, and L. Williams, "Motion signal processing," *Computer Graphics (Proceedings of SIGGRAPH 95)*, pp. 97-104, August 1995, 1995.

[4] A. Witkin, and Z. Popović, "Motion warping," *Computer Graphics (Proceedings of SIGGRAPH 95),* vol. 17, pp. 105–108, August 1995.

[5] C. W. Reynolds, "Flocks, Herds, and Schools: A Distributed Behavioral Model," *Computer Graphics (Proceedings of SIGGRAPH 87),* vol. 21, no. 4, pp. 25-34, July 1987, 1987.

[6] K. Sims, "Evolving virtual creatures," *Computer Graphics (Proceedings of SIGGRAPH 94)*, pp. 15–22, 1994.

[7] X. Tu, and D. Terzopoulos, "Artificial Fishes: Physics, Locomotion, Perception, Behavior," *Computer Graphics (Proceedings of SIGGRAPH 94)*, pp. 43-50, July 1994.

[8] F. Multon, L. France, Marie-Paule *et al.*, "Computer animation of human walking: a survey," *The Journal of Visualization and Computer Animation,* vol. 10, no. 1, pp. 39-54, 1999.

[9] D. Zeltzer, "Motor Control Techniques for Figure Animation," *IEEE Computer Graphics and Applications,* vol. 2, no. 9, pp. 53-59, 1982.

[10] R. Boulic, and R. Mas, "Hierarchical kinematic behaviors for complex articulated figures," *Interactive computer animation*, pp. 40-70: Prentice-Hall, Inc., 1996.

[11] M. H. Raibert, and J. K. Hodgins, "Animation of Dynamic Legged Locomotion," *Computer Graphics (Proceedings of SIGGRAPH 91),* vol. 25, no. 4, pp. 349-358, July 1991, 1991.

[12] M. McKenna, and D. Zeltzer, "Dynamic Simulation of Autonomous Legged Locomotion," *Computer Graphics (Proceedings of SIGGRAPH 90),* vol. 24, no. 4, pp. 29-38, August 1990, 1990.

[13] K. Hase, K. Miyashita, S. Ok *et al.*, "Human gait simulation with a neuromusculoskeletal model and evolutionary computation," *The Journal of Visualization and Computer Animation,* vol. 14, no. 2, pp. 73-92, May 2003, 2003.

[14] A. Bruderlin, and T. W. Calvert, "Goal-Directed, Dynamic Animation of Human Walking," *Computer Graphics,* vol. 23, no. 3, pp. 233-242, 1989.

[15] F. Devillers, S. Donikian, F. Lamarche *et al.*, "A programming environment for behavioural animation," *The Journal of Visualization and Computer Animation,* vol. 13, pp. 263-274, 2002.

[16] B. M. Blumberg, and T. A. Galyean, "Multi-level direction of autonomous creatures for real-time virtual environments," *Computer Graphics (proceedings of SIGGRAPH 95)*, pp. 47–54, August 1995.

[17] S. Regelous, "Massive," Massive Software, Inc., 2001.

[18] L. A. Zadeh, G. J. Klir, and B. Yuan, *Fuzzy Sets, Fuzzy Logic, Fuzzy Systems*: World Scientific Press, 1996.

[19] K. Perlin, and A. Goldberg, "Improv: A System for Scripting Interactive Actors in Virtual Worlds," *Computer Graphics,* vol. 30, pp. 205-216, 1996.

[20] N. I. Badler, C. B. Phillips, and B. L. Webber, *Simulating Humans: Computer Graphics Animation and Control*: Oxford University Press, 1993.

[21] N. I. Badler, M. S. Palmer, and R. Bindiganavale, "Animation control for real-time virtual humans," *Commun. ACM,* vol. 42, no. 8, pp. 64-73, 1999.

[22] C. B. Phillips, J. Zhao, and N. I. Badler, "Interactive real-time articulated figure manipulation using multiple kinematic constraints," in Proceedings of the 1990 symposium on Interactive 3D graphics, Snowbird, Utah, United States, 1990.

[23] T. Marcus, M. Stacy, N. M. Andrew *et al.*, "SmartBody: behavior realization for embodied conversational agents," in Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems - Volume 1, Estoril, Portugal, 2008.

[24] Natural Motion Inc., "Euphoria:core motion synthesis library," Natural Motion, Inc., 2006.

[25] Rockstar North, and Rockstar Toronto, "Grand Theft Auto IV," Rockstar Games, 2008.

[26] LucasArts, "Star Wars: The Force Unleashed," LucasArts, 2008.

[27] Natural Motion Inc., *Endorphin 2.6 User Guide*, Natural Motion, Inc., 2006.

[28] Valve Corporation, "Half-Life 2," Vivendi Universal Games, 2004.

[29] Epic Games, "Unreal Tournament 3," Midway Games, 2007.

[30] M. Mateas, and A. Stern, "Façade," 2005.

[31] R. Arkin, *Behavior-Based Robotics*, Cambridge: MIT Press, 1998.

[32] B. Blumberg, "Old Tricks, New Dogs: Ethology and Interactive Creatures," Media Lab, Massachusetts Institute of Technology, Cambridge, 1996.

[33] M. M. Williamson, "Oscillators and Crank Turning: Exploiting Natural Dynamics with a Humanoid Robot Arm," *Philosophical Transactions of the Royal Society: Mathematical, Physical and Engineering Sciences,* vol. 361, no. 1811, pp. 2207-2223, 2003.

[34] T. Jakobsen, "Advanced Character Physics," in Game Developer's Conference, San Jose, 2001.

[35] K. Perlin, "Unpublished work on bipedal walking," 2003.

[36] K. Perlin, "Real time responsive animation with personality," *IEEE Transactions on Visualization and Computer Graphics,* vol. 1, no. 1, pp. 5-15, 1995.

[37] Microsoft, "XNA Game Studio 3.0," Microsoft Corporation, 2008.

[38] Free Software Foundation. "Lesser Gnu Public License, v.3," http://www.gnu.org/licenses/.

[39] J. Bates, "The Role of Emotion in Believable Agents," *Communications of the ACM,* vol. 37, no. 7, pp. 122-125, 1994.

[40] Havok, "The Havok 5.5 Physics Engine," Havok Inc., 2008.

[41] R. Smith, *Open Dynamics Engine v.0.5 User Guide*, 2006.

[42] S. Kuriyama, Y. Kurihara, Y. Irino *et al.*, "Physiological gait controls with a neural pattern generator," *The Journal of Visualization and Computer Animation,* vol. 13, no. 2, pp. 107-119, 2002.

[43] I. Millington, *Game physics engine development*, Amsterdam ; Boston: Morgan Kaufmann Publishers, 2007.

[44] IO Interactive, "Hitman: Codename 47," Eidos Interactive, 2000.

[45] L. Verlet, "Computer 'Experiments' on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules," *Physical Review,* vol. 159, no. 1, pp. 98-103, July 1967, 1967.

[46] M. Müller, B. Heidelberger, M. Hennix *et al.*, "Position based dynamics," *J. Vis. Comun. Image Represent.,* vol. 18, no. 2, pp. 109-118, 2007.

[47] J. Cleese, G. Chapman, E. Idle *et al.*, "Self-defense Against Fresh Fruit," *Monty Python's Flying Circus, series 1, episode 4, Owl Stretching Time*, British Broadcasting Corporation, 1969.

[48] J. Bowlby, *Attachment and Loss*, New York,: Basic Books, 1969.

[49] A. Jhala, and R. M. Young, "A discourse planning approach to cinematic camera control for narratives in virtual environments," in 20th National Conference on Artificial Intelligence (AAAI-05), Pittsburgh, PA, 2005.

[50] M. S. El-Nasr, and I. Horswill, "Automating lighting design for interactive entertainment," *ACM Computers and Entertainment,* vol. 2, no. 2, April/June, 2004.