invariant. However, this multiplication may be detectable in our finite implementation: some copies on the edge of the tessellation may appear or disappear. In the ideal implementation (requiring more computer power) these copies are barely visible, either being too small or too foggy.

The alternative, to translate the tessellation to follow the observer, quickly leads in the hyperbolic case to severe numerical problems in the action in the group elements. The result is that the fourth, "homogeneous" coordinate of the transformed vertices grows exponentially large and the dehomogenization operation loses precision. This is avoided by the method outlined above.

4.4 Stereo

Modeling stereo vision presented challenges in the non-euclidean case. We first describe the more familiar solution available in the euclidean setting. The observer and the CAVE have a fixed physical reality which should be mirrored in the models we apply to them. That is, the model coordinates for the navigator are the same as the model coordinates of the CAVE. In particular, the interocular separation of the observer stays at a fixed ratio to the CAVE size. We found empirically that an interocular distance of about 1/100 that of the diagonal of the CAVE is small enough to assure fusion. This translates to a distance of about 2 inches, roughly corresponding to human anatomy. In euclidean space, making the CAVE larger is equivalent to shrinking the scene while keeping the CAVE a constant size. However, in non-euclidean settings, this equivalence no longer holds! In these spaces, there is no change of size without also changing shape. Consequently, it is the CAVE and observer that changes size (and shape!) while the scene remains the same. Of course there is no guarantee of fusion; it may become difficult if the observer in H^3 becomes too large while standing near the fixed geometry; but the danger is no different from the physically observed difficulty of fusing stereo when you move your hand closer to your eyes in everyday life.

The pair of images for the stereo effect is produced by rendering each eye separately as described above by hyperbolically translating the scene to locate the given eye at the origin.

4.5 Efficiency measures

To maintain the frame rate required in VR, we needed to disable the software lighting and shading for non-euclidean scenes (OOGL does lighting in software because of the different metrics of the non-euclidean geometries). We kept the model of the tessellation simple – a wireframe, with simple, solid tiles inside. The discrete group software in OOGL automatically culled the copies of the tessellation which lay outside the viewing frustum of a given wall of the CAVE. Also, we kept the number of layers of the tessellation great enough to produce a sense of depth, but small enough to maintain an adequate frame rate.

5 Evaluation

We have combined the discrete group capabilities of OOGL with VR, the only visualization paradigm for an immersive, direct experience of mathematical spaces, to extend the power of interactive 3-d visualization of such spaces. Access to 3-manifolds via a virtual environment is a significant addition to the tools available for mathematical research and education. For example, as pointed out in section 3, GeomCAVE allows direct observation of interesting properties of non-euclidean spaces, such as the right angles of dodecahedra in hyperbolic space. GeomCAVE immediately makes features of OOGL available in VR, such as a collection of geometric models and discrete group operations. Thus, a mathematician who has built a manifold for viewing in maniview would be able to also explore it in GeomCAVE.

6 Further work

- Implement mixed mode navigation in H^3 (see conclusion of Section 4.2).
- Add more features of maniview:
  - Control over the size and shape of the Dirichlet domain.
  - Control over the depth of the tessellation.
  - As hardware improves, re-activate the software shading and fog effects.
- More sophisticated tools for mathematicians:
  - Connections with existing manifold software (such as *snappea* ([5]).
  - Finer interactive control of the discrete group: selecting subgroups, use of color, deformation of the group.
  - Simulation of dynamical systems in non-euclidean spaces.
  - Extend the coverage to the other five Thurston geometries.
- Experiment with audio tessellation along with the geometric data. The resulting echo patterns could distinguish differently-shaped manifolds.

7 Acknowledgements

We would like to extend special thanks to Stuart Levy of the Geometry Center for his help. Thanks are also due to Mark Phillips and Tamara Munzner, also of the Geometry Center, as well as Louis Kauffman, of the University of Illinois at Chicago.

REFERENCES

[1] Callahan, M.J., Hoffman, D. and Hoffman, J.T. Computer Graphics Tools for the Study of Minimal Surfaces. *Communications of the Association for Computing Machinery 31*, 6 (1988), 648-661.

[2] Cruz-Neira, Carolina, Sandin, Daniel J., DeFanti, Thomas A., Kenyon, Robert V. and Hart, John C. The CAVE: Audio Visual Experience Automatic Virtual Environment. *Communications of the Association for Computing Machinery 35*, 6 (June, 1992), 65-72.

[3] Gunn, Charlie. Discrete Groups and Visualization of Three Dimensional Manifolds. *Computer Graphics 27* (July, 1993), 255-262. Proceedings of SIGGRAPH 1993.

[4] Thurston, William. Three Dimensional Manifolds, Kleinian Groups and Hyperbolic Geometry. *BAMS 19* (1982), 417-431.

[5] Weeks, Jeff. snappea — a MacIntosh application for computing 3-manifolds". (available from ftp@geom.umn.edu).

Tracking a Turbulent Spot in an Immersive Environment

*David C. Banks, Institute for Computer Applications in Science and Engineering

‡Michael Kelley, Information Sciences Institute

ABSTRACT

We describe an interactive, immersive 3D system called *Tracktur*, which allows a viewer to track the development of a turbulent flow. *Tracktur* displays time-varying vortex structures extracted from a numerical flow simulation. The user navigates the space and probes the data within a windy 3D landscape. In order to sustain a constant frame rate, we enforce a fixed polygon budget on the geometry. In actual use by a fluid dynamicist, *Tracktur* has yielded new insights into the transition to turbulence of a laminar flow.

1 Introduction

Simulating the evolution of a turbulent spot has consumed thousands of CPU hours (on a Cray 2, Cray YMP, and YMP C-90 over the course of 2.5 calendar years) [1]. We wish to animate 230 time steps produced by the simulation, which are archived as hundreds of gigabytes of data. How does one visualize this large amount of time-varying data at interactive speeds?

A new technique for locating vortices in an unsteady flow [2] compresses the volumetric flow-data by a factor of more than a thousand. This amount of compression seemed to promise interactive visualization of a massive time-varying dataset. We therefore developed a visualization system, *Tracktur*, that uses the compressed vortex representation to help track the development of a turbulent flow [3]. *Tracktur* uses a graphics workstation, 3D tracking, and a stereoscopic display to create a virtual 3D environment populated by time-varying vortex tubes.

2 The Interactive Environment

Our target user is the theoretical flow physicist who produced the time-varying dataset. From his perspective, the significant features of the simulation include the flat plate, the fluid flowing over it, the vortex structures, and the units of the computational domain (both spatial and temporal). The combination of a plane with a continual flow over it suggested to us a windy landscape.

*ICASE, Mail Stop 132C, NASA Langley Research Center, Hampton, Virginia 23681. 804/864-2194 (banks@icase.edu).

‡Information Sciences Institute, 4350 N. Fairfax Drive, Suite 400, Arlington, VA 22203. 703/243-9422 (kelleym@arpa.mil).

One of our early design decisions was to make generous use of texture maps to enrich the virtual world. A grid-texture was an obvious choice for the ground plane, with stenciled textures added to denote streamwise units of the domain. To indicate the free-stream velocity, we animate a cloud-texture on two distant walls. Textures denote the upstream and downstream directions. Surrounded by a textured landscape, a viewer is given persistent reminders of the spatial context he is operating within. The 3D widgets in the environment are also textured to eliminate the cartoon quality that constant-colored polygons convey.

In an actual wind-tunnel experiment, the vortex structures would be only millimeters in size and the free-stream velocity would be about 30 meters per second. The lifetime of the turbulent spot would be less than a second. *Tracktur* displays the 3D animation at more human scales: the geometry is larger and the simulation lasts longer, each by about three orders of magnitude.

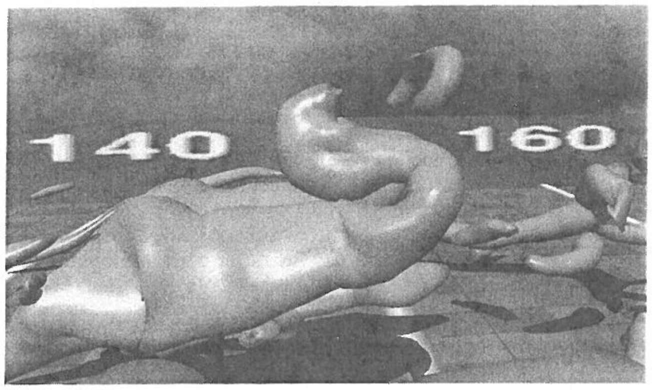
We want to help the scientist comprehend the spatial evolution of a turbulent spot; since the spot convects downstream, we let the viewer be convected along with it to keep it in the field of view. Widgets are convected downstream with the viewer to remain within reach. A time-slider advances to mirror the current time step in the animation; alternatively, the viewer can set the current time step by adjusting the slider. Shadows on the ground plane provide a depth cue at only a small penalty in performance [4]. The viewer can select surface, wire-frame, or fat-line representations of the geometry. The fat-line segments (through the core of the vortices) are given widths to match the thickness of the tube and are illuminated as one-dimensional fibers [5] in order to convey shape from shading.

We also want to permit routine measurements of flow quantities. The viewer is given a data probe – a ray emanating from the pointing device in the virtual environment. *Tracktur* locates the nearest point on a vortex core to the probe ray, then displays attributes (such as spatial position of the point) in a pop-up panel.

3 3D Toolkits

Tracktur is constructed from several component libraries, including public-domain toolkits. The Minimal Reality toolkit [6] provides the basis of a through-the-window interface that uses stereoscopic display and 3D tracking for the head and hand. The CAVE version of the application [7] uses code developed by the Electronic Visualization Laboratory [8].

We developed a custom toolkit to implement 3D menus (using Hershey fonts), buttons, and sliders. We also developed a calibration tool for the 3D trackers to determine the proper matrix transforms. The user interactively aligns coordinate axes (displayed on the screen) to establish the correct rotation matrix. The various transformations are written to a file and need not be recomputed unless the equipment is moved.



A backward-tilted S-shaped vortex head that develops in the late stages of transition from a laminar flow to a turbulent spot.

4 The Fixed Polygon Budget

A difficult aspect of developing an interactive system is preserving a fixed frame rate. Our scene-updates are typically dominated by the time spent drawing the vortex tubes, so we budget a fixed number of polygons with which to model them. The turbulent spot increases in geometric complexity as the simulation progresses: a single vortex tube at time 28 develops into about 150 tubes at time 221. An SGI Onyx with RealityEngine2 graphics sustains about 15 frames per second with a fixed count of 9000 polygons.

In the early stages of the simulation, the polygon budget allows a finer resolution than we have computed. We therefore resample the vortex skeleton at a higher spatial resolution in order to exhaust the supply of polygons. But in the late stages of the simulation it is imperative to dole out the polygons in a miserly fashion. The vortex skeletons are down-sampled according to a set of heuristics designed to preserve significant geometric features. The resampling works as a filter on the original skeletal representation of the vortex core. The first sample-point is always retained. After a point is retained, subsequent points along the skeleton are rejected unless any of the following hold:

- the arclength exceeds a threshold;
- the integrated curvature exceeds a threshold;
- the radius of the cross-section changes quickly.

Sometimes a vortex skeleton enters a small spiral from which it never exits. To guard against wasted samples, we reject points on the skeleton where the ratio of the skeleton's radius to its radius of curvature exceeds a threshold (we use the constant 0.7). These heuristics maintain a reasonable amount of geometric detail at the late stages of the simulation.

5 What Has Been Learned

The scientist who generated the dataset (Dr. Bart Singer) agreed to use the system to study how a turbulent spot develops. He has learned two new things about the evolution of the turbulent spot. In order to place them in their context, we give a brief descriptive summary of the spot's development.

First, Singer discovered a backwards-tilted S-shaped vortex head in the late stages of transition (see figure). The vortex is similar in shape to a structure seen in experimental data for a similar flow. Singer had not observed this feature in his dataset until he used our system. Evidently, the interactivity permitted him to select the right combination of a particular viewpoint and a particular time step. This could, in principal, have been discovered with the visualization system he was a accustomed to using, but its more limited interactivity made the feature much harder to find.

Secondly, the visualization system gave Singer his first view of the dynamic behavior of "necklace" vortices, which define the outer extent of the turbulent spot. They eventually shred into pieces, curling into horseshoe and hairpin vortices. Without *Tracktur*, Singer had been unable to track the necklace vortices through their entire history. These findings are initial evidence that the system can assist in the research task.

6 Conclusions

Visualization tools can certainly *communicate* research results, but it is not yet clear how well they help *produce* research results. We have created an interactive 3D visualization system, called *Tracktur*, and put it into the hands of the scientist. *Tracktur* provides a textured environment for examining the onset of turbulence. The viewer can navigate through the landscape and interact with a turbulent spot through 3D menus, buttons, sliders, and a data probe. In the hands of a fluid scientist, the system has yielded new insights into the development of a turbulent spot.

Acknowledgments

This work was supported under NASA contract No. NAS1-19480. We thank Bill von Ofenheim and the Data Visualization Lab at NASA Langley Research Center for use of their stereo glasses. We thank Jonathan Shade at the San Diego Supercomputer Center for help in creating transparent texture maps.

Bibliography

[1] Singer, Bart A. and Ron Joslin, "Metamorphosis of a hairpin vortex into a young turbulent spot." *Physics of Fluids A*, Vol. 6, No. 11 (Nov. 94).

[2] Banks, David C. and Bart A. Singer, "Vortex Tubes in Turbulent Flows: Identification, Representation, Reconstruction." *Proceedings of Visualization '94.*

[3] "The Tracktur Home Page," World Wide Web URL http://www.icase.edu/~banks/tracktur/vortex/doc/tracktur.html.

[4] Blinn, Jim, "Me and My (Fake) Shadow." *IEEE Computer Graphics & Applications* (Jim Blinn's Corner), January 1988, pp. 82-86.

[5] Banks, David C., "Illumination in Diverse Codimensions." *Proceedings of SIGGRAPH '94* (Orlando, Florida, July 24-29, 1994). In *Computer Graphics* Proceedings, Annual Conference Series, 1994, ACM SIGGRAPH, New York, pp. 327-334.

[6] "MR Toolkit," World Wide Web URL http://web.cs.ualberta.ca/~graphics/MRToolkit.html.

[7] Banks, David C., "The Onset of Turbulence in a Shear Flow Over a Flat Plate." [Demonstration] SIGGRAPH '94 VROOM Exhibit. In *Visual Proceedings: The Art and Interdisciplinary Programs of SIGGRAPH 94*, Computer Graphics Annual Conference Series, 1994, ACM SIGGRAPH, New York, p. 235. Also in "Fluid Mechanics," World Wide Web URL http://www.ncsa.uiuc.edu/EVL/docs/VROOM/HTML/PROJECTS/23Banks.html.

[8] "CAVE User's Guide," World Wide Web URL http://www.ncsa.uiuc.edu/EVL/docs/html/CAVEGuide.html.

Behavioral Control for Real-Time Simulated Human Agents

John P. Granieri, Welton Becket,
Barry D. Reich, Jonathan Crabtree, Norman I. Badler

Center for Human Modeling and Simulation
University of Pennsylvania
Philadelphia, Pennsylvania 19104-6389
granieri/becket/reich/crabtree/badler@graphics.cis.upenn.edu

Abstract

A system for controlling the behaviors of an interactive human-like agent, and executing them in real-time, is presented. It relies on an underlying model of continuous *behavior*, as well as a discrete scheduling mechanism for changing behavior over time. A multiprocessing framework executes the behaviors and renders the motion of the agents in real-time. Finally we discuss the current state of our implementation and some areas of future work.

1 Introduction

As rich and complex interactive 3D virtual environments become practical for a variety of applications, from engineering design evaluation to hazard simulation, there is a need to represent their inhabitants as purposeful, interactive, human-like agents.

It is not a great leap of the imagination to think of a product designer creating a virtual prototype of a piece of equipment, placing that equipment in a virtual workspace, then populating the workspace with virtual human operators who will perform their assigned tasks (operating or maintaining) on the equipment. The designer will need to instruct and guide the agents in the execution of their tasks, as well as evaluate their performance within his design. He may then change the design based on the agents' interactions with it.

Although this scenario is possible today, using only one or two simulated humans and scripted task animations [3], the techniques employed do not scale well to tens or hundreds of humans. Scripts also limit any ability to have the human agents react to user input as well as each other during the execution of a task simulation. We wish to build a system capable of simulating many agents, performing moderately complex tasks, and able to react to external (either from user-generated or distributed simulation) stimuli and events, which will operate in near real-time. To that end, we have put together a system which has the beginnings of these attributes, and are in the process of investigating the limits of our approach. We describe below our architecture, which employs a variety of known and previously published techniques, combined together in a new way to achieve near real-time behavior on current workstations.

We first describe the machinery employed for behavioral control. This portion includes perceptual, control, and motor components. We then describe the multiprocessing framework built to run the behavioral system in near real-time. We conclude with some internal details of the execution environment. For illustrative purposes, our example scenario is a pedestrian agent, with the ability to locomote, walk down a sidewalk, and cross the street at an intersection while obeying stop lights and pedestrian crossing lights.

2 Behavioral Control

The behavioral controller, previously developed in [4] and [5], is designed to allow the operation of parallel, continuous behaviors each attempting to accomplish some function relevant to the agent and each connecting sensors to effectors. Our behavioral controller is based on both potential-field reactive control from robotics [1, 10] and behavioral simulation from graphics, such as Wilhelms and Skinner's implementation [20] of Braitenberg's *Vehicles* [7]. Our system is structured in order to allow the application of optimization learning [6], however, as one of the primary difficulties with behavioral and reactive techniques is the complexity of assigning weights or arbitration schemes to the various behaviors in order to achieve a desired observed behavior [5, 6].

Behaviors are embedded in a network of *behavioral nodes*, with fixed connectivity by links across which only floating-point messages can travel. On each simulation step the network is updated synchronously and without order dependence by using separate load and emit phases using a simulation technique adapted from [14]. Because there is no order dependence, each node in the network could be on a separate processor, so the network could be easily parallelized.

Each functional behavior is implemented as a subnetwork of behavioral nodes defining a path from the geometry database of the system to calls for changes in the database. Because behaviors are implemented as networks of simpler processing units, the representation is more explicit than in behavioral controllers where entire behaviors are implemented procedurally. Wherever possible, values that could be used to parameterize the behavior nodes are made accessible, making the entire controller accessible to machine learning techniques which can tune components of a behavior that may be too complex for a designer to manage. The entire network comprising the various sub-behaviors acts as the controller for the agent and is referred to here as the *behavior net*.

There are three conceptual categories of behavioral nodes employed by behavioral paths in a behavior net:

perceptual nodes that output more abstract results of perception than what raw sensors would emit. Note that in a simulation that has access to a complete database of the simulated world, the job of the perceptual nodes will be to realistically limit perception, which is perhaps opposite to the function of perception in real robots.

motor nodes that communicate with some form of motor control for the simulated agent. Some motor nodes enact changes directly on the environment. More complex motor behaviors, however, such as the *walk motor node* described below, schedule a motion (a step) that is managed by a separate, asynchronous execution module.

control nodes which map perceptual nodes to motor nodes usually using some form of negative feedback.

This partitioning is similar to Firby's partitioning of continuous behavior into active sensing and behavior control routines [10], except that motor control is considered separate from negative feedback control.

2.1 Perceptual Nodes

The perceptual nodes rely on simulated sensors to perform the perceptual part of a behavior. The sensors access the environment database, evaluate and output the distance and angle to the target or targets. A sampling of different sensors currently used in our system is described below. The sensors differ only in the types of things they are capable of detecting.

Object: An object sensor detects a single object. This detection is global; there are no restrictions such as visibility limitations. As a result, care must be taken when using this sensor: for example, the pedestrian may walk through walls or other objects without the proper avoidances, and apparent realism may be compromised by an attraction to an object which is not visible. It should be noted that an object sensor always senses the object's current location, even if the object moves. Therefore, following or pursuing behaviors are possible.

Location: A location sensor is almost identical to an object sensor. The difference is that the location is a unchangeable point in space which need not correspond to any object.

Proximity: A proximity sensor detects objects of a specific type. This detection is local: the sensor can detect only objects which intersect a sector-shaped region roughly corresponding to the field-of-view of the pedestrian.

Line: A line sensor detects a specific line segment.

Terrain: A terrain sensor, described in [17], senses the navigability of the local terrain. For example, the pedestrian can distinguish undesirable terrain such as street or puddles from terrain easier or more desirable to negotiate such as sidewalk.

Field-of-View: A field-of-view sensor, described in [17], determines whether a human agent is visible to any of a set of agents. The sensor output is proportional to the number of agents' fields-of-view it is in, and inversely proportional to the distances to these agents.

2.2 Control Nodes

Control nodes typically implement some form of negative feedback, generating outputs that will reduce perceived error in input relative to some desired value or limit. This is the center of the reactivity of the behavioral controller, and as suggested in [9], the use of negative feedback will effectively handle noise and uncertainty.

Two control nodes have been implemented as described in [4] and [5], *attract* and *avoid*. These loosely model various forms of *taxis* found in real animals [7, 11] and are analogous to proportional servos from control theory. Their output is in the form of a recommended new velocity in polar coordinates:

Attract An *attract* control node is linked to θ and d values, typically derived from perceptual nodes, and has angular and distance thresholds, t_θ and t_d. The *attract* behavior emits $\Delta\theta$ and Δd values scaled by linear weights that suggest an update that would bring d and θ closer to the threshold values. Given weights k_θ and k_d :

$$\Delta\theta = \begin{cases} 0 & \text{if } -t_\theta \leq \theta \leq t_\theta \\ k_\theta(\theta - t_\theta) & \text{if } \theta > t_\theta \\ k_\theta(\theta + t_\theta) & \text{otherwise} \end{cases}$$

$$\Delta d = \begin{cases} 0 & \text{if } d \leq t_d \\ k_d(d - t_d) & \text{otherwise.} \end{cases}$$

Avoid The *avoid* node is not just the opposite of *attract*. Typically in *attract*, both θ and d should be within the thresholds. With *avoid*, however, the intended behavior is usually to have d outside the threshold distance, using θ only for steering away. The resulting avoid formulation has no angular threshold:

$$\Delta\theta = \begin{cases} 0 & \text{if } d > t_d \\ k_\theta(\pi - \theta) & \text{if } d \leq t_d \text{ and } \theta \geq 0 \\ k_\theta(-\pi - \theta) & \text{otherwise} \end{cases}$$

$$\Delta\theta = \begin{cases} 0 & \text{if } d > t_d \\ k_d(t_d - d) & \text{otherwise.} \end{cases}$$

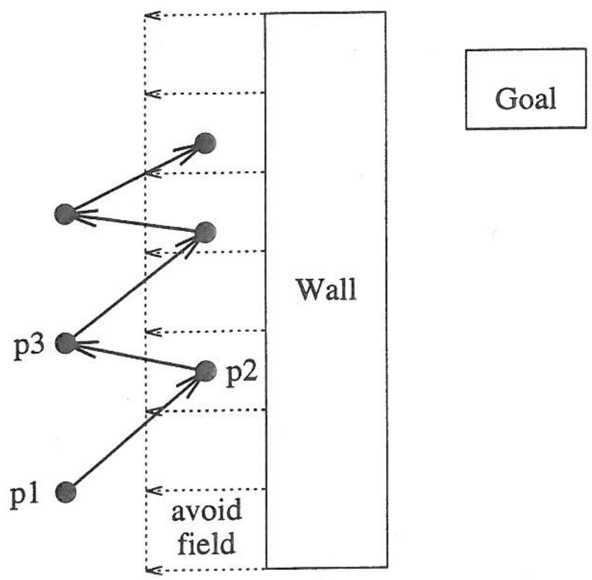


Figure 1: Sawtooth path due to potential field discontinuities

2.3 Motor Nodes

Motor nodes for controlling non-linked agents are implemented by interpreting the Δd and $\Delta \theta$ values emitted from control behaviors as linear and angular adjustments, where the magnitude of the implied velocity vector gives some notion of the urgency of traveling in that direction. If this velocity vector is attached directly to a figure so that requested velocity is mapped directly to a change in the object's position, the resulting agent appears jet-powered and slides around with infinite damping as in Wilhelms and Skinner's environment [20].

2.3.1 Walking by sampling potential fields

When controlling agents that walk, however, the motor node mapping the velocity vector implied by the outputs of the control behaviors to actual motion in the agent needs to be more sophisticated. In a walking agent the motor node of the behavior net *schedules* a step for an agent by indicating the position and orientation of the next footstep, where this decision about where to step next happens at the end of every step rather than continuously along with motion of the agent. The velocity vector resulting from the blended output of all control nodes could be used to determine the next footstep; however, doing so results in severe instability around threshold boundaries. This occurs because we allow thresholds in our sensor and control nodes and as a result the potential field space is not continuous. Taking a discrete step based on instantaneous information may step across a discontinuity in field space. Consider the situation in Fig. 1 where the agent is attracted to a goal on the opposite side of a wall and avoids the wall up to some threshold distance. If the first step is scheduled at position p_1, the agent will choose to step directly toward the goal and will end up at p_2. The agent is then well within the threshold distance for walls and will step away from the wall and end up at p_3, which is outside the threshold. This process then repeats until the wall

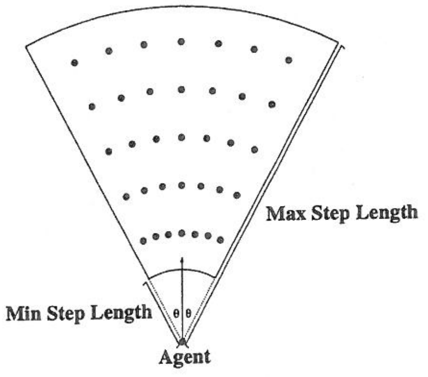


Figure 2: The fan of potential foot locations and orientations

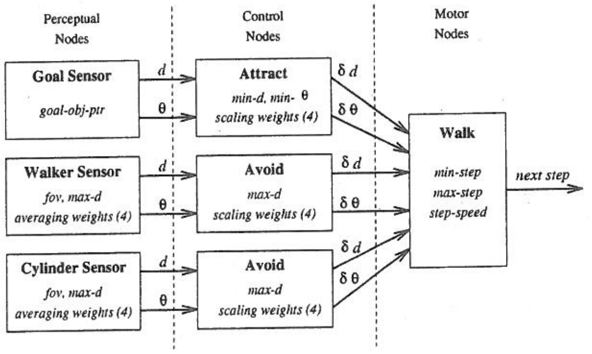


Figure 3: An example behavior net for walking

is cleared, producing an extremely unrealistic sawtooth path about the true gradient in the potential field.

To eliminate the sawtooth path effect, we sample the value of the potential field implied by the sensor and control nodes in the space in front of the agent and step on the location yielding the minimum sampled 'energy' value. We sample points that would be the agent's new location if the agent were to step on points in a number of arcs within a fan in front of the agent's forward foot. This fan, shown in Fig. 2, represents the geometrically valid foot locations for the next step position under our walking model. This sampled step space could be extended to allow side-stepping or turning around which the agent can do [3], though this is not currently accessed from the behavior system described in this paper. For each sampled step location, the potential field value is computed at the agent's new location, defined as the average location and orientation of the two feet.

2.4 An example behavior net

The example behavior net in Fig. 3 specifies an overall behavior for walking agents that head toward a particular goal object while avoiding obstacles (cylinders in this case) and each other. The entire graph is the *behavior net*, and each path from perception to motor output is considered a *behavior*. In this example there are three behaviors: one connecting a goal sensor to an attraction controller and then to the walk node (a goal-attraction behavior), another connecting a sensor detecting proximity of other walking agents to an avoidance controller and then to the walk node (a walker-avoidance behavior), and a final behavior connecting a cylinder proximity sensor to an avoidance behavior and then to the walk node (a cylinder-avoidance behavior).

Each node has a number of parameters that determine its behavior. For example, the walker sensor and the cylinder sensor nodes have parameters that indicate how they will average all perceived objects within their field of view and sensing distance into a single abstract object. The Attract and Avoid nodes have scaling weights that determine how much output to generate as a function of current input and the desired target values.

The walk motor behavior manages the sampling of the potential field by running data through the perceptual and control nodes with the agent pretending to be in each of the sampled step locations. The walk node then schedules the next step by passing the step location and orientation to the execution module.

Note that this example has no feedback, cross-talk, or inhibition within the controller, though the behavioral controller specification supports these features [5]. Although this example controller itself is a feed-forward network, it operates as a closed-loop controller when attached to the agent because the walk node's scheduling of steps affects the input to the perceptual nodes.

Our use of *attract* and *avoid* behaviors to control groups of walking agents may appear on the surface like Ridsdale's use of *hot* and *cold* tendencies to control agents in his *Director's Apprentice* system [18]. However, his system was not reactive and on-line as our behavioral controller is, it did not limit perception of agents, it had no structured facilities for tuning behavior parameters, and it did not take advantage of developments in reactive control and behavioral simulation. His system focused on the use of an expert system to schedule human activity conforming to stage principles and used hot and cold tendencies to manage complex human behavior and interaction. We limit the use of behaviors to reactive navigation and path-planning, using parallel transition networks rather than one large expert system to schedule events, and we look to symbolic planning systems based on results in cognitive science, such as [3, 8, 16], to automate high-level human behavior and complex human interactions.

3 Parallel Automata

Parallel Transition Networks (PaT-Nets) are transition networks that run in parallel with the behavior net, monitor it, and edit it over time [8]. They are a mechanism for scheduling arbitrary actions and introducing decision-making into the agent architecture. They monitor the behavior net (which may be thought of as modeling low level instinctive or reflexive behavior) and make decisions in special circumstances. For example, the agent may get caught in a dead-end or other local minimum. PaT-Nets recognize situations such as these, override the "instinctive" behavior simulation by reconfiguring connectivity and modifying weights in the behavior net, and then return to a monitoring state.

In our pedestrian example we combine object and location sensors (in perceptual nodes) with *attract* control nodes, and proximity and line sensors (in perceptual nodes) with *avoid* control nodes. Pedestrians are attracted to street corners and doors, and they avoid each other, light poles, buildings, and the street except at crosswalks.

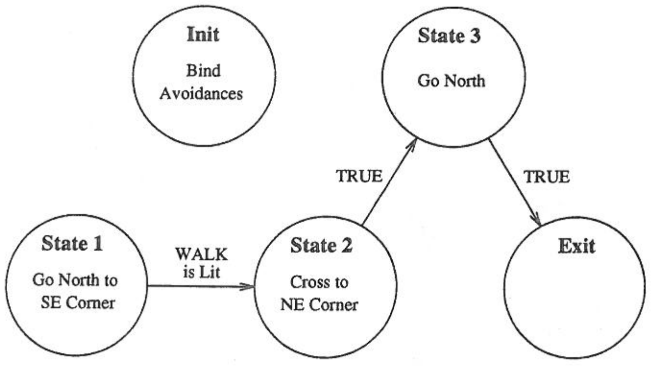


Figure 4: `North-net`: A sample `ped-net` shown graphically

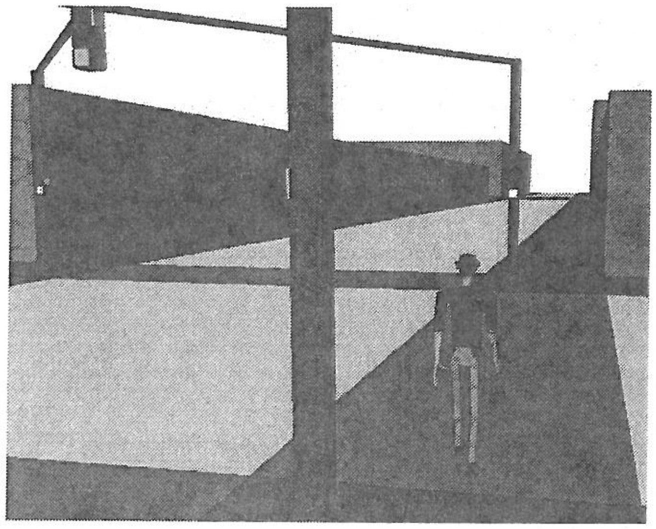


Figure 5: A pedestrian crossing the street

We use PaT-Nets in several different ways. `Light-nets` control traffic lights and `ped-nets` control pedestrians. `Light-nets` cycle through the states of the traffic light and the *walk* and *don't walk* signs.

Fig. 4 is a simple `ped-net`, a `north-net`, which moves a pedestrian north along the eastern sidewalk through the intersection. Initially, avoidances are bound to the pedestrian so that it will not walk into walls, the street, poles, or other pedestrians. The avoidances are always active even as other behaviors are bound and unbound. In State 1 an attraction to the southeast corner of the intersection is bound to the pedestrian. The pedestrian immediately begins to walk toward the corner avoiding obstacles along the way. When it arrives the attraction is unbound, the action for State 1 is complete. Nothing further happens until the appropriate walk light is lit. When it is lit, the transition to State 2 is made and action *Cross to NE Corner* is executed. The agent crosses the street. Finally, the agent heads north.

Fig. 5 shows a pedestrian controlled by a `north-net`. The transition to State 2 was just made so the pedestrian is crossing the street at the crosswalk.

4 Real-Time Simulation Environment

The run-time simulation system is implemented as a group of related processes, which communicate through shared memory. The system is broken into a minimum of 5 processes, as shown in Fig. 6. The system relies on IRIS Performer [19] for the general multiprocessing framework. Synchronization of all processes, via spin locks and video clock routines, is performed in the CONTROL process. It is also the only process which performs the edits and updates to the run-time visual database. The CULL and DRAW processes form a software rendering pipeline, as described in [19]. The pipeline improves overall rendering throughput while increasing latency, although the two frame latency between CONTROL and DRAW is not significant for our application. Our CONTROL process is equivalent to the APP process in the Performer framework. We have used this framework to animate multiple real-time human figures [12].

4.1 CONTROL Process

The CONTROL process runs the main simulation loop for each agent. This process runs the PaT-Nets, and underlying behavior net for each agent. While each agent has only one behavior net, they may have several PaT-Nets running, which sequence the parameters and connectivity of the nodes in the behavior net over time (as shown in Fig. 6).

By far the costliest computation in the CONTROL process, for the behaviors modeled in this example application, is the evaluation of the **Walk** motor node in the behavior net, and specifically the selection of the next foot position. Since this computation is done only once for every footfall, it usually runs only every 15 frames or so (the average step time being about 1/2 second, and average frame rate 30Hz). If the CONTROL process starts running over its allotted frame time, the **Walk** nodes will start reducing the number of points sampled for the next foot position, thereby reducing computation time. The only danger here is described in Section 2.3.1, the potential for a sawtooth path. If many agents are walking at similar velocities, they can all end up computing their next-step locations at the same frame-time, creating a large computation spike which causes the whole simulation to hiccup. (It is visually manifested by the feet landing in one frame, then the swing foot suddenly appearing in mid-stride on the next frame.) We attempt to even out the computational load for the **Walk** motor node evaluation by staggering the start times for each agent, and thereby distributing the computation over about 1/2 second for all agents.

Another computational load in the CONTROL process comes from the evaluation of the conditional expressions in the Pat-Nets, which may occur on every frame of the simulation. They are currently implemented via LISP expressions, so evaluating a condition involves parse and eval steps. In practice, this is fairly fast as we precompile the LISP, but as the PaT-Nets increase in complexity it will be necessary to replace LISP with a higher performance language (i.e. compiled C code). This may remove some of the generality and expressive power enjoyed with LISP.

Another technique employed to improve performanc , when evaluating a large number of Pat-Nets and behavior nets, is to have the CONTROL process spawn copies of itself, with each copy running the behavior of a subset of the agents. This works as long as updates to the visual database are exclusive to each CONTROL process. (In practice this is the case, since the current behavior net for one agent will not edit any parameters for another agent in the visual database.) Of course, the assumption in spawning more processes is that there are available CPUs to run them.

The CONTROL process also provides the outputs of the motor nodes in the behavior net to the MOTION process. These outputs, in the case of the walking behavior, are the position and orientation of the agent's next foot fall. It also evaluates the motion data (joint angles) coming from the MOTION process, and performs the necessary updates to the articulation matrices of the human agent in the visual database.

4.2 SENSE Process

The SENSE process controls and evaluates the simulated sensors modeled in the perceptual nodes of the behavior net. It provides the outputs of the perceptual nodes to the CONTROL process, which uses them for the inputs to the control nodes of the behavior net. The main computational mechanism the sensors employ are intersections of simple geometric shapes (a set of points, lines, frustums or cones) with the visual database, as well as distance computations. This process corresponds to an ISECT process in the Performer framework.

The major performance parameters of this process are the total number of sensors as well as the complexity and organization of the visual database. Since it needs read-only access to the visual database, several SENSE processes may be spawned to balance the load between the number of sensors being computed, and the time needed to evaluate them. (These extra processes are represented by the dotted SENSE process in Fig. 6.) There is a one frame latency between the outputs of the perceptual nodes and the inputs to the control nodes in the behavior net (which are run in the CONTROL process), but this is not a significant problem for our application.

4.3 MOTION Process

Once the agent has sensed its environment and decided on on appropriate action to take, its motion is rendered via real-time motion generators, using a motion system that mixes pre-recorded playback and fast motion generation techniques.

We use an off-line motion authoring tool [2, 13] to create and record motions for our human figures. The off-line system organizes motion sequences into *posture graphs* (directed, cyclic graphs). Real-time motion playback is simply a traversal of the graph in time. This makes the run-time motion generation free from frame-rate variations. The off-line system also records motions for several levels-of-detail (LOD) models of the human figure. (Both the bounding geometry of the figure, as well as the articulation hierarchy (joints) are represented at several levels of detail.) The three levels-of-detail we are using for the human figure are:

1. A 73 joint, 130 DOF, 2000 polygon model, which has articulated fingers and flexible torso, for use in close-up rendering, and fine motor tasks (*Jack*®),

2. A 17 joint, 50 DOF, 500 polygon model, used for the bulk of rendering; it has no fingers, and the flexible torso has been replaced by two joints,

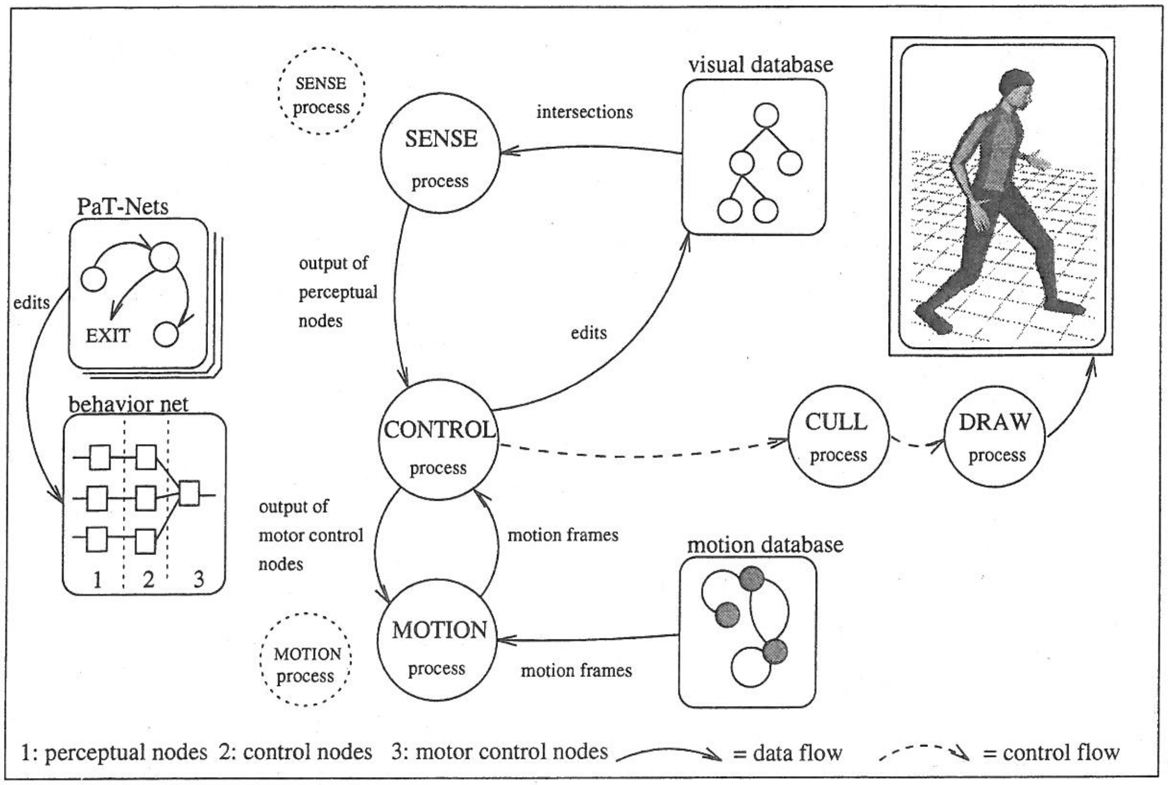


Figure 6: The multiprocessing framework for the real-time behavior execution environment

3. An 11 joint, 21 DOF, 120 polygon model used when the human agent is at a large distance from the camera.

This process produces a frame of motion for each agent, then sleeps until the next frame boundary (the earliest any new motion could be needed). It provides the correct motion frame for the currently active LOD model in the visual database. For certain types of sensors modeled in the perceptual nodes, this process will also be requested to provide a full (highest LOD) update to the visual database, in the case where a lower LOD is currently being used, but a sensor needs to interact with the highest LOD model.

The motion database consists of one copy of the posture graphs and associated motion between nodes of the posture graph. Each transition is stored at a rate of 60HZ, on each LOD model of the human agent. This database is shared by all agents. Only a small amount of private state information is maintained for each agent.

The MOTION process can effectively handle about 10-12 agents at update rates of 30Hz (on a 100MHz MIPS R4000 processor). Since the process only has read-only access to the motion database, we can spawn more MOTION processes if needed for more agents.

4.4 Walking as an example

A MOTION process animates the behaviors specified by an agent's motor nodes by playing back what are essentially pre-recorded chunks of motion. As a time-space tradeoff, this technique provides faster and less variable run-time execution at the cost of additional storage requirements and reduced generality. The interesting issues arise in how we choose a mapping from motor node outputs to this discrete representation; it plays a significant role in determining how realistic the animated agents will be.

The primary motor behavior to be executed is walking. Our full walking algorithm combines kinematics with dynamic balance control and is capable of generating arbitrary curved-path locomotion [15]. In order to reduce computational costs, however, we have not incorporated the algorithm directly into our run-time system. Instead, as implied by the preceding discussion, we record canonical "left" and "right" steps generated by the algorithm (which **is** a component of our off-line motion authoring system) and then play them back in an alternating fashion to produce a continuous walking motion.

The input to the appropriate MOTION process's walking subsystem consists of the specification of the desired next foot position and orientation (for the swing foot). This input is itself already discretized, as the motor node responsible (the **Walk** motor node) for evaluating how desirable it is for the agent to be at particular positions only computes the desirability criteria at a set number of points (in Fig. 2). However, even given that there are only n possibilities for the placement of the swing foot on the next step, this would still require us to record order n^2 possible steps, since the planted foot could be in any one of the n different positions at the start of the step (determined by the **last** step taken) and any one of the n at the end.

Without recording all n^2 distinct steps it is necessary to choose the best match among those that we do record. One of the most important criteria in obtaining realistic results is to minimize foot slippage relative to

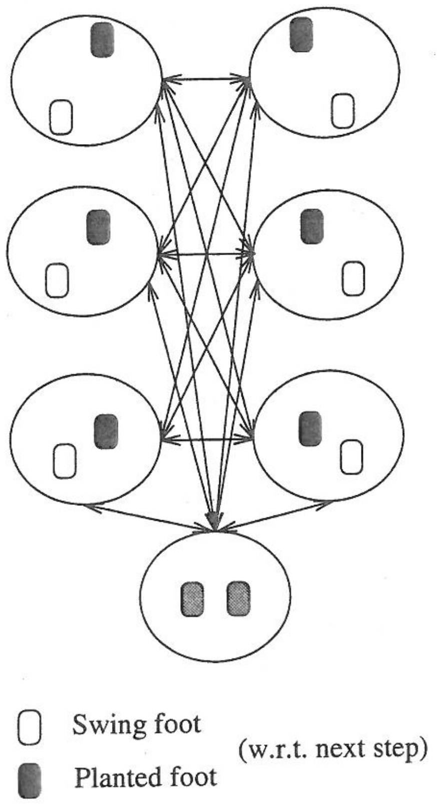


Figure 7: Posture graph for variable step length walking (3 step sizes)

the ground; foot slippage occurs when the pre-recorded movement (in particular its amount and direction) does not match that specified by the **walk** motor node at run time. On the basis that translational foot slippage is far more evident than rotational slippage (at least from our informal observations), we currently adopt an approach in which we record three types of step: short, medium, and long. Turning is accomplished by rotating the agent around his planted foot smoothly throughout the step. Having three step sizes significantly increases the chances of being able to find a close match to the desired step size, and, in fact, the **walk** motor node can be constrained to **only** consider the three arcs of the next foot location fan (see Fig. 2) that correspond exactly to our recorded step sizes. Doing so eliminates translational slippage, but has the sawtooth hazard.

The posture graph for all possible step-to-step transitions is shown in Fig. 4.4. Notice that even with only three kinds of straight-line walking there are many possible transitions, and hence numerous motion segments to be recorded. However, allowing for variable step length is very important. For instance, an attract control node can be set to drive the agent to move within a certain distance of a goal location; were there only a single step size, the agent might be unable to get sufficiently close to the goal without overshooting it each time, resulting in degenerate behavior (and possible virtual injury).

One thing worthy of mention with respect to the number of different walking steps required to reproduce arbitrary curved-path locomotion is that while there are theoretically order n^2 of them, the similarities are significant. It is thus possible that it will prove feasible to store a single full set of steps along with a little more information to represent how those steps can be modified slightly to realistically turn the agent left or right, and make it sufficiently fast for our real-time applications.

5 Conclusions and Future Work

We have designed a multiprocessing system for the real-time execution of behaviors and motions for simulated human-like agents. We have used only toy examples to date, and are eager to push the limits of the system to model more complex environments and interactions amongst the agents.

Although our agents currently have limited abilities (locomotion and simple posture changes), we will be developing the skills for interactive agents to perform maintenance tasks, handle a variety of tools, negotiate terrain, and perform tasks in cramped spaces. Our goal is a system which does not provide for all possible behaviors of a human agent, but allows for new behaviors and control techniques to be added and blended with the behaviors and skills the agent already possesses.

We have used a coarse grain parallelism to achieve interactive frame rates. The behavior net lends itself to finer grain parallelism, as one could achieve using a threaded approach. Our system now is manually tuned and balanced (between the number of agents, the number of sensors per agent, and the complexity of the visual database). A fruitful area of research is in the automatic load balancing of the MOTION and SENSE processes, spawning and killing copies of these processes, and doling out agents and sensors, as agents come and go in the virtual environment. Results in real-time system scheduling and approximation algorithms will be applicable here.

6 Acknowledgments

This research is partially supported by ARO DAAL03-89-C-0031 including U.S. Army Research Laboratory; Naval Training Systems Center N61339-93-M-0843; Sandia Labs AG-6076; ARPA AASERT DAAH04-94-G-0362; DMSO DAAH04-94-G-0402; ARPA DAMD17-94-J-4486; U.S. Air Force DEPTH through Hughes Missile Systems F33615-91-C-0001; DMSO through the University of Iowa; and NSF CISE CDA88-22719.

References

[1] Ronald C. Arkin. Integrating behavioral, perceptual, and world knowledge in reactive navigation. In Pattie Maes, editor, *Designing Autonomous Agents*, pages 105–122. MIT Press, 1990.

[2] Norman I. Badler, Rama Bindiganavale, John Granieri, Susanna Wei, and Xinmin Zhao. Posture interpolation with collision avoidance. In *Proceedings of Computer Animation '94*, Geneva, Switzerland, May 1994. IEEE Computer Society Press.

[3] Norman I. Badler, Cary B. Phillips, and Bonnie L. Webber. *Simulating Humans: Computer Graphics, Animation, and Control.* Oxford University Press, June 1993.

[4] Welton Becket. *Simulating Humans: Computer Graphics, Animation, and Control*, chapter Controlling forward simulation with societies of behaviors.

[5] Welton Becket and Norman I. Badler. Integrated behavioral agent architecture. In *The Third Conference on Computer Generated Forces and Behavior Representation*, Orlando, Florida, March 1993.

[6] Welton M. Becket. *Optimization and Policy Learning for Behavioral Control of Simulated Autonomous Agents*. PhD thesis, University of Pennsylvania, 1995. In preparation.

[7] Valentino Braitenberg. *Vehicles: Experiments in Synthetic Psychology*. The MIT Press, 1984.

[8] J. Cassell, C. Pelachaud, N. Badler, M. Steedman, B. Achorn, W. Becket, B. Douville, S. Prevost, and M. Stone. Animated conversation: rule-based generation of facial expression, gesture and spoken intonation for multiple conversational agents. In *Proceedings of SIGGRAPH '94. In Computer Graphics*, pages 413–420, 1994.

[9] Thomas L. Dean and Michael P. Wellman. *Planning and Control*. Morgan Kaufmann Publishers, Inc., 1991.

[10] R. James Firby. Building symbolic primitives with continuous control routines. In *Artificial Intelligence Planning Systems*, 1992.

[11] C. R. Gallistel. *The Organization of Action: A New Synthesis*. Lawrence Elerbaum Associates, Publishers, Hillsdale, New Jersey, 1980. Distributed by the Halsted Press division of John Wiley & Sons.

[12] John P. Granieri and Norman I. Badler. In Ray Earnshaw, John Vince, and Huw Jones, editors, *Applications of Virtual Reality*, chapter Simulating Humans in VR. Academic Press, 1995. To appear.

[13] John P. Granieri, Johnathan Crabtree, and Norman I. Badler. Off-line production and real-time playback of human figure motion for 3d virtual environments. In *IEEE Virtual Reality Annual International Symposium*, Research Triangle Park, NC, March 1995. To appear.

[14] David R. Haumann and Richard E. Parent. The behavioral test-bed: obtaining complex behavior from simple rules. *The Visual Computer*, 4:332–337, 1988.

[15] Hyeongseok Ko. *Kinematic and Dynamic Techniques for Analyzing, Predicting, and Animating Human Locomotion*. PhD thesis, University of Pennsylvania, 1994.

[16] Micheal B. Moore, Christopher W. Geib, and Barry D. Reich. Planning and terrain reasoning. In *Working Notes - 1995 AAAI Spring Symposium on Integrated Planning Applications.*, 1995. to appear.

[17] Barry D. Reich, Hyeongseok Ko, Welton Becket, and Norman I. Badler. Terrain reasoning for human locomotion. In *Proceedings of Computer Animation '94*, Geneva, Switzerland, May 1994. IEEE Computer Society Press.

[18] Gary Ridsdale. *The Director's Apprentice: Animating Figures in a Constrained Environment*. PhD thesis, Simon Fraser University, School of Computing Science, 1987.

[19] John Rohlf and James Helman. IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics. *Computer Graphics*, pages 381–394, 1994.

[20] Jane Wilhelms and Robert Skinner. A 'notion' for interactive behavioral animation control. *IEEE Computer Graphics and Applications*, 10(3):14–22, May 1990.

Impulse-based Simulation of Rigid Bodies

Brian Mirtich *
John Canny †
University of California at Berkeley

Abstract

We introduce a promising new approach to rigid body dynamic simulation called impulse-based *simulation. The method is well suited to modeling physical systems with large numbers of collisions, or with contact modes that change frequently. All types of contact (colliding, rolling, sliding, and resting) are modeled through a series of collision impulses between the objects in contact, hence the method is simpler and faster than constraint-based simulation. We have implemented an impulse-based simulator that can currently achieve interactive simulation times, and real time simulation seems within reach. In addition, the simulator has produced physically accurate results in several qualitative and quantitative experiments. After giving an overview of impulse-based dynamic simulation, we discuss collision detection and collision response in this context, and present results from several experiments.*

1 Introduction

The foremost requirement of a dynamic simulator is physical accuracy. The simulation is to take the place of a physical model, and hence its utility is directly related to how well it mimics this model. A second important requirement is computational efficiency. Many applications (e.g. electronic prototyping [9]) benefit most from interactive simulation; others (e.g. virtual reality) demand real time speeds.

This paper discusses a new approach to dynamic simulation called impulse-based simulation, founded on the twin goals of physical accuracy and computational efficiency. The initial results from our impulse-based simulator look very promising, both from speed and accuracy standpoints. In this paper we give an overview of the impulse-based approach, then discuss collision detection and resolution and results from several experiments.

mirtich@cs.berkeley.edu, Department of Computer Science, 387 Soda Hall, University of California, Berkeley, CA 94720. Supported in part by NSF grant #FD93-19412.

†*jfc@cs.berkeley.edu*, Department of Computer Science, 529 Soda Hall, University of California, Berkeley, CA 94720. Supported in part by NSF grant #FD93-19412.

1.1 Related work

Moore and Wilhelms give one of the earliest treatments of two fundamental problems in dynamic simulation: collision detection and collision response [14]. Hahn also pioneered dynamic simulation, modeling sliding and rolling contacts using impact equations [8]. His work is the precursor of our method, although we extend the applicability of impulse dynamics to resting contacts, and model multiple objects in contact with impulse trains as well. These early approaches all suffered from inefficient collision detection and unrealistic assumptions concerning impact dynamics (e.g. infinite friction at the contact point).

Cremer and Stewart describe *Newton* [7, 17], probably the most advanced general-purpose dynamic simulator in use today. Newton's forte is the formulation and simulation of constraint-based dynamics for linked rigid bodies, although the contact modeling is fairly simplistic. Baraff has studied multiple rigid bodies in contact [1, 2], and shown that computing contact forces in the presence of friction is NP-hard [3]. A summary of his work in this area appears in [4].

There are few full treatments of frictional collisions. Routh [16] is still considered the authority on this subject, and more recently, Keller gives an excellent treatment of frictional collisions [10]. Our analysis is extremely similar to that of Bhatt and Koechling, who independently derived the same key equation for integration of relative contact velocities during impact. They give a classification of frictional collisions, based on the flow patterns of tangential contact velocity [6].

Wang and Mason have studied two-dimensional impact dynamics for robotic applications, based on Routh's approach [18]. Finally, a number of researchers have investigated several problems and paradigms for dynamic simulation and physical-based modeling [5, 19, 20].

2 The impulse-based method

One of the most difficult aspects of dynamic simulation is dealing with the interactions between bodies in contact. Most of the work which has been done in this area falls into the category of constraint-based methods [4, 5, 7, 19]. An example will illustrate the approach. Consider a ball rolling along a table top. The normal force which the table exerts on the ball is a constraint force that does no work on the ball, but only enforces a non-penetration constraint. In the Lagrangian constraint-based approach, this force is not modeled explicitly, but is accounted for by a constraint on the configuration of the ball (here, its z-coordinate is held constant). Alternatively, one may model the forces explicitly, solving for their magnitudes using Lagrange multipliers. However this still requires complete, exact knowledge of the instantaneous state of contact between the objects, since that determines where and when such forces can exist.

A problem with this method is that as a dynamic system evolves, the constraints may change many times, e.g. the ball may roll off the table, may hit an object on the table, etc. Determining the correct equations of motion for the ball means keeping track of these changing constraints, which can become complicated. Moreover, it is not even always clear what type of constraint should be applied; there exist at least two models for rolling contact which in some cases predict different behaviors [11]. Finally, impacts are not easily incorporated into the constraint model, as they generally give rise to impulses, not constraint forces present over some interval. These collision impulses must be handled separately, as in [1].

In contrast to constraint-based methods, impulse-based dynamics involves no explicit constraints on the configurations of the moving objects; when the objects are not colliding, they are in ballistic trajectories. Furthermore, all modes of continuous contact are handled via trains of impulses applied to the objects, whether they be resting, sliding, or rolling on one another. Under impulse-based simulation, a block resting on a table is actually experiencing many rapid, tiny collisions with the table, each of which is resolved using only local information at the collision point.

Now consider the case of a ball bouncing along the terrain shown in figure 1. Under constraint-based simulation, the

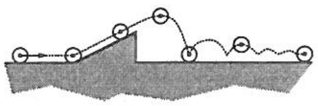


Figure 1: *A nightmare for constraint-based simulation.*

constraints change as the ball begins traveling up the ramp, leaves the ramp, and settles into a roll along the ground. All these occurrences must be detected and processed. Impulse-based simulation avoids having to worry about such transitions. In this sense, it is a more physically sound treatment since it does not establish an artificial boundary between, for example, bouncing and rolling, but instead handles the entire continuum of contact between these phases.

We do not wish to discredit constraint-based methods of dynamic simulation; indeed, there are many situations for which they are the perfect tool. We believe the impulse-based method is better suited to simulating many common physical systems, especially those which are collision intensive, or that have many changes in contact mode. We examine the possibility of using both methods of simulation together, combining the strengths of each, in section 6.

Two obvious questions concerning impulse-based simulation are: (1) Does it work, i.e. does it result in physically accurate simulations?, and (2) Is it fast enough to be practical? We defer more thorough answers to these questions to section 5, but for now state the following: impulse-based dynamic simulation *does* produce physically accurate results, and the approach is extremely fast. Simulations can certainly be run interactively with our current implementation, and we believe real time simulation is a reachable goal.

3 Collision detection

Impulse-based dynamic simulation is inherently collision intensive, since collisions are used to affect all types of interaction between objects. Hahn found collision detection to be the bottleneck in dynamic simulation [8], and efficient data structures and algorithms are needed to make impulse-based simulation feasible.

Currently in our simulator, all objects are geometrically modeled as convex polyhedra or combinations of them. The polyhedral restriction is not at all severe, because our collision detection system is very insensitive to the complexity of the geometric models, permitting fine tessellations. Indeed, some of the simulations described in section 5 use polyhedral models with over 20,000 facets, with negligible slowdown.

3.1 Prioritizing collisions

Obviously, checking for possible collisions between all pairs of objects after every integration step is too inefficient. Instead, collisions are prioritized in a heap (see figure 2). For

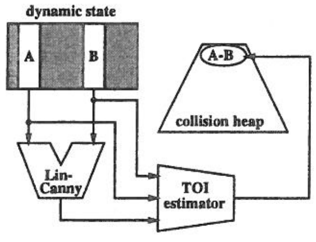


Figure 2: *Prioritizing collisions in a heap.*

each pair of objects in the simulation, there is an element in the heap, which also contains a lower bound on the time of impact (TOI) for the given pair of objects. The heap is sorted on the TOI field, thus the TOI field of the top heap element always gives a "safe" value for the next collision free integration step.

After an integration step, the distance between the objects on the top of the heap (call them A and B) must be recomputed. In our implementation, we use the Lin-Canny closest features algorithm [12]. This is an extremely efficient algorithm which maintains the closest features (vertices, edges, or faces) between a pair of convex polyhedra. It is fastest in applications like dynamic simulation, when the objects move continuously through space and geometric coherence can be exploited.

Collisions are declared when the distance between objects falls below some threshold ε_c. First suppose the distance between A and B lies above ε_c. In this case, the dynamic states of A and B along with the output of the Lin-Canny algorithm are used to compute a new conservative bound on the time of impact of A and B. The A–B heap pair is updated with this new value, possibly affecting its heap position, and the integrator is ready for another step.

If the distance between A and B is less than ε_c, a collision is declared. The collision resolution system computes and applies collision impulses to the two objects, changing their dynamic state. At this point the TOI is recomputed for these objects as before, however another step is necessary: the TOI between all object pairs of the form A–x and B–x must also be recomputed. The reason is that the TOI estimator uses a ballistic trajectory assumption to bound the time of impact for a pair of objects. Applying collision impulses to objects violates this assumption, and so every previous TOI involving such an object becomes invalid. Note that this is an $O(n)$ update step.

3.2 Further reducing collision checks and TOI updates

The strategy described above reduces collision checks significantly, especially between objects which are far apart or moving slowly. However, the number of collision checks is still $O(n^2)$ because they are performed periodically between every pair of objects. A more serious problem is the $O(n)$ TOI update step that must be performed every time a collision impulse is applied to an object. What the heap scheme misses is the fact that some objects never come near each other, and collision checks as well as TOI updates for such pairs of objects are unnecessary.

To alleviate this problem, we employ a spatial tiling technique based on Overmars' efficient point-location algorithms in fat subdivisions [15]. For each object i in the simulation, one can easily find an enclosing, axis-aligned rectangular volume B_i which is guaranteed to contain the object during the next integration step. This is possible because of the ballistic trajectory assumption.

The idea is to keep track of which objects are near each other, by keeping track of which bounding boxes overlap. To this end, the physical space is partitioned into a cubical tiling with resolution ρ. Under this tiling, Coordinates in physical space are mapped to integers under the tiling map τ:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} \xrightarrow{\tau} \begin{bmatrix} \lfloor x/\rho \rfloor \\ \lfloor y/\rho \rfloor \\ \lfloor z/\rho \rfloor \end{bmatrix}. \tag{1}$$

Let S_i be the set of tiles which B_i intersects. We store i in a hash table multiple times, hashed on the coordinates of each tile in S_i. Clearly objects i and j can only possibly collide during the next integration step if i and j are both present in some hash bucket. Only in this case do we keep object pair i-j in the collision heap. Furthermore, if object i experiences a collision impulse, TOIs need only be recomputed for object pairs i-k, where object k shares a hash bucket with object i.

This scheme tremendously reduces the number of collision checks and TOI computations that must be performed, since most objects are generally in the vicinity of only a small subset of the set of all objects. Collision detection is still $O(n^2)$ in the worst case, but almost always better. Consider for example the case of simulating a vibratory bowl feeder sorting hundreds of small parts. Since the number of parts near another part can be bounded by a constant, the number of collision checks are $O(n)$.

One added wrinkle is that one must actually employ a hierarchy of spatial tilings and hash tables of varying resolutions, in order to prevent having to hash a sofa according to tiles the size of ice cubes. The hierarchy is needed to keep the rate of bucket updates small. See Overmars for more information on this multiple resolution hashing scheme [15].

3.3 Time of impact estimator

The time of impact (TOI) estimator takes the current dynamic state (pose and velocity) of two objects as well as the closest points between them, and returns a lower bound on the time of impact for those two objects. We assume the objects are convex; concavities are handled by convex decomposition.

Let $\mathbf{c}_i$ and $\mathbf{c}_j$ be the current closest points between two objects i and j on a collision course. Let $\hat{\mathbf{d}}$ be a unit vector in the direction of $\mathbf{c}_i - \mathbf{c}_j$, and d be the distance between $\mathbf{c}_i$ and $\mathbf{c}_j$. A convexity argument shows that no matter where the ultimate contact points are located, these contact points must cover the distance d in the direction of $\hat{\mathbf{d}}$ before collision can occur. From this one obtains a conservative bound on the time of collision:

$$t_c \geq \frac{d}{(\mathbf{v}_j - \mathbf{v}_i) \cdot \hat{\mathbf{d}} + r_i\omega_i + r_j\omega_j}, \tag{2}$$

where $\mathbf{v}$ denotes center of mass velocity, r denotes maximum "radius," ω denotes maximum angular velocity magnitude, and the subscripts refer to the body. This bound assumes both objects are ballistic, so that gravitational effects cancel out. If, for instance, object i is a fixed table top, then the gravitational acceleration of j must be accounted for.

The conservation of momentum can be used to bound the angular velocity magnitude of a body in a ballistic trajectory:

$$\omega_{max} \leq \frac{\|(J_x\omega_x, J_y\omega_y, J_z\omega_z)^T\|}{\min(J_x, J_y, J_z)}, \tag{3}$$

where $\mathbf{J}$ is the vector of diagonal elements of the diagonalized mass matrix, and ω is the current angular velocity.

4 Computing collision impulses

When two bodies collide, an impulse $\mathbf{p}$ must be applied to one of the bodies to prevent interpenetration; an equal but opposite impulse $-\mathbf{p}$ is applied to the other. Once $\mathbf{p}$ and its point of application are known, it is a simple matter to compute the new center of mass and angular velocities for each body. After updating these velocities, dynamic state evolution can continue, assuming ballistic trajectories for all moving objects. The point of application is computed by the collision detection system, and hence the central problem in collision resolution is to determine the collision impulse $\mathbf{p}$. Accurate computation of this impulse is critical to the physical accuracy of the simulator. We now discuss how $\mathbf{p}$ may be computed; a more detailed discussion can be found in [13].

4.1 Assumptions for collisions

For impulse-based simulation, it is not feasible to make gross simplifying assumptions such as frictionless contacts or perfectly elastic collisions. Our approach for analyzing general frictional impacts is similar to that of Routh [16], although we derive equations which are more amenable to numerical integration. Keller also gives an excellent treatment [10], and Bhatt and Koechling's analysis is quite similar to ours [6]. There are three assumptions central to our analysis:

1. Infinitesimal collision time
2. Poisson's hypothesis
3. Coulomb friction model

The infinitesimal collision time assumption is commonly made in dynamic simulation [10]. It implies that the positions of the objects can be treated as constant over the course of a collision. Furthermore, the effect of one object on the other can be described by an impulse, which unlike a normal force can instantaneously change velocities. This assumption does *not* imply that the collision can be treated as a discrete event. The velocities of the bodies are not constant during the collision, and since collision (frictional) forces depend on these velocities, it is necessary to examine the dynamics during the collision. In short, a collision is a single point on the time line of the simulation, but to determine the collision impulses which are generated, one must use a magnifying glass to "blow up" this point, examining what happens inside the collision.

Poisson's hypothesis is an approximation to the complex deformations and energy losses which occur when two real bodies collide. Trying to explicitly model these stresses and deformations is too slow for interactive simulation; Poisson's hypothesis is a simple empirical rule that captures the basic behavior during a collision. A collision is divided into a compression and a restitution phase, based on the direction of the relative contact velocity along the surface normal. The boundary between these phases is the point of maximum compression, at which point the relative normal contact velocity vanishes. Let p_{total} be the magnitude of the normal component of the impulse imparted by one object onto the other over the entire collision, and p_{mc} be the magnitude of the normal component of the impulse just over the compression phase, i.e. up to the point of maximum compression. Poisson's hypothesis states

$$p_{total} = (1+e)p_{mc} \tag{4}$$

where e is a constant between zero and one, called the coefficient of restitution, that is dependent on the objects' materials.

Our final assumption is the Coloumb friction law. At a particular point during a collision between bodies A and B, let $\mathbf{u}$ be the contact velocity of A relative to B, let $\mathbf{u}_t$ be the tangential component of $\mathbf{u}$, and let $\hat{\mathbf{u}}_t$ be a unit vector in the direction of $\mathbf{u}_t$. Let $\mathbf{f}_n$ and $\mathbf{f}_t$ be the normal and tangential (frictional) components of force exerted by B on A, respectively. Then

$$\mathbf{u}_t \neq \mathbf{0} \quad \Rightarrow \quad \mathbf{f}_t = -\mu\|\mathbf{f}_n\|\hat{\mathbf{u}}_t \tag{5}$$

$$\mathbf{u}_t = \mathbf{0} \quad \Rightarrow \quad \|\mathbf{f}_t\| \leq \mu\|\mathbf{f}_n\| \tag{6}$$

where μ is the coefficient of friction. While the bodies are sliding relative to one another, the frictional force is exactly opposed to the direction of sliding. If the objects are sticking (i.e. $\mathbf{u}_t$ vanishes), all that is known is that the total force lies in the friction cone.

4.2 Initial collision analysis

A possible collision is reported whenever the distance between two bodies falls below the collision epsilon, ε_c. This is only a *possible* collision, because the objects may be receding. If the normal component of the relative velocity of the closest points has appropriate sign, no collision impulse should be applied. Note we are assuming the existence a normal direction; polyhedral objects have discontinuous surface normals, however reasonable surface normals can always be found.

Establish a collision frame with the z-axis aligned with the collision normal, directed towards body 1. Let $\mathbf{u} = \mathbf{u}_1 - \mathbf{u}_2$ be the relative contact velocity between bodies 1 and 2. When $u_z < 0$, a collision impulse must be applied to prevent interpenetration; it is necessary to analyze the dynamics of the bodies during the collision to determine this impulse. We use γ to denote the collision parameter; that is, γ is a variable which starts at zero, and continuously increases through the course of the collision until it reaches some final value, γ_f. All velocities are functions of γ, and $\mathbf{p}(\gamma)$ denotes the impulse delivered to body 1 up to point γ in the collision. The goal is to determine $\mathbf{p}(\gamma_f)$, the final total impulse delivered.

Initially, one might choose γ to be time since start of impact, but in fact this is not a very good choice. If the dynamics are studied with respect to time, the collision impulses are computed by integrating force. Unfortunately, the forces generated during a collision are not easily known; one can assume a Hooke's law behavior at the contact point, begging the question of how to choose the spring constants. Nonetheless, a variety of "penalty methods" do attempt to choose such spring constants.

A way of avoiding this problem is to choose a different parameter for the collision, namely $\gamma = p_z$, the normal component of the impulse delivered to body 1. The scalar p_z is zero at the moment the collision begins, and increases during the entire course of the collision, so it is a valid parameter. Let $\Delta\mathbf{u}(\gamma)$ denote the total change in relative contact velocity at point γ in the collision, and $\mathbf{p}(\gamma)$ be the impulse delivered to body 1 up to this point. Straightforward physics leads to the equation

$$\Delta\mathbf{u}(\gamma) = M\mathbf{p}(\gamma) \tag{7}$$

(see [13] for a detailed analysis). Here, M is a 3×3 matrix dependent only upon the masses and mass matrices of the colliding bodies, and the locations of the contact points relative to their centers of mass. By our infinitesimal collision time assumption, M is constant over the entire collision. It is useful to differentiate equation 7 with respect to the collision parameter γ, obtaining

$$\mathbf{u}'(\gamma) = M\mathbf{p}'(\gamma). \tag{8}$$

4.3 Sliding mode

While the tangential component of $\mathbf{u}$ is non-zero, the bodies are sliding relative to each other, and $\mathbf{p}'$ is completely constrained. Let $\theta(\gamma)$ be the relative direction of sliding during the collision, that is $\theta = \arg(u_x + iu_y)$.

Lemma 1 *If the collision parameter γ is chosen to be p_z, then while the bodies are sliding relative to one another,*

$$\mathbf{p}' = \begin{bmatrix} -\mu\cos\theta \\ -\mu\sin\theta \\ 1 \end{bmatrix}. \tag{9}$$

Proof: $p'_x = \frac{dp_x}{dp_z} = \frac{dp_x}{dt}\frac{dt}{dp_z} = f_x\frac{dt}{dp_z}$, where $\mathbf{f}$ is the instantaneous force exerted by body 2 on body 1. Under sliding conditions, $f_x = -(\mu\cos\theta)f_z = -(\mu\cos\theta)\frac{dp_z}{dt}$. Combining results gives $p'_x = -\mu\cos\theta$. The derivation for p'_y is similar. Finally, $p'_z = \frac{dp_z}{dp_z} \equiv 1$. □

It is now clear why p_z is a good choice for the collision parameter. By applying the results of lemma 1 to equation 8, with θ expressed in terms of u_x and u_y, we obtain:

$$\begin{bmatrix} u'_x \\ u'_y \\ u'_z \end{bmatrix} = M \begin{bmatrix} -\mu\frac{u_x}{\sqrt{u_x^2+u_y^2}} \\ -\mu\frac{u_y}{\sqrt{u_x^2+u_y^2}} \\ 1 \end{bmatrix}. \tag{10}$$

This nonlinear differential equation for $\mathbf{u}$ is valid as long as the bodies are sliding relative to each other. By integrating the equation with respect to the collision parameter γ (i.e. p_z), we can track $\mathbf{u}$ during the course of the collision. Projections of the trajectories into the u_x-u_y plane are shown in figure 3 for a particular matrix M; the crosses mark the initial sliding velocities.

The basic impulse calculation algorithm proceeds as follows. After computing the initial $\mathbf{u}$ and verifying that u_z is negative, we numerically integrate $\mathbf{u}$ using equation 10. During this integration, u_z will increase[1]. When it reaches zero, the point of maximum compression has been attained.

[1]Baraff and others have noted that it is possible to construct cases for which u_z decreases as p_z increases [3]. However, this situation seems to be extremely rare; it has not occurred in any of our simulations.