Games and Physics: Design Issues

Colin Low
Internet and Mobile Systems Laboratories
HP Laboratory Bristol
HPL-2000-124
September 28th, 2000*

E-mail colin_bw@hp.com

games, physics, simulation, rigid-body, modelling

The use of physics simulation in games is an important step that increases the overall level of realism, but it is a significant increase in complexity. This paper provides an overview of simulation issues that a games designer should be aware of before planning an ambitious use of physics.

* Internal Accession Date Only
Approved for External Publication

Games and Physics: Design Issues

Colin Low

Hewlett Packard Laboratories,
Stoke Gifford,
Bristol BS34 8QZ
colin_low@hp.com

25th. July 2000

Abstract

The use of physics simulation in games is an important step that increases the overall level of realism, but it is a significant increase in complexity. This paper provides an overview of simulation issues that a games designer should be aware of before planning an ambitious use of physics.

1. Introduction

A computer game is a form of simulation, but in many games physical accuracy is not a primary requirement. The introduction of physics simulation into games means that physics has to compete with graphics, audio, multiplayer group communication, and non-player AI for a heavily constrained time budget. The choice of how much or how little physics to use is an issue for the game designer, who is likely to rank entertainment above realism in his or her list of design goals.

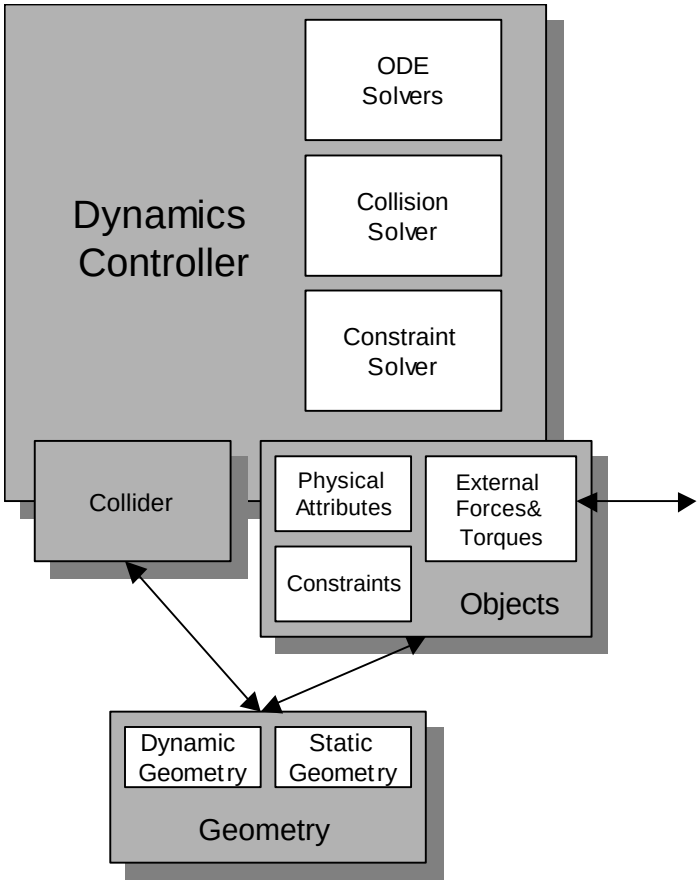
Part of the process for creating a good design is a broad grasp of what is possible in several different areas, deciding for example, whether better physical simulation would lead to unacceptable non-player AI, whether stacking boxes provides more entertaining game play than an atmospheric 3D audio environment. To place this in context, a cautionary postmortem on an ambitious use of physics simulation in the game *Trespasser* can be found at [16].

This paper reviews design and algorithm issues in simulating rigid body physics. Rigid body physics is the base level of what a physics simulation is likely to provide. It is appropriate for jointed polygonal models of people and animals, vehicles, furniture, etc. Flexible materials, deformable materials, and fluids are not discussed. The goal is to equip the game designer, the architect, and the technical lead with the background to ask critical questions about physics simulation.

2. Control Model

A dynamics controller uses the physical attributes of objects in a scene to compute new positions and orientations for objects at each simulation time step. The basic input to the controller is the set of objects subject to physical modelling, and the attributes defined for each object. These attributes will include geometry, density, elasticity, friction coefficients, constraints, and external forces and torques.

A constraint restricts the motion of an object. A typical constraint would be the joint connecting the upper arm to the lower arm on a humanoid figure. The motion of an object is defined by external forces and torques, and by internal forces and torques. A good way to imagine internal forces is to imagine two skaters on ice holding hands. External forces are generated using their ices skates. Internal forces can be felt directly as the force they have to exert through their hands in order to remain together.



We cannot model the behaviour of each skater by knowing only the external forces; it is necessary to solve for the internal forces. The dynamics controller uses a module, the constraint solver, to discover these internal forces so that dynamics of an object can be calculated as if it was moving in isolation without a constraint.

The equations of motion for rigid bodies can be derived from Newton's laws, and are second order differential equations. These are solved using a general Ordinary Differential Equation (ODE) solver, which integrates the equations of motion using a succession of small timesteps.

When a new position for an object is computed, it may turn out that it has collided with another object. Because the Dynamics Controller uses a discrete time step, the object may have penetrated the second object, and so the Dynamics Controller has to back up, and use a series of smaller time steps to find the instant of the collision. It then uses a Collision Solver to work out the impulsive forces due to the collision, and

adds these forces to any external and constraint forces. It then continues the simulation, and after a certain number of simulation time steps it updates the attributes of each modelled object. In particular, it will update the position and orientation of each object. An excellent overview of each of these steps can be found in [1].

The Dynamics Controller is unlikely to be the only controller updating the geometry of a scene. Games are pragmatic, and it is likely that there will be an Animation Controller for keyframe and motion capture animation, and perhaps an Inverse Kinematics (IK) Controller for directed motions. There will be an Input Controller for mouse, keypad, joystick or keyboard input. There may be a Skin&Bones Controller with direct control over vertex geometry. The game programmer will want direct control at certain points in the game play. This has the potential to greatly complicate matters, particularly where Physics, Animation, IK, and Skin&Bones are purchased as third-party subsystems.

The difficulty is in establishing a common representation for scene geometry that is optimal for the rendering pipeline and is consistent with the different controller subsystems. One solution is to convert geometry into the private internal representations of each subsystem, and use callbacks to update the primary scene information. This is not efficient, becomes progressively less efficient as more subsystems become involved, and is particularly acute in the case of collision detection.

3. Collision Detection

Games can often use simple heuristics to avoid the complexity of full collision detection. A simple method is to divide scene geometry into two parts: a relatively small number of dynamic objects, and a static and potentially massive “world” which is normally zoned or spatially filtered to limit the number of polygons in the view frustrum. Collision between objects can be detected using bounding spheres or boxes, and with the world by using ray-casting.

Certain types of game, such as space and vehicle games, may be able to continue with simplified collision models, but there are conceptually simple physics problems, such as stacks of boxes, where the accuracy and efficiency of the collision detector is an important factor in determining the realism and scalability of physics simulation.

Lin and Gottschalk provide a comprehensive review of collision detection approaches in [10]. Algorithms appropriate to games assume polygonal objects, or polygon soups, and to achieve scalability they are structured into a broad phase and a narrow phase. The broad phase uses a computationally cheap method such as dimension reduction [8] to decide whether objects might collide. The narrow phase uses a computationally more expensive but exact method, such as the Lin-Canny [11] or V-Clip [12] algorithms, to discover whether objects do in fact intersect. By exploiting temporal and geometric coherence (objects do not move much in a simulation time step) it is possible to reduce a problem of N pairwise comparisons to a problem linear in N.

A third-party rigid body physics SDK will come with an internal collision detector. This collision detector will need to have access to a set of polyhedral objects, or to a polygon soup, and given the trend towards graphical photo-realism in games, it will probably be dealing with reduced-polygon models (see section 8 on Physics LOD below). This has a strong impact on object modelling and run-time representation, as the physics engine and the rendering pipeline are handling *different models*.

If there is a set of objects in the scene which are not being physically modelled, but for which collision detection is important, then one can imagine a situation where *two* collision detectors are running in the same application – the physics detector running on its own private set of models, and an open detector used for all other purposes.

An answer would be to have a single collision detector as a general resource used by the physics engine and other game sub-systems. The question then is what models it uses – high-resolution models used for graphical rendering, or low-resolution models used in physics simulation.

Another issue concerns scalability for large-scale environments. A number of techniques, such as zoning or BSP trees, are used in games to make large level or terrain environments tractable. The level designer uses an intuition about *visual* complexity as a guide in deciding what to add to the level. The addition of physics, and the need to carry out collision detection among objects and terrain, adds a new level of complexity that may or may not be correlated with visual complexity. Whatever method is used for scaling the visual complexity has to be applicable to the collision complexity, or some additional zoning or spatial filtering specifically for collision detection will need to be added.

4. Collision Response

When two objects collide the result is determined by the physical properties of the objects as embodied in a collision/contact response model. There are two regimes for solving the collision problem: with and without friction. There are three common methods: impulse-based [13], analytic or constraint-based [2], and penalty force [5]. There are also multiple friction models which have been implemented.

Analytic algorithms for multiple objects with multiple static or sliding contacts in the presence of friction can be very complex, and require advanced numerical methods skills [3][4], although Baraff presents an algorithm [2]which circumvents some of these difficulties.

It is tempting to use penalty force methods, and many simulations have done, but Baraff provides warnings in [5] which are fully confirmed by the experiences of the implementers or Trespasser [16]. Collision response in games does not need to be totally realistic, but it does need to be stable and efficient. Despite these caveats, penalty force methods continue to be used because they are conceptually simple and potentially robust – see [14] for a recent example.

5. Stability

The equations of motion for rigid body dynamics are second-order differential equations that are solvable using a number of generic ODE solvers. The most common methods are the Euler method, Midpoint method, a variety of Runge-Kutta methods, and implicit methods, with or without adaptive stepsize control [15]. The choice of method is often a matter for experiment, as the simpler methods can become unstable due to modelling inaccuracy, and the more accurate methods can require much more computation.

Instability in a simulation is easy to spot, because energy is not conserved. Objects coupled by springs vibrate with larger and larger amplitudes. Colliding objects gain energy. A ball hits a wall, whizzes into a chair, the chair flies into the pile of boxes, and within seconds everything in the scene is flying about as if demonically possesed.

A key consideration is *stiffness*. This can be visualised simply. Any spring has a characteristic frequency. Strong springs vibrate at higher frequencies than weak springs. To simulate a strong spring with a higher frequency, we must use a simulation time step smaller than its characteristic period. The time step of a simulation is governed by the characteristic frequency of the strongest spring. Another way to think about stiffness is that the topography of the multi-dimensional energy landscape is dominated by the deep gullies of stiff equations. Technically, stiff equations are characterised by very different scales of independent variable, for example, strong springs and weak springs.

Simulated springs are common in rigid body dynamics. One application is to prevent colliding bodies from penetrating each other by using a spring restoring force. This is the *penalty force* method for enforcing the non-penetration constraint. Another application is the use of springs as constraints to join objects together. In both cases the constraint is met most precisely when very strong (i.e. stiff) springs are used.

The worst effects of strong springs can be countered by using implicit methods [15].

The simplest way to implement a rigid body simulator is to use penalty force methods and an Euler or Runge-Kutte integrator. This is not a good combination and is likely to exhibit instabilities.

6. Constraints

A rigid body in unconstrained motion has 6 independent degrees of freedom (DOF): three translational DOFs and three rotational DOFs. Any collection of N bodies with less than 6*N DOFs has dependent DOFs and is constrained.

Two objects connected at a point with a full rotational joint should have 12 DOF, but 3 translational DOF are lost because of the joint, leaving 9 DOFs: 3 translational (the two bodies must move together) and 6 rotational (there are no rotational constraints). The loss of degrees of freedom results in internal forces (and torques) at the joint. Common constraint types are point-to-point, point-to-nail, and point-to-path [7]. It is possible to treat objects as unconstrained if the internal forces at joints can be computed; each object trajectory is then the trajectory of a ballistic object with known external forces and torques.

Two methods commonly used to compute internal forces are the Penalty Force method [5] and the Lagrange Multiplier method [6]. The Penalty Force method introduces a deviation metric that is differentiable and non-zero when a constraint is not met. A restoring force (or torque) is computed from this metric. This method is essentially similar to adding internal springs to the system that exert restoring forces in such a way that deviations from constraints are minimized. The Lagrange Multiplier method solves for the unknown forces directly. In both cases a set of linear equations have to be solved for the unknown forces.

A major difference between the two methods is that the internal forces in the Lagrange Multiplier method are chosen in such a way that they do no work. That is, no energy is going into or out of constrained DOFs. For example, if a bead is sliding on a wire, the constraint force is always normal to the wire, and so the constraint force can never do work when the bead slides along the wire. This is not true of penalty forces.

In a penalty force system, a constraint force is zero when the constraint deviation is zero, and so constraint forces imply non-zero deviations. These non-zero deviations in

the opposite direction to restoring forces store energy, so they are not neutral in their impact on the dynamics of the system.

In order to enforce constraints accurately, penalty forces need to be large relative to other forces in the system. This is exactly the situation that produces a stiff set of ODEs as described in the Section 5.

Another issue is that the relationship between deviation and penalty force is determined by a proportionality constant. If this value is too small, constraints will not be met accurately, and if too large, may cause instability in the ODE Solver. The size of this constant has to be determined by experiment, and will usually vary from constraint to constraint. This dependency on experimentally determined "fudge factors" is a significant drawback.

7. Inverse Kinematics

One desirable (stretch) goal in games is to make Non Player Characters (NPCs) autonomous, intelligent and interesting. Adding physics simulation to an environment makes it possible to think of NPCs more as software robots than scripted automata: that is they sense and interact with their environment in a physically plausible way. A large body of inverse dynamic and kinematic algorithms [9] become available from the robot control literature.

Inverse kinematics is particularly interesting for controlled and directed motions, such as taking a step, reaching for an object, pressing a button and so on. That is, a goal is specified, and the controlled object moves to satisfy the goal. This can provide an excellent simulation of the way a human being reaches out to pick up an object.

A simple but useful way to provide this is by using pseudo-forces. The controlled object (end effecter) is attracted towards the goal by using a pseudo force that vanishes as the goal is satisfied. Instead of making acceleration proportional to force, velocity is proportional to force, to produce a smooth motion that terminates at the goal. This algorithm shares everything with a Newton-Euler rigid body simulator except the equation of motion, and there is value in combining inverse kinematics with dynamics in the physics engine. For example, an arm can be used to reach out for something, or to punch. The reaching motion is not designed to transfer energy and momentum, whereas the punching motion is designed to collide with significant energy and momentum. The punch simulation could use a goal *behind* the object to be hit, and switch from kinematics to full dynamics collision response at the moment of contact.

This raises the larger question of how animation controllers in general are integrated with physical simulation. There are many ways to specify a motion, and it is desirable that all controllers maintain a consistent, shared view of position, velocity and acceleration so that valid physical properties are maintained at all times regardless of how the motion is incurred.

8. Visual Rendering and Physics Models

A polygonal model suitable for visual rendering will in most cases be much more detailed than is required for physical simulation. It is possible to use a reduced polygon model purely for physics modelling to minimise the considerable cost of collision detection – that is, there would be a detailed model for graphical rendering, and a reduced polygon model used by the physics engine which would be

approximately coincident with the visual rendering model. The discrepancy between the two models may become apparent during collisions, when the visual surface may not coincide with the collision surface. One would like to be able to use visual models to generate reduced-polygon physical models automatically during off-line modelling.

If this approach is used, there is the increased memory cost of maintaining two polygonal models, one for the visual rendering pipeline, and the other for the physics engine.

9. Modelling

The physical properties of a set of objects are sufficiently complex that it is useful to be able to specify properties such as shape, density, joints and joint constraints, springs and dampers, collision properties etc. using a visual modelling tool.

The complexity arises when using multiple modelling plugins are combined in a tool such as 3D Studio Max. In an ideal world there would be a single export format which combined every object property – visual appearance, transformation hierarchy, object and world coordinate systems, vertex control for skinning, physics level of detail, physical properties, constraints and so on. This is so far from being the case the situation often seems somewhere between comic and tragic. Even where the quality of individual plugins is irreproachable, this does not mean that merging the exports into a unified run-time model is easy, or that the result will be compact and efficient.

10. Conclusions

Games are mass market products. They need to be robust, compact, efficient and predictably responsive even with large game environments. Physics simulation is a natural extension to the increasing realism of games, but it needs to be approached with a degree of caution. Algorithmic complexity increases very rapidly for even simple problems such as stacks of objects, and short-cutting the complexity by using simple algorithms can lead to unpredictable instabilities. That this can happen in practice as well as theory is demonstrated by the published post-mortem on the game Trespasser [16].

Collision detection is a computationally demanding requirement that needs to be thought through in the same way as visual scaling. Games designers may have to rethink scaling approaches for large environments. Although collision complexity and visual complexity are correlated, the used of reduced polygon physics models means that there is considerable scope for achieving high visual complexity *and* good physics modelling. Lastly, the addition of physics to the modelling process is a significant new step, and unifying physics models with traditional modelling (which can include new capabilities such as skin and bones, IK etc) can be non trivial.

11. References

[1] Baraff, David, Witkin, Andrew and Kass, Michael, *Physically Based Modeling Course Notes, Course 36 SIGGRAPH 99*, http://www-viz.tamu.edu/courses/viza659/00spring/s99-course36.pdf

[2] Baraff D., Fast contact force computation for nonpenetrating rigid bodies. *Computer Graphics Proceedings*, Annual Conference Series: 23-34 at http://www.cs.cmu.edu/~baraff/papers/

[3] Baraff D., Coping with friction for non-penetrating rigid body simulation. *Computer Graphics* 25(4): 31-40, 1991 at http://www.cs.cmu.edu/~baraff/papers/

[4] Baraff D., Analytical methods for dynamic simulation of non-penetrating rigid bodies. *Computer Graphics* 23(3): 223-232, 1989 at http://www.cs.cmu.edu/~baraff/papers/

[5] Baraff, D., Non-penetrating rigid body simulation, in *State of the Art Reports*, Eurographics '93, Barcelona, Spain, September 1993 at http://www.cs.cmu.edu/~baraff/papers/

[6] Baraff, David, *Linear Time Dynamics using Lagrange Multipliers*, Computer Graphics Proceedings, SIGGRAPH 96 1993 at http://www.cs.cmu.edu/~baraff/papers/

[7] Barzel, Ronen, Barr, Alan H., *A Modelling System Based on Dynamic Constraints*, Computer Graphics, Vol. 22 No.4 August 1988

[8] J. Cohen, M. Lin, D. Manocha and K. Ponamgi, *I-COLLIDE: An Interactive and Exact Collision Detection System for Large-Scaled Environments***,** Proceedings of ACM Int. 3D Graphics Conference , pp. 189-196, 1995 at http://www.cs.unc.edu/~lin/papers.html

[9] Featherstone, R., Robot Dynamics Algorithms, Kluwer Academic Publishers, 1987, 2nd. Printing 1998

[10] M. Lin and S. Gottschalk, *Collision Detection between Geometric Models: A Survey* In the Proceedings of IMA Conference on Mathematics of Surfaces 1998 at http://www.cs.unc.edu/~lin/papers.html

[11] Lin, M. and Canny, J., *A fast algorithm for incremental distance calculation*, IEEE Conference on Robotics and Automation, pages 1008-1014, 1991

[12] Mirtich, Brian, V-Clip: fast and robust polyhedral collision detection;; ACM Trans. Graph. 17, 3 (Jul. 1998), Pages 177 – 208 at http://www.merl.com/people/mirtich/pubs.html

[13] Mirtich, Brian, & Canny, John, Impulse-Based Simulation of Rigid Bodies, *Proc. of 1995 Symposium on Interactive 3D Graphics*, pp. 181-188, April 1995 at http://www.merl.com/people/mirtich/pubs.html

[14] Mirtich, Brian, "Timewarp Rigid Body Simulation", to appear in SIGGRAPH 00, July 2000, at http://www.merl.com/people/mirtich/pubs.html

[15] Press, William H., Teukolsky, Saul A., Vetterling, William T., Flannery, Brian P., *Numerical Recipes in C*, Cambridge University Press, Second Edition 1992

[16] Wyckoff, Richard, *Postmortem, Dreamworks Interactive's Trespasser*, Game Developer's Magazine, June 1999, now at http://www.gamasutra.com/features/19990514/trespasser_03.htm