Evaluation of real-time physics simulation systems

Conference Paper · January 2007

DOI: 10.1145/1321261.1321312 · Source: DBLP

2 authors:

Adrian Boeing
University of Western Australia
26 PUBLICATIONS 341 CITATIONS
SEE PROFILE

Thomas Braunl
University of Western Australia
195 PUBLICATIONS 1,903 CITATIONS
SEE PROFILE

Evaluation of real-time physics simulation systems

Adrian Boeing
School of Electrical, Electronic and Computer Engineering
University of Western Australia

Thomas Bräunl
School of Electrical, Electronic and Computer Engineering
University of Western Australia

Abstract

We present a qualitative evaluation of a number of free publicly available physics engines for simulation systems and game development. A brief overview of the aspects of a physics engine is presented accompanied by a comparison of the capabilities of each physics engine. Aspects that are investigated the accuracy and computational efficiency of the integrator properties, material properties, stacks, links, and collision detection system.

Keywords: dynamic simulation, physics engine, evaluation.

CCS Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling - Physically based modeling; D.2.11 [Software Engineering]: Software Architectures - Data abstraction.

1 Introduction

Recently there has been a marked increase in the number of free, publicly available physics engines. Given the plethora of physics engines available it can be very difficult for a developer to select an appropriate physics engine for their application. For a game developer many aspects come into consideration including available features, supported platforms, ease of use, and run-time performance. Researchers and simulation engineers are typically more concerned with the accuracy of a physics system.

In the past it has been very difficult to compare physics engines, however recently a number of physics engine abstraction systems have become available such as PAL (Physics Abstraction Layer), OPAL (Open Physics Abstraction Layer), and GangstaWrapper. These abstraction layers allow developers to implement one version of their physics system through a unique interface and test their application with multiple engines. Additionally they simplify the task of comparing physics engines directly.

OPAL is the least complete, providing only an interface to one physics engine. The GangstaWrapper provides an interface to four physics engines, whereas PAL provides support for ten engines (See Section 3). GangstaWrapper is no longer maintained, however provides a solid interface for the physics engines it supports. PAL is still being maintained and expanded, however does not feature as many configurations as GangstaWrapper.

An alternative approach to achieving physics engine interoperability is the COLLADA standard. Coumans and Victor [2007] provide a brief overview article of the COLLADA physics standard and provide a short comparison of the capabilities of the Bullet, Novodex (Ageia PhysX), ODE (Open Dynamics Engine) and Havok physics engines.

Seugling and Rolin [2006] published an article comparing three different physics engines, Newton, Novodex (Ageia PhysX), and ODE (Open Dynamics Engine). Their evaluation focused primarily on the performance of the systems for simulators. In this article similar tests will be conducted and analyzed with an additional focus on gaming technology. From their test results they concluded that Novodex (Ageia PhysX) provided the best results. Although most of the tests provided a quantitative difference in performance the final evaluation was determined from a very rough grading system. As a result the final findings did not necessarily reflect significant performance differences in the individual tests between physics engines.

The main task of all physics engines is to solve the forward dynamics problem. Simply stated the forward dynamics problem is: given the forces acting on a system, what is the motion of the system?

There are a number of factors that influence the characteristics of a physics engine. These range from the simulation paradigm, collision detection and response to the type of numerical integrator, and whether air resistance is considered. As a result each physics engine will provide quite different results despite stimulating the exact same system. A good overview of common approaches to dynamic simulation is provided by Erleben [2004].

There are six essential factors that determine the overall performance of the physics engine:

- *Simulator Paradigm*, determines which aspects can be accurately simulated. This affects the accuracy in resolving constraints. An overview of simulator paradigms is presented in [Erleben, 2004]. Mirtich provides a comparison of constraint–based methods and impulse based methods in [Mirtich, 1996], and a comparison of penalty based methods with constraint-based methods is presented by Baraff [1992].
- *The integrator*, determines the numerical accuracy of the simulation. Some integration methods are discussed by Baraff [1997], and integrator stepping methods are also covered by Erleben [2004].
- *Object representation*, contributes to the efficiency and accuracy of collisions in the simulation. Various aspects of object representation choices are discussed in [Hadap et al. 2004] and [Ratcliff 2007].
- *Collision detection and contact determination*, also contribute to the efficiency and accuracy of collisions in the simulation. This is discussed in [Kavan, 2003] and [Hadap et al. 2004].

- *Material properties*, determines which physical models, if any, the simulation can approximate (eg: Coloumb friction). Friction properties are covered by Kaufman [2005].
- *Constraint implementation*, determines which constraints are supported and how accurately they can be simulated. See [Erleben, 2004].

The most straightforward numerical integrator is Euler's method [Baraff, 1997]:

$$x(t_0 + h) = x_0 + h\dot{x}(t_0) \tag{1}$$

In physics engines a Symplectic Euler integrator is often employed due to its ease of use. The Symplectic Euler integrator is similar to the Euler integrator, except that the updated velocity is used before calculating the position.

$$\dot{x}(t_0 + h) = \dot{x}_0 + h\ddot{x}(t_0) \tag{2}$$
$$x(t_0 + h) = x_0 + h\dot{x}(t_0)$$

There are a large number of constraints that can be simulated. The most useful constraints are ones that model the behavior of real life systems. The three most common constraints are prismatic, revolute, and spherical constraints. Prismatic constraints are also referred to as slider constraints. They allow translation along a specified axis, and no rotational movement. A revolute constraint allows rotation only in one plane, and hence is referred to as a hinge constraint. A spherical constraint can also be referred to as a ball and socket constraint. It allows rotation about a point.

There are also a number of less common constraints. A universal constraint consists of two revolute constraints connected at 90° relative to each other; this provides a similar range of motion to a spherical constraint, except without one axis of rotation. A fixed constraint simply attaches one body directly to another restricting all degrees of freedom. The distance constraint simply maintains a certain distance between bodies. Finally, the corkscrew constraint limits translation along one axis, and only allows rotation about that axis.

Simulated vehicles and characters (or rag dolls) are often referred to as constraints in the literature. These are actually containers, including multiple specific constraints types such as springs for simulating suspensions.

2 PAL Software Design

The physics abstraction layer (PAL) provides a set of unique interfaces to various common properties of physics engines. There are eight basic interface groups provided. These are the interfaces for the core physics engine, a body, materials, geometries, links, sensors, actuators, and terrain representations. The interfaces are designed to be able to support varied levels of simulation capabilities.

For example, the terrain and geometries are kept separate, as some engines are only capable of supporting a static plane geometry, or a static heightfield. Conversely, many physics engines are capable of using various geometries for static or dynamic bodies interchangeably. These physics engines are then able to support a unique geometry implementation applicable for both static and dynamic bodies. Taking a similar design approach to all interfaces enables a maximum level of support for engines following different designs.

For materials there are two interfaces provided: unique materials, and material interactions. This allows the support for engines that are able to specify the material properties for the interactions between different materials.
Sensors and actuators are mostly engine independent as sensors and actuators can interact with PAL directly through querying bodies and geometries. PAL supports inclinometer, gyroscope, accelerometer, position sensitive device, contact, velocimeter, compass, GPS, and transponder sensors. A number of actuators are also supported including direct force and impulse actuators, as well as DC motor, servo, propeller, hydrofoil, and spring actuators. Additionally actuators to support liquid effects such as drag and buoyancy are included.

The supported dynamic geometries are boxes, capsules (cylinders with capped ends), convex hulls and spheres. The bodies supported are a geometry independent body, as well as a compound body. These can have any geometry attached to them. A box, capsule, convex hull and sphere body are also provided. This allows engines that are incapable of supporting varied geometry to directly support a box body, whereas engines that are capable of supporting geometry independently from a body can support the full functionality.

This design approach ensures that a large number of physics engines can be supported and the PAL design does not restrict the types of engines that can be supported. It also enables incremental support of a physics engine's features as the engine is developed.

The design concept used to facilitate an abstract extensible architecture and provide a central repository is a versioned pluggable factory [Culp, 1999]. A software factory class offers a set of services for generating instances of various subclasses without explicitly requiring the name of the class we wish to construct. [Gamma, et al., 1995] A pluggable factory expands this concept by allowing plug-ins to automatically extend the applications functionality without requiring any modifications to the application code itself.

To implement a versioned pluggable factory, the factory class requires a registry that maintains a list of all available components, the version of the component, and a method for creating a component. When a component is created, the factory can search through the registry for the desired class type, construct it, and return it for use. Each class that needs to be accessible via the factory requires a method that allows a copy of itself to be created, as well as method to add its information to the factories registry. By creating a static copy of the class the information is automatically registered at the very beginning of the application, before any user code is executed. Implementation details of this approach for C++ are provided in [Culp, 1999].

3 Physics Engine Review

Most physics engines have a particular target application to which they are optimized. This results in different performance in each of the above categories, and often extra features are made available specifically included for the target application. PAL supports ten different physics engines, of which seven are tested in this comparison. The engines supported by a PAL are AGEIA PhysX (also referred to as Novodex), Bullet Physics Library, Dynamechs, JigLib, Meqon, Newton Physics SDK, Open Dynamics Engine, OpenTissue Library, Tokamak, True Axis Physics SDK.

Engine	License	Cost : Edu/Com	Platform:PC	Platform:Console
AGEIA PhysX / Novodex	EULA	Free/Free	Win32/Linux/-	Xbox360/PS3/-
Bullet	Open, Zlib	Free/Free	Win32/Linux/Mac	Xbox360/PS3/-
JigLib	Open	Free/Free	Win32/Linux/-	-/-/-
Newton	EULA	Free/Free	Win32/Linux/Mac	-/-/-
Open Dynamics Engine	Open, LGPL/BSD	Free/Free	Win32/Linux/Mac	Xbox360/PS3/PSP
Tokamak	Open, BSD	Free/Free	Win32/Linux/-	-/-/-
True Axis	EULA	Free/Hobby+Full	Win32/Linux/Mac	Xbox360/-/-

Table 1 – Comparison of engines license and supported platforms

Engine	Corkscrew	Distance	Fixed	Generic (6D)	Prismatic	Revolute	Spherical	Universal	Vehicle
AGEIA PhysX / Novodex	n	y	y	y	y	y	y	n	y
Bullet	n	n	n	y	n	y	y	n	y
JigLib	y	y	y	y	n	y	y	n	y
Newton	y	n	n	y	y	y	y	y	y
Open Dynamics Engine	n	n	y	n	y	y	y	y	n
Tokamak	n	n	n	y	n	y	y	n	n
True Axis	n	n	n	y	y	y	y	n	y

Table 2 – Comparison of engines constraints support

Engine	Box	Capsule	Cylinder	Cone	Convex Mesh (Dynamic)	Compound Object	Heightfield (Static)	Plane	Sphere	Triangle Mesh (Static)
AGEIA PhysX / Novodex	y	y	n	n	y	y	y	y	y	y
Bullet	y	y	n	y	y	y	y	y	y	y
JigLib	y	y	n	n	n	y	y	y	y	y
Newton	y	y	y*	y	y	y	n	n	y	y
Open Dynamics Engine	⍰	y	y	n	n	y	y	y	y	y
Tokamak	y	y	n	n	y	y	n	n	y	y
True Axis	y	y	y	n	y	y	n	n	y	y

* Newton supports a Chamfer cylinder

Table 3 – Comparison of engines geometry support

Engine	Static Friction	Kinetic Friction	Restitution
AGEIA PhysX / Novodex	y	y	y
Bullet	y	n	y
JigLib	y	y	n
Newton	y	y	y
Open Dynamics Engine	y	y	y
Tokamak	y	n	y
True Axis	y	n	y

Table 4 – Comparison of engines material support

There are three engines supported by PAL that are not tested. Dynamechs is not tested as it does not support collisions between two dynamic bodies. It only supports collisions between dynamic and static bodies. Meqon is not tested as it is no longer available, and the OpenTissue Library was not included since it is not a complete physics engine, rather a meta library and thus it is difficult to construct a fair and general test configuration.

There are a number of aspects that are interesting to compare. As most physics engines are used as middleware, a typical project will already have a target platform and budget based on other factors [Saral, et al., 2004]. A comparison matrix of the different engines licenses, costs, and supported platforms is provided in Table 1. The cost column indicates first the cost for noncommercial use, then the cost for commercial use.

Table 2 indicates the types of constraints supported by each engine. Provided an engine supports the generic six degrees of freedom constraint, then all other constraints can be constructed. The vehicle column in the table indicates whether vehicles are supported natively by the engine. This does not mean that the engine supports constraint-based vehicle models as opposed to ray cast vehicles.

Provided the application developer has the necessary skills all custom constraints can be constructed from the generic constraint. However, it is uncommon for a developer to implement more than one custom constraint for their application.

All engines provide an interface for a generic constraint. ODE has no explicit support for a generic constraint however as it is an open source project it can be easily modified to simulate any custom constraint. No physics engine supported all constraints.

The different geometry supported by each engine is indicated in Table 3. Provided an engine has support for a static triangle mesh, then other meshes such as height fields can be easily stimulated. However it may provide inferior performance compared to engines that natively support height fields. Since physics engines typically require only a basic geometry representation for simulated objects simple geometries are usually sufficient provided they can be combined in a compound object that estimates the simulated object.

The material properties supported by each engine are presented in Table 4. For gaming applications is usually sufficient if some form of static friction and restitution is available.

The AGEIA PhysX physics engine provides a number of additional features not indicated in the tables above. It is the most full featured engine provided in this comparison. Since it is a commercial engine the implementation details are unknown, however fixed and variable time steps are possible. It provides an additional number of joint constraints including cylindrical, point on plane, point on line, springs, and pulleys. A number of vehicle representations are provided, and a dynamic triangular mesh geometry is also provided. Anisotropic friction is supported for the materials, and the engine includes a number of advanced features including fluids, character controllers, swept geometries, soft bodies, cloth, as well as a serialization API and advanced hardware support for AGEIAs own physics processing unit. Historically the PhysX engine derives from a previous offering named Novodex, which was then updated to include support for AGEIA's custom hardware, and incorporated technology obtained from purchasing Meqon. However, the Novodex API naming convention was retained. For this reason it is typically referred to as Novodex.

A relative newcomer to the physics scene is the Bullet physics library. For this reason it does not currently provide many additional features, the only feature included not listed in the tables above is a support for swept geometries and swept collision detection. It is a hybrid impulse and constraint-based engine that supports both variable and fixed time steps. It also includes a partial graphics processing unit (GPU) physics implementation.

JigLib is a hobby physics engine developed by Danny Rowlhouse. It is an impulse based approach that uses a Euler integrator and fixed time stepping. It is representative of what is possible for an in-house physics engine to achieve. It provides an additional velocity-based constraint.

The Newton Game Dynamics physics engine is also a closed engine and hence the implementation details are unknown. It provides a few additional features such as buoyancy, an additional "up-vector" constraint, an adaptive friction model, examples of various custom constraints, and a rag doll container.

The Open Dynamics Engine is a constraint-based physics engine that uses a Euler integrator and fixed time stepping. It provides an additional 2D constraint, and has been ported to a large number of platforms.

Tokamak is an impulse-based engine that uses a Euler integrator and fixed time stepping. Additional features are a container for animated bodies, and support for breakable joints.

True Axis is another closed engine, however provides the source code in obfuscated form. This has both advantages and disadvantages since it is the simulation developers responsibility to build an optimal library. Microsoft's Visual Studio 2005 was used to build True Axis for this evaluation. True Axis provides a few additional features including a line list constraint, serialization functionality and swept collision detection.

4 Physics Engine Evaluation

Five tests were performed to assess the aspects of the physics engines. These are integrator, material, constraint, collision and stacking tests.

4.1 Integrator Performance

The integrator is responsible for calculating a body's position given the forces acting on it. The performance of the integrator effects the accuracy of the simulation. This is not of a great concern for gameplay, as game designers are unconcerned with physically accurate representations. Simulation engineers however should be concerned with the integrator performance, especially since they are likely to layer additional environmental effects such as air resistance or water resistance on top of the physics engine.

To test integrator performance a very simple test is performed. A sphere is constructed at the origin and allowed to drop from gravitational forces. Gravity is set to -9.8m/s, and the time step is set to 0.01. The positions presented by the physics engines are then recorded and compared to ideal cases for various integrators. From classical physics position of a body with no initial velocity can be calculated from:

$$r = \frac{1}{2}at^2 \tag{3}$$

Where r is the bodies displacement
a is the bodies acceleration
and t is time.

Figure 1 and Figure 2 illustrate the accumulated position errors due to the integrator relative to the ideal case presented above. The errors have been normalized with respect to the Symplectic Euler integrator. Most physics engines provide results similar to the Symplectic Euler integrator, or 2nd order Euler. Novodex (Ageia PhysX) provided the best results. The integrator for the Newton physics engine provided the worst results. The results were close to what would be expected if the physics system was simulating air drag of an extremely smooth object (eg: an aircraft wing [Aerodynamic Database Drag Coefficients]). However this effect is due to forced velocity dampening by the Newton Euler integrator.

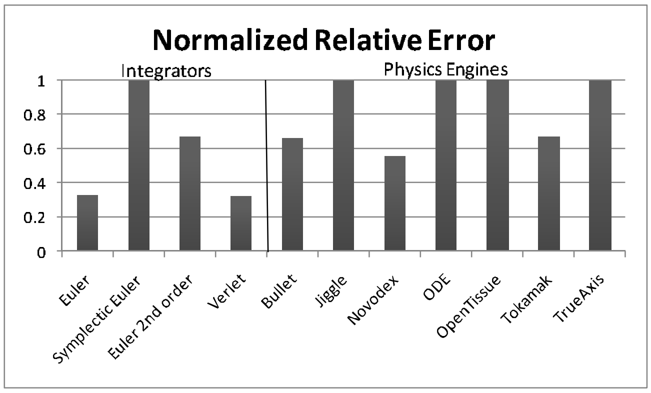

Figure 1 – Positional error from cumulative numerical integrators relative to the ideal case normalized to the Symplectic Euler integrator error

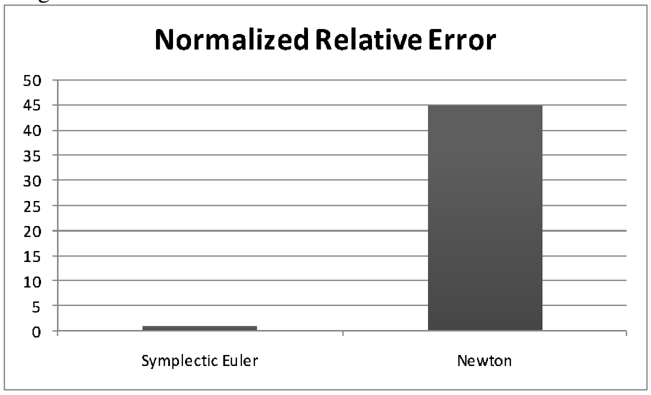

Figure 2 – Positional error comparison of Symplectic Euler integrator and Newton physics engine

4.2 Material Properties

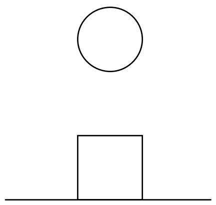

Figure 3 – Materials test configuration

Materials are responsible for stimulating friction and restitution properties during a collision. From a gaming perspective accurate simulation of physical friction models is not as important as simply being able to model different behaviors with different material properties. In contrast, accurate friction and restitution models are critical for simulation engineers.

The materials restitution properties were tested by colliding a box with a sphere. The box is placed on the ground and the sphere is placed one meter above. The box was of dimensions 1×1×1m3, and a mass of 1kg, the sphere had a radius of 0.5m, and a mass of 1kg. Three different values of restitution were tested, 0.1, 0.5 and 0.9. Since the box on the ground is stationary the relationship between the dropped height and the coefficient of restitution is given in classical physics by:

$$C_R = \sqrt{\frac{h}{H}} \tag{4}$$

Where C_R is the coefficient of restitution
h is the bounce height
and H is the drop height.

A graph of the bouncing boxes positioned over time for a restitution coefficient of 0.5 is depicted in Figure 4. The maximum heights obtained for the three different restitution values are given in Figure 5. These results indicate that only TrueAxis provides a good approximation of coefficients of restitution. None of the engines handle low values of restitution correctly.

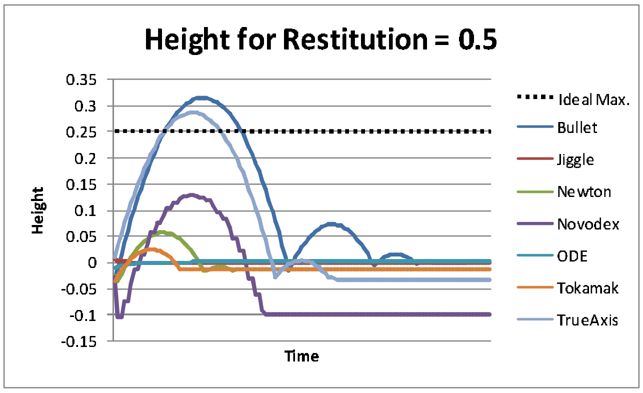

Figure 4 - Bounce height for a coefficient of restitution of 0.5

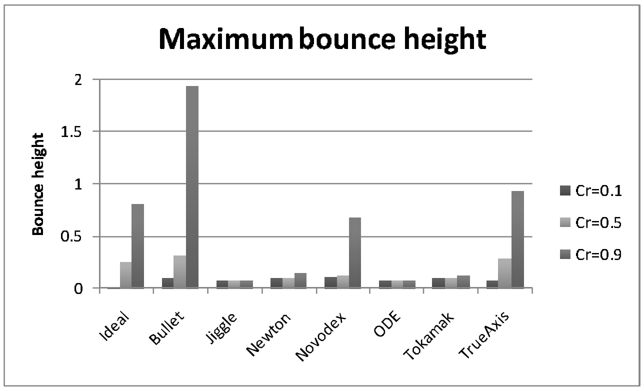

Figure 5 – Maximum bounce height for varying values of restitution.

For gaming applications an accurate restitution model is unnecessary, more important is that there is a correlation between an increase in restitution value and the bounce height. Bullet and Novodex give acceptable relative increases in the bounce height, and to a lesser extent Newton and Tokamak showed a correlation.

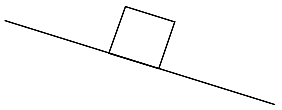

Figure 6 - Friction test configuration

To test the static friction a 5×1×5m box was placed on an inclined plane. A static friction coefficient was assigned to the materials of the box and the plane, and the angle of the plane was then incrementally increased to test the angle at which the box would first start sliding. This process was repeated for the range of static coefficients from 0.1 to 0.7, increasing by 0.1. The angle of the plane was tested in the range of 0 to 0.7 in increments of 0.05 radians.

The Jiglib and ODE physics engines were not included in this test as the PAL implementation does not support resetting a bodies orientation after construction.

The Newton physics engine provides the closest approximation of the ideal results. Novodex also provides a good approximation, however applies too much static friction effect. All engines display an increase in the angle required before motion occurs, indicating they are all suitable for game applications. For simulation systems only Newton provides an accurate model.

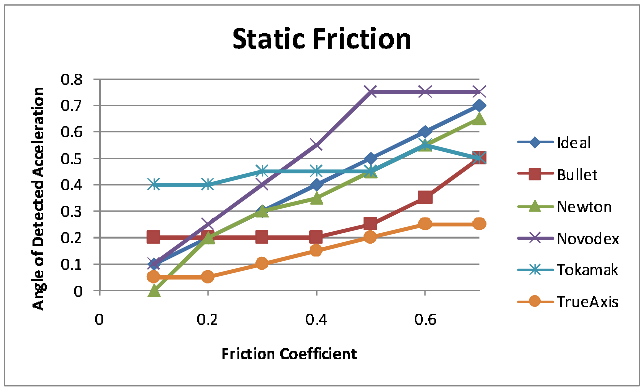

Figure 7 – Angle of the plane at which the body began movement versus the static friction coefficient

4.3 Constraint Stability

Constraint stability is one of the areas of importance for game designers. If constraints are unstable numerical errors can cause constrained bodies to slowly drift apart. This results in unrealistic looking results. This is also of critical importance for simulation engineers simulating multi-body robotic systems, the traditional application of dynamic simulation systems.

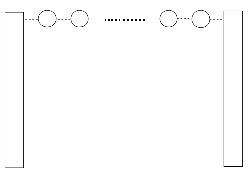

Figure 8 –Constraint test configuration

To test the constraints stability a chain of spherical links connecting a number of spheres was simulated. The chain was attached to two boxes as indicated in Figure 8. Each sphere in the chain had a radius of 0.2m, and a mass of 0.1kg. The mass of the boxes was 400 times the number of constraints.

The two side boxes were as high as the number of constraints, and the supporting base measured 1x1m². The test was run for 20 seconds. Figure 9 illustrates the constraint error measured from the accumulated difference in the distance between two links minus relative to the initial case. The Newton physics engine is not illustrated as it contains significantly greater error than other physics engines, averaging 30 times the error of other engines.

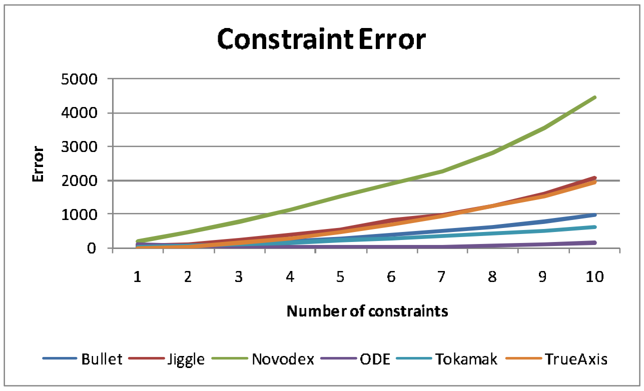

Figure 9 - Constraint error

The Tokamak engine provides the best results for the constraints solving them in the least time with the second best accuracy. ODE provides the most accurate results but requires the most time to solve the constraints. Novodex provides the second greatest constraint error. This is an interesting result as Novodex is often employed in robotic simulation systems such as the Microsoft Robotics Studio. It should also be noted that ODE's slower and more accurate WorldStep integrator was employed, which is not always used in robotic simulators.

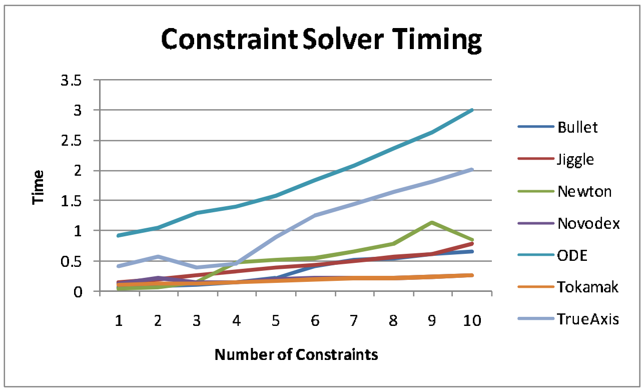

Figure 10 – Constraint timing

4.4 Collision System

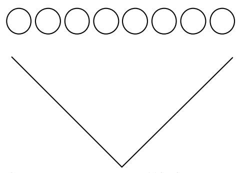

Figure 11 – Collision test configuration

The collision system is an essential part of the physics engine. Failure to detect a collision during a simulation leads to incorrect results. Similarly gameplay becomes inconsistent if game objects fall through the simulated game world. To test the collision system an inverted square pyramid mesh is constructed. The pyramid apex is 1m deep, and the opening of the pyramid measures 2×2m². A 8x8 grid of spheres with a radius of 0.04m is dropped into the open pyramid.

Penetration of the pyramid is detected by comparing all of the spheres positions to the polygons that make up the pyramid. If any sphere is less than its radius away from the pyramid's polygons, then a penetration error is accumulated. This error is depicted in Figure 12. The engines that are not included in this graph (Novodex, ODE and Tokamak) fail the collision detection test (ie: spheres fall through the pyramid).

At the time of the impact a large spike in the penetration error is experienced by all engines except Jiggle. Bullet manages to recover from the error and settles into a steady state with almost no error. Newton and TrueAxis penetration error evens out, but not at a low enough level to stop the motion of the spheres.

The Tokamak engine only barely fails this test with one sphere passing through the pyramid. Novodex and ODE fail the test completely, due to the inability of these engines to correctly reorder and optimize the mesh structure passed to them by PAL or bugs in the mesh collision detection routines. A different implementation of this test may allow Novodex and ODE to pass.

For some gaming applications and integrator step of 100Hz is unrealistic, and larger steps are common. To test this extreme, the same test was repeated at 15Hz. The Bullet engine fails this test, however TrueAxis performs very well, and is capable of passing this test at just 5Hz.

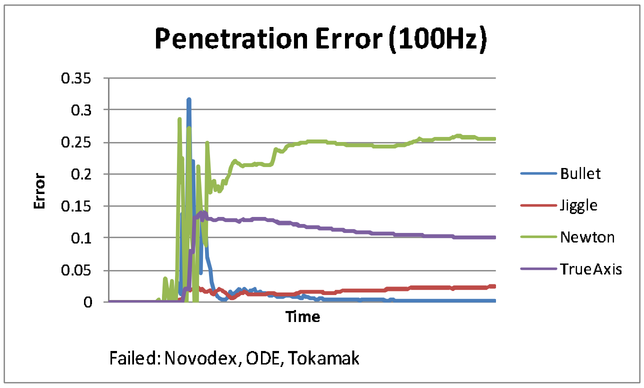

Figure 12 - Collision penetration error over time

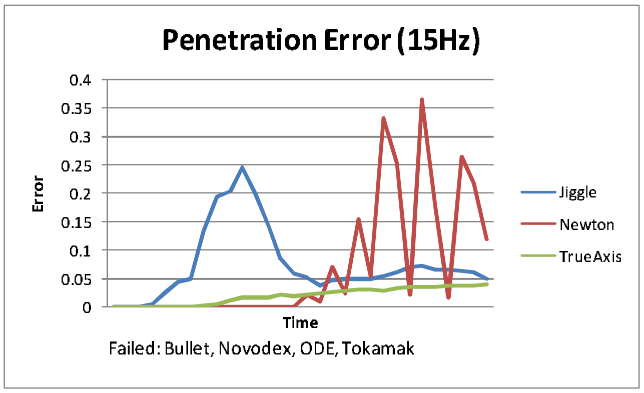

Figure 13 – Collision penetration error over time with an integrator step of 15Hz

4.5 Stacking Test

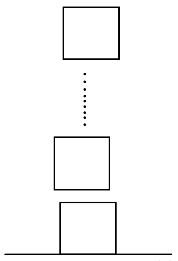

Figure 14 - Box stacking test configurations

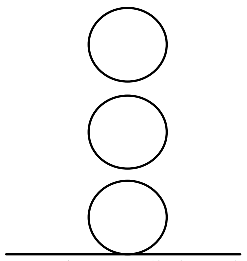

Figure 15 – Realistic sphere stacking test

A test that is important for game developers, but relatively unimportant for most simulation engineers is the efficiency of a physics engine in handling stacked objects. In this test a set of $1\times1\times1m^3$, 1kg cubes are dropped in a stack on top of one another, with a distance of 0.1m between them. Each cube is displaced by a random amount of maximal 0.1m in both directions parallel to the ground. Automatic body sleeping is disabled. It is not feasible to verify what the physically correct behavior for a stack of objects is, i.e. at which point the stack should collapse. The results can then only be examined by visual inspection, and all the physics engines pass this test.

A test for visually realistic results is to stack three spheres directly on top of each other. In the real world dropping three spheres on to one another should not result in a stack. However, every physics engine that was tested stacked the three spheres providing visually unrealistic results. Although the results produced by the engines are a mathematically correct implementation of the physics models failure to add noise to the simulation results in visually unrealistic outcomes. Since no physics engine supports any noise models every engine fails this test.

One metric that is possible to measure is the time taken to update the physics engine. The computation time required to update the physics engine for the corresponding number of stack objects is illustrated in Figure 16.

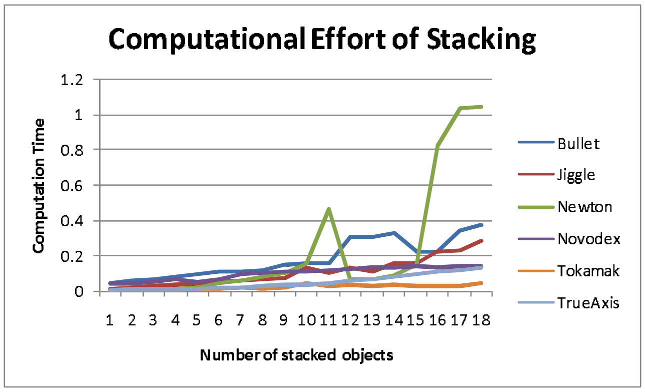

Figure 16 - Computational effort of stacked objects

5 Conclusion

The physics abstraction layer provides a uniform, extensible interface to the physics engines. Since physics engine's API's are constantly changing, whether a response to new hardware (eg: GPU, multi-core CPU), or new code or capabilities (eg: adding vehicle support or new contributions from the open source community) PAL significantly reduces the workload of an application developer by providing a static API. This enables developers to quickly switch between physics engines, take advantage of specialized hardware, or target a new platform.

All of the physics engines analysed provide properties suitable for game development. The most interesting result is that some of the engines employed in the research and simulation community are inappropriate choices for common problem tasks.

No one engine performed best at all tasks, and almost every test was performed best by a different engine. This illustrates the complexity involved in determining which physics engine a developer should select, and the difficulty in developing a general purpose physics engine. The tests performed in this evaluation should provide a guide to developers as to which engine to select.

The only test which none of the simulators passed was the realistic stacking of three spheres. None of the simulators included any noise to improve the realism of the simulation.

Novodex (Ageia PhysX) performed the best in the integrator test. True Axis delivered the best results for modeling restitution, whereas Newton provided the best estimation for static friction. Tokamak provided the excellent results for solving large chain constraints, in terms of computational efficiency and error. It also was the most efficient for computing stacked objects. ODE provided the best results for constraint accuracy when configured to use an accurate integrator. In the collision penetration test Jiggle and Bullet performed very well, and TrueAxis performed very well for large integrator step sizes.

Of the open source engines the Bullet engine provided the best results overall, outperforming even some of the commercial engines. Tokamak was the most computationally efficient, making it a good choice for game development, however TrueAxis and Newton performed well at low update rates. For simulation systems the most important property of the simulation should be determined in order to select the best engine.

The evaluation tests provided by PAL allow engine developers to directly compare their physics engines and identify any errors in their implementations, as well as highlight any cases where their engine performs well. This should assist engine developers in improving their implementations.

Acknowledgements

We would like to thank the feedback we received from all the people in the dynamic simulation community, especially the developers of the physics engines PAL supports, Axel Seugling for providing permission to re-implement and publish parts of their work, and Dirk Gregorius for his advice and feedback.

References

AERODYNAMIC DATABASE DRAG COEFFICIENTS. http://aerodyn.org/Drag/tables.html (accessed August 2, 2007).

AGEIA PHYSX. http://www.ageia.com/ (accessed August 14, 2007).

BARAFF, D. 1992. *Dynamic Simulation of Non-Penetrating Rigid Bodies*. PhD Thesis, Computer Science Department, Cornell University.

BARAFF, D. 1997 *Physically Based Modeling: Principles and Practice.* Carnegie Mellon University,

BULLET PHYSICS LIBRARY. http://bullet.sourceforge.net/ (accessed August 14, 2007).

COUMANS, E. and VICTOR, K. 2007. *COLLADA physics.* In *Proceedings of the twelfth international conference on 3D web technology*. 104-105.

CULP, T. 1999. Industrial Strength Pluggable Factories. In C++ Report, vol 10.

DYNAMECHS (DYNAMICS OF MECHANISMS): A MULTIBODY DYNAMIC SIMULATION LIBRARY. http://dynamechs.sourceforge.net/ (accessed August 14, 2007).

ERLEBEN, K. 2004 *Stable, Robust, and Versatile Multibody Dynamics Animation.* PhD Thesis., Department of Computer Science, University of Copenhagen.

GAMMA, E., HELM R.. JOHNSON, R., and VLISSIDES J.1995 *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley.

HADAP, S. EBERLE, D. VOLINO, P. LIN, M. C. REDON, S. and ERICSON, C. 2004. Collision detection and proximity queries. ACM SIGGRAPH 2004 Course #15 Notes.

JIGLIB - RIGID BODY PHYSICS ENGINE. http://www.rowlhouse.co.uk/jiglib/ (accessed August 14, 2007).

KAUFMAN, D. EDMUNDS T., and PAI D. 2005. Fast frictional dynamics for rigid bodies. In *Proceedings of ACM SIGGRAPH 2005.* Vol. 24 No. 5, 946-956

KAVAN, L. 2003. Rigid body collision response. In *Proceedings of the 7th Central European Seminar on Computer Graphics*.

Meqon. http://meqon.com/ (accessed August 14, 2007).

MIRTICH, B. 1996. *Impulse-based Dynamic Simulation of Rigid Body Systems*. PhD Thesis, University of California, Berkeley.

NEWTON GAME DYNAMICS. http://www.newtondynamics.com/ (accessed August 14, 2007).

OPEN DYNAMICS ENGINE. http://www.ode.org/ (accessed August 14, 2007).

OPEN PHYSICS ABSTRACTION LAYER. http://opal.sourceforge.net/ (accessed August 14, 2007).

OPENTISSUE LIBRARY. http://www.opentissue.org/ (accessed August 14, 2007).

PHYSICS ABSTRACTION LAYER. http://pal.sourceforge.net/ (accessed August 14, 2007).

RATCLIFF J. 2007. Automatic Generation of Dynamics Models, Game Developers Conference 2007 Course Notes. March.

SARAL, U. and SCHMIEDER C. 2004. *Pricing Strategy Process*. Linkopings University. January 21, 2004. http://www.continuousphysics.com/ftp/pub/test/index.php?dir=physics/papers/&file=meqon_pricing_exjobb.pdf (accessed August 16, 2007).

SEUGLING, A, and ROLIN M. 2006. *Evaluation of Physics Engines and Implementation of a Physics Module in a 3d-Authoring Tool.* Masters Thesis, Department of Computing Science, Umea University.

SIMULATION OVERVIEW. Microsoft . http://msdn2.microsoft.com/en-us/library/bb483076.aspx (accessed August 16, 2007).

THE GANGSTA WRAPPER. http://sourceforge.net/projects/gangsta (accessed August 14, 2007).

TOKAMAK OPEN SOURCE PHYSICS ENGINE. http://www.tokamakphysics.com/ (accessed August 14, 2007).

TRUE AXIS. http://www.trueaxis.com/ (accessed August 14, 2007).