Hindawi Publishing Corporation
International Journal of Computer Games Technology
Volume 2014, Article ID 485019, 12 pages
http://dx.doi.org/10.1155/2014/485019

Research Article

Unifying Rigid and Soft Bodies Representation: The Sulfur Physics Engine

Dario Maggiorini, Laura Anna Ripamonti, and Federico Sauro

Department of Computer Science, University of Milan, Via Comelico 39, 20135 Milan, Italy

Correspondence should be addressed to Laura Anna Ripamonti; ripamonti@di.unimi.it

Received 29 January 2014; Accepted 28 March 2014; Published 29 May 2014

Academic Editor: Ali Arya

Copyright © 2014 Dario Maggiorini et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Video games are (also) real-time interactive graphic simulations: hence, providing a convincing physics simulation for each specific game environment is of paramount importance in the process of achieving a satisfying player experience. While the existing game engines appropriately address many aspects of physics simulation, some others are still in need of improvements. In particular, several specific physics properties of bodies not usually involved in the main game mechanics (e.g., properties useful to represent systems composed by soft bodies), are often poorly rendered by general-purpose engines. This issue may limit game designers when imagining innovative and compelling video games and game mechanics. For this reason, we dug into the problem of appropriately representing soft bodies. Subsequently, we have extended the approach developed for soft bodies to rigid ones, proposing and developing a unified approach in a game engine: Sulfur. To test the engine, we have also designed and developed "Escape from Quaoar," a prototypal video game whose main game mechanic exploits an elastic rope, and a level editor for the game.

1. Introduction

A physics simulation framework is generally conceived as a middleware application. Physics simulators can be classified into two main groups: *scientific* simulators and *real-time* simulators (also called "physics engines" or simply "engines" by game developers—see, e.g., [1, 2]). A scientific simulator focuses on the accuracy of the simulation, disregarding the optimization of computational time, and its major application fields include fluid dynamics, engineering simulations, weather forecasts, and movies. On the other hand, real-time simulators (or physics engines) aim at computing as fast as possible the simulation; generally, this result is obtained by simplifying the underlying mathematical model of the simulated phenomenon. Therefore, the resulting simulation loses some accuracy. While a less accurate result could become a relevant problem in a scientific application, it becomes a by far less cumbersome issue in the area of video games. Actually, the first and main reason for a video game to exist is to provide *fun* to its players [3], which is achieved not only through alluring game mechanics, but also by providing an environment that fosters *immersivity* [4–6]. To enhance immersion in their game, designers should know everything about the physics that applies to the world they have created, in order to mimic it in the most appropriate and convincing way [7, 8]. This knowledge includes at least two synergic aspects: on the one hand, players have "a sense of how real-world works," and, on the other hand, many games include some elements of "ultraphysics," such as teleport, magic, gods intervention, faster-than-light travel, and hyperdimensionality. The physics simulated by the engine should implement all the laws that the game requires and no more, and both physics and ultraphysics laws must adhere to players' naïve physics understanding [7]. At the same time, the simulated physics—even if simplified—should guarantee that no evident discrepancies from the expected behaviours will suddenly pop up, destroying the illusion of the players.

The history of real-time engine for physics simulation is quite recent. This is due mainly to two constraints: on one hand, till the last decade, CPUs processing power was not enough to handle the heavy burden of mathematical models underpinning physics simulation, and, on the other hand, developers' attention was focused primarily on the enhancement of graphic engines that lead to the creation of

graphic accelerators and graphics processing units (GPUs) and fostered the rush toward high quality rendering and photorealism. Until 1998, the only games focusing on physics were driving simulations (one title for all: Gran Turismo [9], whose franchise reached its sixth chapter in 2013). From that year on, several engines focusing on physics started to appear, among which is the well-known Havok [10]. At the moment, a quite significant number of other engines have entered the market, among which are PhysX, Euphoria, Digital Molecular Matter, Bullet, ODE, CryEngine, Unreal Development Kit (UDK), Unity3D, and many others. Nonetheless, the interest in the improvement of engines is still high, because a good simulation increases both players' commitment in the game and the exploitation of completely new gameplays.

1.1. Structure of a Physics Engine for Video Games. Engines work on discrete intervals of time and are generally composed of three main subsystems: one aimed at calculating new positions of physical entities, one focused on detecting collisions among entities, and the last one in charge of managing collisions (see Figure 1).

The first component is a numerical integrator in charge of solving some differential equations representing movement and determining, for each frame, the new position of the moving objects. The collision detection subsystem generally implements a hierarchical data structure to simplify the search for collisions (all the pairs of objects too far to collide are excluded *a priori* from the collision detection process). The remaining objects are then tested from a geometrical point of view to verify if any intersection is taking place. The final goal is to generate a list of colliding objects, which specifies their contact points and relative velocities. This list is passed on to the collision resolution subsystem that manages the physics of the collisions (e.g., by making colliding objects bounce away or shatter). From the engine perspective, generally, in-game objects can belong to three different types: *particles*, *particles systems* (free or also connected among them), and *rigid bodies*. In this approach, soft bodies are—generally—rendered as systems of particles connected by spring joints [1, 11]. Rigid bodies are more complex than particles to represent: a (system of) particle(s) has just a *position*, while a rigid body can be seen like a particle, which has both a *position* and an *orientation* (hence, also rotation mechanics is involved). Both kinematic and dynamic can be applied to (systems of) particles and rigid bodies (see, e.g., [12]).

1.2. Simulating Rigid Bodies as Particles Systems. The most diffused approach implemented into physics engines, based on the distinction among particles and rigid bodies, has several drawbacks. As easily imaginable, the physics of (systems of) rigid bodies is quite complex, from both implementative and mathematical points of view, since it requires selecting the most effective way to manage: center of mass, momentum, and moment of inertia. To lower this complexity, we propose an innovative and unified approach rooted into the idea of representing rigid bodies through particles systems. Besides providing a simpler representation, this approach would also

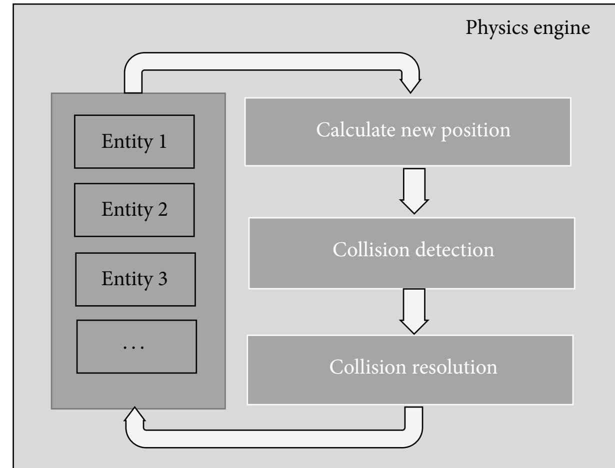
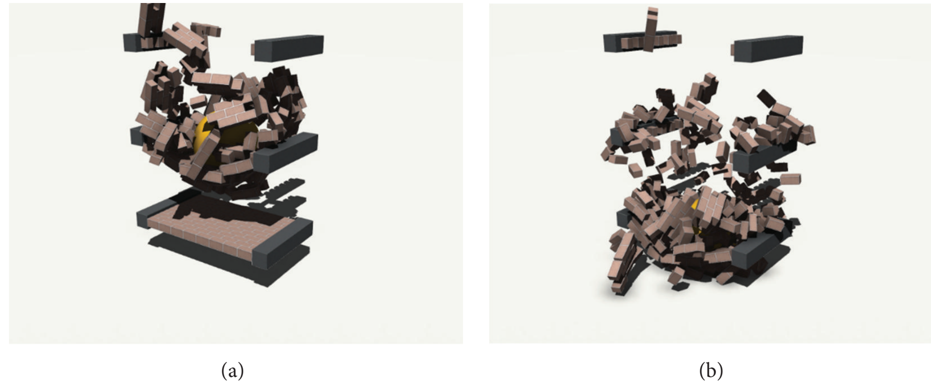


Figure 1: Structure of a general-purpose physics engine for video games.

offer the possibility to simulate in an easier way *destructible* objects (see Figure 2). These entities present several peculiarities because they start out in an undamaged state (they appear like one single cohesive object), but they must be able to break into many pieces according to different behaviours (i.e., a piece of glass shatters in a different way from a piece of wood). Anyway, as we will see, using particles systems to represent rigid bodies is not an easy task since it requires integrating impulsive forces generated by springs, a process that frequently presents problems of instability (see Section 2).

The approach we propose provides two types of joints among particles: *elastic* and *semirigid* joints. The elastic joint is a classical spring joint with a spring able to reach a good level of stiffness thanks to an opportune integration method and a high refreshing rate. The semirigid connection is assured by maintaining constant the distance among the particles in the system and by adopting the Verlet approach to integration [13–15]. We have developed and tested this new approach for games in 2D, in order to limit complexity, implementation, and testing time. Nonetheless, we have designed all the components of the engine keeping in mind the possibility to extend it quite easily to the third dimension. In particular, we have developed the following.

(i) Atlax. It is a framework designed to manage all the low-level tasks typically required by a real-time graphic application, such as interfacing the operating system for managing audiovisual input and output, windows and rendering context management, data compression, and 3D models. Particular emphasis has been put on the design of the timing system, in order to guarantee the possibility of decoupling the rendering refreshing rate from physics processing rate (that can rise up to 10 KHz). In particular, the timestamp selected for the physics simulation remains fixed for the whole simulation run. This choice has been made in order to avoid unexpected behaviors (such as objects exploding for no apparent reason). As



(a) (b)

Figure 2: Example of destructible objects: a series of walls crumbling.

a consequence, when the simulated system is very complex and the required computational time cannot cope with the predefined time interval, the simulation will only slow down, without getting unstable.

(ii) Sulfur. It is a static library that implements the physics engine. Sulfur contains several classes, implementing particles and particles systems, elastic and rigid joints, collision detection, and resolution. It is a middleware software application; hence, it is completely separated from Atlax.

(iii) SulfurChamber. It is a sandbox application aimed at providing a graphical user interface (GUI) for Sulfur, where it is possible to create and to modify physic entities.

(iv) Escape from Quaoar [16]. It is a video game based on mechanics aimed at exploiting the peculiarities of Sulfur.

The remainder of this paper is organized as follows: in Section 2, we briefly examine the state of the art in physics simulation, also digging into the main characteristics of the more diffused off-the-shelves engines. In Section 3, we will describe how the Sulfur game engine has been designed, while in Section 4 we describe the applications we have developed to implement and test our approach. Namely, they include Atlax, a low-level framework that interacts with the operating system, Sulfur, the physics engine, SulfurChamber, a sandbox aimed at experiencing and experimenting with Sulfur, and Escape from Quaoar, a complete video game based on Sulfur. Finally, in Section 5, we summarize the main results we have obtained so far and present some closing remarks and some future development for the engine.

2. State of the Art in Physics Simulation

The final goal of the majority of the game engines is the simulation of a limited number of physic phenomena, among which the most important are the kinematics and dynamics of rigid and soft bodies, represented through Newton's laws of motion and Hooke's law. The literature on these issues in the field of gaming is quite limited and more often oriented to practitioners or to teaching activities (see, e.g., [1, 2, 8, 11, 17, 18], etc.). Also, we want to point out that it is out of the main scope of the present work to simulate the dynamics of fluids, since it would imply addressing several more physics phenomena, not included in our research focus—at least for now. Fluid motion is governed by the Navier-Stokes equations, and its simulation typically requires a computational mesh of the fluid domain. Problems arise when the geometry of the fluid takes complex shapes, since mesh generation may become a major bottleneck. Moreover, exterior flow problems (like, e.g., the flow around an airfoil) require special handling. To tackle these issues, different techniques have been proposed and developed, and they are continuously improved, among which it is worth remembering at least the *smoothed particle hydrodynamics* (SPH) and the *vortex particle methods* (see, e.g., [19]). In particular, SPH has been created to simulate nonaxisymmetric phenomena in astrophysics, and it was proposed for the first time by Monaghan [20]. Since its creation, SPH has evolved into other fields, among which are fluid simulation (see, e.g., [21, 22]) and solid mechanics (see, e.g., [23–25], etc.). It has also been improved in order to support interactive simulations [26]; thus, it is frequently adopted to simulate fluid dynamics in 2D video games, even for mobile devices. Both SPH and vortex particle methods are mesh-free interpolation techniques, which use "particles" to represent parcels of fluid. Their main difference lies in the fact that SPH solves the momentum equation, while vortex methods solve the vorticity equation. Furthermore, in contrast to vortex methods, each particle in SPH shows only a local influence, thus making it easier to handle collisions with, for example, the boundaries of a container. Nonetheless, their main application field is fluids simulation or—at most—the simulation of fractures of rigid bodies; in both cases, accuracy of the simulation is more important than its velocity.

The game objects managed by an interactive physics simulation engine are, as we already saw (Section 1), of different types and natures (generally particles or rigid bodies) and belong to different categories, which require specific interactions with the engine. In particular, it is important to distinguish between *physics-driven* and *game-driven* objects. Physics-driven objects completely depend on the simulator (e.g., a stone rolling down from a slide), while game-driven

objects are not subject to the simulation, because they are usually controlled by the player (e.g., a sword wielded by a character); nonetheless, they may have physics interactions with the environment (e.g., the player uses the sword to break a chest). Last but not least, the environment usually contains a certain number of "fixed" objects (e.g., the floor) that will interact only with the collision management system. It is important to notice that the collision management system should be present in whichever game engine, even if the physics is not applied, in order to determine superimposition and interaction among game objects. For this reason, the collision management system is usually separated from the remainder of the engine.

The adoption of a physics engine may have some relevant implications. The user should be well aware of these implications, starting from the moment she is starting designing a new game. In particular, the presence of the engine introduces *unpredictability*, *emerging behaviours*, and the necessity to *balance the values assumed by several variables* [2]. As a matter of fact, an object whose behavior is governed by some approximated physics law may, from time to time, show a behavior that is not desired or that is unexpected. In these cases, the best solution is to "overwrite" the simulation with predefined animation. In the same vein, players may exploit some physic phenomena to their advantage, bending the gameplay in ways not foreseeable by the game designer (an example for all is the following: players may aim a rocket launcher at the floor to get a boost for achieving a higher jump). Finally, yet importantly, in a physics simulation, there are many variables (e.g., friction, gravity, etc.) whose values may affect in many ways the overall game system. It is of paramount importance that, during the design and testing phases of the game, these values are properly balanced in order to produce an environment that matches the perception of physics laws that the player has for that specific game.

As we saw, the main purpose of a physics engine is to approximate bodies dynamics, that is to say, how they behave when they are subjects to a system of forces. Hence, we need, on the one hand, to determine, for each instant of time, position, velocity, and rotation for each body, and, on the other hand, to detect and resolve possible collisions among bodies. To obtain these goals, a real-time physics simulation is based on appropriate numerical resolution techniques for motion laws aimed at refreshing, at each interaction, position, velocity, and orientation in space of an object starting on its previous state and on the forces applied on it. Actually, it is important to underline that, since game engines are interactive simulators, it is not possible to foresee how the system of forces applied to each in-game object will evolve in time. For this reason, it is necessary to approximate the future state by applying numerical methods to the integration of differential equations representing motion laws. There is a certain number of different possible approaches to numerical integration, and each method shows different qualities and drawbacks. Digging deeply into the numerical integration issue is out of the scope of the present work. Nonetheless, we will now briefly go through the most diffused approaches adopted in the field of physics simulators for video games, highlighting the main features of each approach. Numerical methods are all based on Taylor's polynomial evidenced in Taylor's theorem. They can be classified on the basis of their *order*, *convergence,* and *stability* (for a detailed description of numerical integration methods, see, e.g., [17]).

The simplest numerical integrator is the *Euler* method (also called *Explicit Euler* method), a first order method usually adopted by programmers that are tackling for the first time physics simulation. In spite of its simplicity and execution velocity, this method is highly inaccurate, since it focuses on velocity to determine the next position of an object.

The *SUVAT* (displacement S, initial velocity U, final velocity V, acceleration A, and time T) method is another first order method, slightly more accurate than the previous one, since it takes into consideration also acceleration. Nonetheless, it is quite unstable and it works well only when acceleration is constant along the integration interval.

The *symplectic Euler* method (also called *semi-implicit Euler* method or *Newton-Størmer-Verlet* (NSV) method) is another first order method, quick and simple, but—due to the fact that it is a symplectic method—by far more stable than the previous approaches. This method behaves quite well also when simulating oscillatory behaviours (like in the case of springs).

The *Runge-Kutta 2* method (or *Midpoint* method) is more accurate than its predecessors (it is a second order integrator), but it is also slower.

The Runge-Kutta 4 method is a fourth order integrator; hence, it becomes useful only when execution velocity is not a constraint, but accuracy is very relevant.

Finally, yet importantly, the class of the *implicit* methods guarantees a pretty good stability, even when rigid springs are involved, but sacrifices simplicity of implementation and velocity of computation.

2.1. Characteristics of the More Diffused Off-the-Shelves Engines. Before diving into the design and development from scratch of a real-time interactive physic simulator, we have taken into account the existing frameworks used for physics simulation in the video games industry. We have compared the characteristics of the more diffused ones among them, with the aim of verifying if at least one of them could match our specific requirements. The requirements we were looking for can be summarized as follows:

(1) efficient support to *physics simulation in 2D* (eventually easily extensible to 3D);

(2) *support for both rigid and soft bodies*, with a seamless gamma of stiffness;

(3) *two-way interaction between rigid and soft bodies.*, that is to say, the capability of soft bodies to both collide and exert forces on rigid bodies and vice versa;

(4) *affordability for a small indie team* with low budget. This requirement mirrors the fact that big players in the video game industry usually develop and maintain their own solutions for physics simulation or buy third party software applications.

All the off-the-shelves engines we have examined are unable to satisfy the whole range of constraints we have set, even the most renowned applications (such as Unity3D), as Table 1 sums up (the requirements listed above have been put in columns). Here, below, we list their major characteristics.

Box2D [27] is free to use and is tailored for 2D, but it is lacking soft-body simulation.

Bullet [28] is a widely used open-source physics simulation library, and it has been used for many games. It recently added support to soft bodies, and it is a good candidate to compare our results with. There is also limited support to 2D environments, but only for rigid bodies. Although with some tricks and some new code it could be possible to use it anyway, it would not satisfy our requirement (1): efficient and native support to 2D soft-body simulation.

CryEngine 3/BeamNG [29] is an extremely powerful game engine, one of the market leaders. It has a pretty new soft-body physics simulator provided by the team of Rigs of Rods (BeamNG). Unfortunately, the licensing system for indie developers is not completely clear, and it is very likely that it does not provide the full features set that the paid version offers. Moreover, it would be absolutely overdimensioned and cumbersome to use for a 2D game.

Digital Molecular Matter (DMM) [30] is different from the previous engines, because it is based on a simulation method called finite element analysis [31–34], which is far more accurate than the ones usually adopted in game engines. It aims to achieve realistic results and supports a wide range of platforms. It matches part of our requirements, but, unfortunately, it is not free to use and again does not support 2D environments.

Havok [10] is one of the market leaders. It runs on every platform (mobiles as well) and provides an excellent combination of performance and accuracy. The core system handles rigid-body dynamics, and it can be augmented with a set of subsystems, like the recently added *Cloth* module, which is aimed at simulating character's clothes and hair. It is not very affordable, and it does not satisfy both requirements (1) and (2): there is a clear separation between rigid and soft bodies and it can be applied only to 3D environments.

Newton Game Dynamics [35] is another open-source library, not supporting soft bodies. It is a bit outdated one also.

Open Dynamics Engine (ODE) [36] is free to use, but it is outdated and only supports rigid bodies.

PhysX [37] is another market leader. It does not support as many platforms as Havok, and its main feature—GPU acceleration—is currently only available using Nvidia video cards. The features set we are looking for seems to be more than fully implemented in an upcoming extension called *FLEX*, which promises unified rigid-body, soft-body, and fluid simulation, but—unfortunately—it is not available on the market yet, and it does not seem to cover 2D simulations. Moreover, the product is free only when it is used to develop for the Windows operating systems.

Unreal Development Kit (*UDK)* [38] is another powerful and widely used game engine that supports our required features. Unfortunately, although it offers a license tailored for indie developers, it is not as cheap as several other solutions and it is known to be quite cumbersome to use. Again, it is not targeted at 2D.

Unity3D [39] is one of the most popular game development frameworks currently around. It supports every platform and provides a cheap license for independent developers. It provides a basic physics module that does not include any kind of soft-body simulation.

3. Designing the Sulfur Real-Time Interactive Physics Engine

As we saw, the spikiest issues in physics simulation for video games are the management of the rotation component in the motion of rigid bodies, the accurate detection of collisions (especially when complex nonconvex objects are involved [40]), and a convincing visual representation of collisions among rigid bodies.

The purpose of a physics engine is to approximate the dynamic behaviours of objects subject to a set of forces. The main idea behind Sulfur is to deploy an alternative approach to rigid bodies simulation by extending the methodology commonly applied to soft bodies, in order to avoid calculating the rotation component of motion. Actually, this effect would emerge spontaneously from a particle system, in which each particle is linked to some others and translates when subject to forces.

3.1. Simulating Soft Bodies with Sulfur. A particle is described by its *mass*, *position,* and *velocity* (the latter two are represented by bidimensional vectors in the 2D case). Moreover, when a particle is connected to some others, topological information is needed too. To represent a deformable or soft body (like, e.g., some jelly), we need a set of particles connected by some elastic joint. To describe the constraints linking two particles connected by an elastic (spring) joint, we can use Hooke's law that is corrected with the damping. The resulting equation is

$$F(t, x(t), \dot{x}(t)) = -k(|d| - l_0)\vec{d} + b(\dot{x}(t) - \dot{x_p}(t)), \quad (1)$$

where

$F = -k(|d| - l_0)\vec{d}$ is Hooke's law;

$F(t, \dot{x}(t)) = -b\dot{x}(t)$ is the damper;

$\dot{x_p}(t)$ is the velocity of the other particle connected to the joint;

t is time;

k is the elastic constant;

l_0 is the length of the spring;

d is the variation in the length of the spring;

b is the damping constant.

Adding the damper is extremely useful, not only because it increases the realism of the resulting simulated behavior, but also—and perhaps mainly—because it helps mitigate oscillations intensity and duration. Since it is our aim to simulate also rigid bodies using particle systems, it is of

Table 1: Comparison of the more diffused game engines based on the requirements we have defined.

	Requirements			
	1	2	3	4
Box2D	Yes	No	No	Yes
Bullet	Partial	Yes in 3D	Yes in 3D	Yes
CryEngine 3/BeamNG	No	Yes	Yes	Partial
Digital Molecular Matter (DMM)	No	Yes	Yes	No
Havok	No	No	No	No
Newton Game Dynamics	Yes	No	No	Yes
Open Dynamics Engine (ODE)	Yes	No	No	Yes
PhysX	No	Upcoming	Upcoming	Only Windows
Unreal Development Kit (UDK)	No	Yes	Yes	Partial
Unity3D	Yes	No	No	Yes

Figure 3: The rope is composed of 40 particles used to tune damping and refreshing rate in its starting state.

fundamental importance to be sure that the simulation will behave appropriately in a large interval of possible values for the elastic constant. For this reason, it has been necessary to balance damping, to define the most appropriate refreshing rate for the simulator, and to select the best possible numeric integrator, that is to say, the one that guarantees the most correct behavior while, at the same time, presenting good performances in terms of calculation time (we are dealing with real-time systems). To obtain these goals, we have tested different configurations of damping and refreshing rate on a linear configuration composed of a system of 40 particles, which simulates a rope (dots in Figure 3), connected by elastic joints (dashes in Figure 3). The starting state of the rope is horizontal and at rest (the spring between two consecutive particles is at rest; hence, the distance between the particles coincides with the length of the joint). When the simulation is started, both gravity and air friction start to affect the system (Figure 4). In particular, the air friction constant value has been set to 0.02, the length of the spring to 0.05 m, and the mass of the particles to 0.05 kg. It is quite easy to spot the moment in which the system loses stability: the rope vibrations get uncontrollable and the rope explodes due to growing elastic snaps (Figure 5). Table 2 summarizes the maximum values for the elastic constant when the refreshing rate is set to 500 Hz, while Table 3 summarizes the same values when the damping constant is set to 0.2. It is possible to notice that the values smaller than 50.0 (for the elastic constant) are not included, since, in such cases, the spring is too weak and its simulation loses any significance.

From Table 2, it is possible to notice that the accuracy of the integrator does not imply stability; for example, Runge-Kutta 4 (RK4) is a fourth order integrator, but it is quite ineffective with a high stiffness. In the same vein, NSV and inverse Euler (IE) are by far more effective than RK4, even if they both are only first order integrators. When the refreshing rate is taken into consideration (Table 3), it becomes clear that

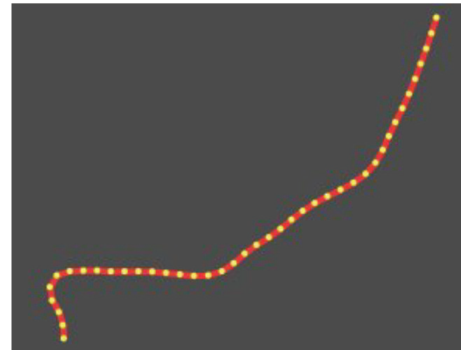


Figure 4: The rope oscillates correctly under the effects of friction and gravity.

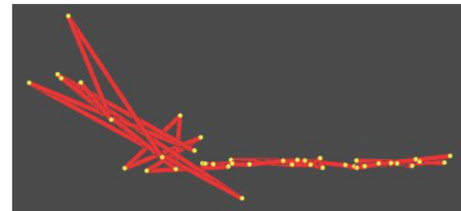


Figure 5: The rope explodes because the system has become instable.

semi-implicit integrators behave by far better than explicit integrators: in the first case, it is possible even to raise the value of the elastic constant six times without losing stability, while in the latter case that value can be—at most—doubled.

For these reasons, we have chosen to set the default refreshing rate at 500 Hz and the damping constant to 0.2 and to implement the symplectic Euler integrator (NSV).

Last but not least, it is important to underline that, for both soft and rigid bodies represented by means of a particles system, it is necessary to introduce some "internal" joint that will guarantee that the object will not collapse on itself after a collision. This phenomenon is depicted in Figure 6: the deformable square on the left of the figure has collided with the floor, but it has no internal joint among the particles, while

Table 2: Maximum values of the elastic constant with variable damping and fixed refreshing rate (500 Hz).

	Damping				
	0.0	0.1	0.2	0.3	0.4
Eulero	—	50	100	150	200
SUVAT/RK2	—	100	200	300	400
RK4	—	150	300	450	600
NSV/IE	50	8500	9000	9500	11000

Table 3: Maximum values of the elastic constant with variable refresh rate and fixed damping (0.2).

	Frequency (refresh rate—Hz)				
	60	100	250	500	1000
Eulero	—	—	50	100	1000
SUVAT/RK2	—	—	100	200	400
RK4	—	60	150	300	600
NSV/IE	150	450	2700	8000	47500

the deformable square on the right has—correctly—bounced away after the collision.

3.2. Simulating Rigid Bodies with Sulfur. Increasing the stiffness of the elastic joint alone is not enough to simulate effectively rigid bodies by means of a particles system. Actually, not only the stiffness should be enough to avoid deformation of objects during collisions, but also the simulation should behave properly when the system includes particles with a very different mass. In this latter case, the lighter particles will accelerate more than the heavier ones, even if they are all subject to the same force. Consequently, the system will unbalance and become instable. To avoid these undesirable effects, we are forced to introduce a new type of joint and, subsequently, to select an integrator that is able to handle it adequately. In particular, the new joint is a spring with infinite stiffness. In this case, the distance of two particles situated at the two ends of a joint is always constant, and Hooke's law is no longer useful. This particular joint will not generate a force (like in the case of the soft body), but an instantaneous change in velocity, thus modifying particles positions. Therefore, the integrator we have applied to soft bodies is no longer useful. As a matter of fact, traditional integrators are based on the assumption that the instantaneous acceleration is enough to describe completely the particle movement; in the case of this new joint, these integrators will lose the kinetic energy produced by the stiff spring, bringing highly inaccurate results.

A valid choice for an alternative integrator is the *Verlet* method [14, 15], a symplectic integrator of the second order, that is able to calculate the new position $x(t+\Delta t)$ of a particle after a certain interval of time has passed. The Verlet method is based on Taylor's theorem; it is applied two times: the first time forward (in time) and then backward (in time), as described by the following formulae:

$$\begin{aligned} x(t+\Delta t) &= x(t) + \dot{x}(t)\Delta t + \frac{1}{2}\ddot{x}(t)\Delta t^2 + \frac{1}{6}x^{(3)}(t)\Delta t^3 + O\left(\Delta t^4\right), \\ x(t-\Delta t) &= x(t) - \dot{x}(t)\Delta t + \frac{1}{2}\ddot{x}(t)\Delta t^2 - \frac{1}{6}x^{(3)}(t)\Delta t^3 + O\left(\Delta t^4\right). \end{aligned} \quad (2)$$

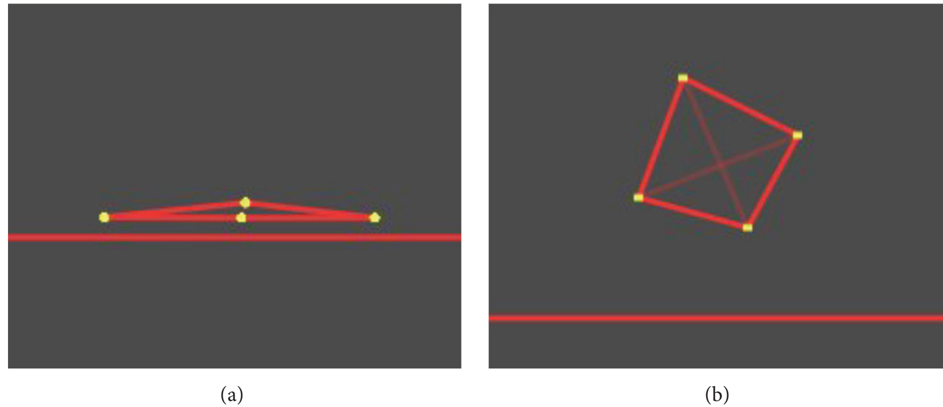
Hence,

$$x(t+\Delta t) = 2x(t) - x(t-\Delta t) + \ddot{x}(t)\Delta t^2 + O\left(\Delta t^4\right). \quad (3)$$

In (3), velocity does not appear explicitly; if needed, it can be derived from the difference between the starting and final positions of the particle(s). It is precisely for this reason that the Verlet method works well with our rigid joint: the integrator takes into account both forces and variations in particles positions without losing any kinetic energy. The *Velocity Verlet* and *Leapfrog* integrators are more accurate versions of the Verlet integrator [17], but they are more demanding from a computational point of view, without supplying any significant improvement to the stability of the system. For these reasons, the Sulfur engine is based on the Verlet method.

3.3. Managing Collisions with Sulfur. Even if we have chosen to create an engine aimed at 2D environments, the idea to simulate any type of body using only particles systems requires a careful analysis of all the possible implications and consequences on the collision management system. In particular, our approach allows the creation of particles aggregates with arbitrary shapes (linear, concave, and convex). Moreover, we intend to exploit the intrinsic modular nature of these aggregates to simulate fractures. Therefore, we need to apply a collision detection system that is able to track even the smallest possible aggregate: two particles connected by a joint. It is possible to notice that a single particle is an immaterial point with infinitesimal size; hence, it is not subject to collisions. These peculiarities imply that applying the most diffused techniques for collision detection (such as *Bounding Volume Hierarchy (BVH), Sweep And Prune (SAP), Separating Axis Theorem (SAT), and Gilbert-Johnson-Keerthi (GJK) distance algorithm*; see, e.g., [41–48]) is not a viable way. In the same vein, the *Bentley-Ottmann* algorithm, also called *Sweep-Plane* algorithm [49], which is aimed at finding intersections between segments on a plane, is not adaptable to our specific case. In particular, it considers only one-dimensional segments that cannot cope with some extreme case, such as those depicted in Figure 7. The segment on the left of the figure would pass through the floor unnoticed, unless we release the constraint of one dimensionality. Similarly, the segment on the right would cross, unnoticed, the joint in the floor.

Since no standard method to manage collision detection seems to be able to support our approach to rigid-body



(a) (b)

Figure 6: A deformable soft square—with and without internal joint—collides with the floor.

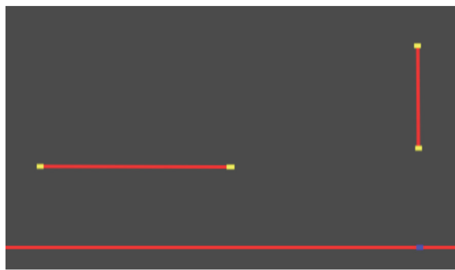


Figure 7: Two segments that should collide with the floor.

representation, we have developed an original approach, which preserves the two "canonical" phases: *broad* and *narrow* phases of collision detection.

The first—*broad*—phase of collision detection is aimed at detecting the objects potentially superimposing. To avoid the trap of one-dimensional segments, we include each segment in a "capsule" (see Figure 8(a)), which simulates thickness. We then subdivide the plane with a uniform grid and keep track of the cells crossed by each segment in each moment (see Figure 8(b)). When a cell registers a collision, the capsules are superimposing and they are selected and included in a list of objects that will be further processed during the following *narrow* phase. To guarantee that the broad phase is executed in the shortest possible time, we have implemented the *Bresenham Line* algorithm [50], which supplies good performances, with an execution time which depends linearly from the number of capsules present on the grid.

The subsequent *narrow* phase of collision detection takes as input the cells selected during the broad phase and verifies whether or not the couple of segments is really in contact and, in case, determines the contact point.

Finally, the *reaction* to the collision is calculated according to a projection approach: the bodies are moved away from each other the minimum distance necessary to separate them. The joint among the particles will guarantee that the whole object will move away in a convincing way.

To add realism to the simulation of collisions, we have also added a simplified version of a friction model. We take into account only sliding friction, without any distinction between static and dynamic frictions. The main idea is to apply to the particles a force that is tangent to the direction of the collision and proportional to the projection of velocity on this tangent, but in the opposite direction. This is only a first attempt to include friction in our simulator, and it is under improvement for further developments.

4. Atlax, Sulfur, SulfurChamber, and Escape from Quaoar

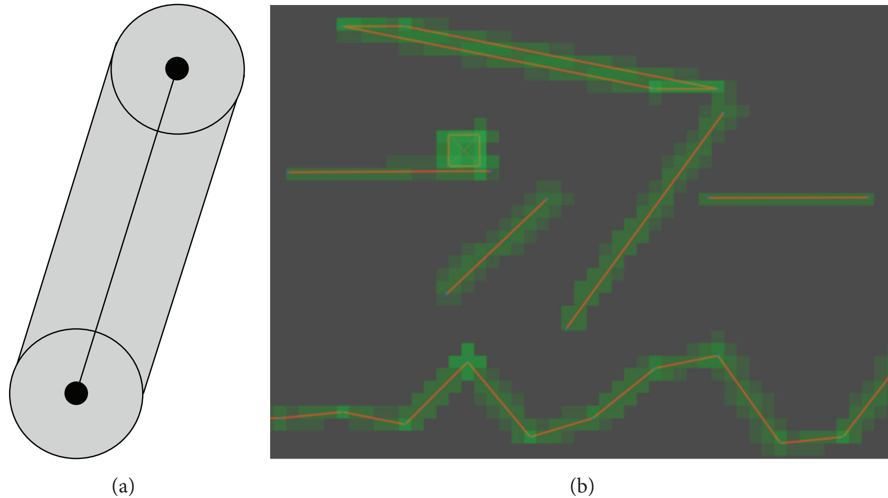
Both Atlax and Sulfur have been developed in C++, which is the most diffused language for computational intensive applications and for interactive physics simulations. All the libraries we have adopted are independent of the operating system in order to provide the maximum possible portability. All the codes we have developed have been tested on both Windows and Linux.

Atlax is a framework application aimed at supporting the development of interactive graphic applications. Its most relevant feature is to be based transparently on two low-level different libraries: *wxWidgets* [51], a cross-platform graphical user interface (GUI) library, and *Simple DirectMedia Layer* [52], a cross-platform library designed to provide low-level access to audio, keyboard, mouse, joystick, and graphics hardware via OpenGL [53] and DirectX [54]. The first library is used when we need to create applications with a GUI, such as a sandbox for the physics engine. The second library is specifically aimed at video games. Therefore, this approach made it possible to create an engine that can be used both for the real game and for the editor, just by switching the underpinning library.

Sulfur is the library of the physics engine. It is strongly linked to Atlax, since the two share some classes (notably to execute calculus and to dynamically manage particles and joints). It implements the approach we have summarized in Section 3.

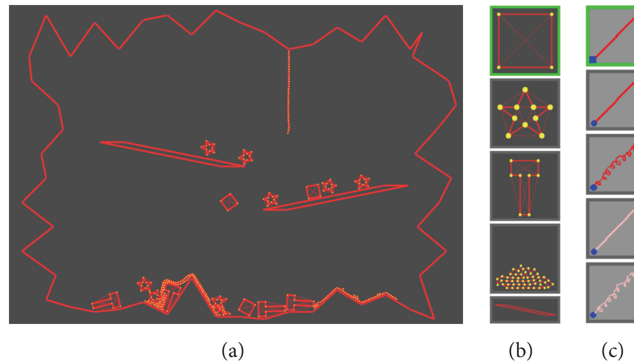
Based on Atlax and Sulfur, we have developed two applications aimed at testing the effectiveness of the approach: SulfurChamber and Escape from Quaoar.

SulfurChamber is the sandbox software application for the Sulfur physics engine, and it is based on the Atlax-wxWidgets



(a) (b)

FIGURE 8: A single capsule surrounding a segment (a) and the representation of capsules on the grid.



(a) (b) (c)

FIGURE 9: Soft and rigid bodies in SulfurChamber.

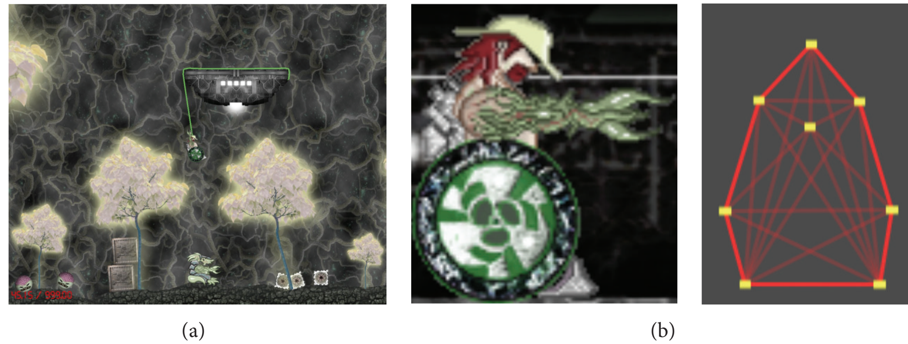
version. Its main purpose is to allow the experimenting of different configurations of particles with varying elasticity, gravity, sliding friction, and air friction (Figure 9(a)). In particular, it is possible to create objects starting from a predefined set (Figure 9(b)) or by designing them from scratch by combining sets of particles and joints (Figure 9(c)). We have tested the performances of SulfurChamber on several different combinations of hardware and operating system, that is to say, AMD Atlon64 3200 with Windows XP, Intel Core2Duo T7700 with Windows 7, and Intel Core-i7 930 with Linux-Ubuntu. The simulation has been run by simulating an increasing number of identical squared boxes (each of which is composed of four particles: four main rigid joints and two internal structural joints). Couples of boxes were dropping from above continuously; hence, no box was in a state of rest (this would have the consequence of excluding the box from the physics simulation). The values of friction, mass, and gravity were set to the same values for the whole simulation. Table 4 summarizes the percentage of consumption of the CPU time dedicated to executing calculations to refresh the physics simulation. From the data emerges that the maximum numbers of boxes for which the system remains stable and reactive are around 60 or less, depending on the combination

TABLE 4: Performance of Sulfur with different hardware and software configurations.

	Number of dropping boxes per second						
	20	40	60	80	100	150	200
Athlon64—Win XP	12%	30%	45%	65%	83%	100%	100%
Core2Duo—Win 7	10%	22%	42%	55%	72%	95%	100%
i7-930—Ubuntu	5%	10%	20%	25%	35%	50%	75%

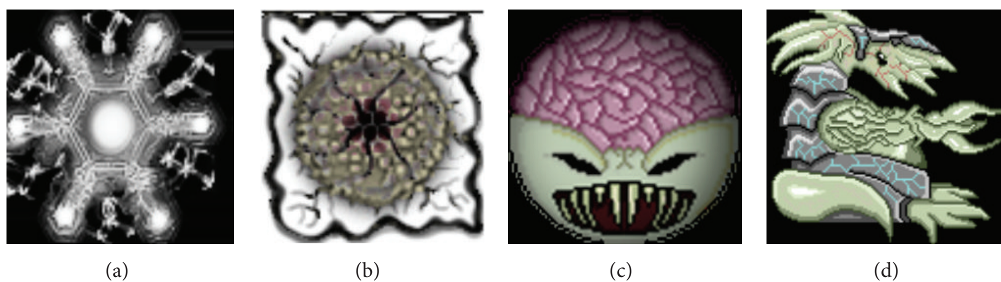
hardware-operating system (when the CPU consumption reaches 80%–90%, the simulation obviously starts to slow down, till getting irremediably stuck). Even if this maximum number of objects was enough for the video game we have developed to test the opportunities offered by Sulfur, it is evident that the physics engine would benefit a lot from some improvements aimed at optimizing code and performances.

Escape from Quaoar [16] is a side-scroller platform video game that has been designed and developed to test the performances of Sulfur in a real application. For this reason, the gameplay explicitly exploits and stresses the whole range of features provided by the physics simulator. The core mechanic is based on an elastic rope (see Figure 10(b)), that



(a) (b)

Figure 10: Screenshots from Escape from Quaoar: example of a level (a) and the main character texture and underpinning particle system (b).



(a) (b) (c) (d)

Figure 11: The first three objects have been modeled by elastic aggregates of particles, while the latter one is a rigid body, which stays fixed on the floor.

is, the only locomotion mean at disposal of the player. All the in-game objects are physics-driven (except for the main character arm that is moved by the player to fire the rope towards specific points). Monster and objects encountered in the levels are—soft or rigid—particles systems masked by appropriate textures (see Figure 11(a)). The overall behavior of the game is very satisfying, and the game has participated in the 13th Independent Game Festival (IGF) [55].

5. Conclusion and Future Work

The main goals of our work have been to design and create a physics simulator for video games that are able to unify rigid and soft bodies simulation, endowed with effective and reactive collision detection and resolution system. The Sulfur physics engine has reached this goal, overcoming the stiffness problem intrinsic to traditional approaches by using the Verlet integrator coupled with the adoption of particles systems to simulate not only soft bodies, but also rigid ones. This approach has an interesting side effect: it does not mess up with angles and rotations. In the same vein, we have obtained a sound approach to collision management by coupling a static grid with Bresenham algorithm. Hence, with a relatively small effort, we are able to manage a relevant number of dynamic objects. Last but not least, we have extensively tested the proposed solution by developing 26 levels for a side-scroller video game, whose mechanics is based on elastic bodies, obtaining satisfying results.

Nonetheless, we are well aware that Sulfur is still in its infancy. It needs improvements and developments from several points of view, such as the following:

(i) increasing accuracy of simulation for friction;

(ii) creating an efficient and effective approach to manage curve and circular objects (at present they can only be simulated with a huge number of particles and joints);

(iii) optimizing calculation-intensive code section, by moving the workload to the GPU;

(iv) introducing angular joints, to decrease the number of particles used to represent a single object;

(v) introducing a more sophisticated simulation of fractures (at the moment, fractures are obtained by removing some joints);

(vi) extending Sulfur to 3D simulation;

(vii) extending Sulfur to simulate fluids motion;

(viii) finally, yet importantly, reducing the refreshing rate (whose value at present is around 500 Hz), without scarifying stability while—possibly—increasing performances. This goal could be reached by selecting an appropriate, but more complex, integrator that could reduce the number of iterations.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

References

[1] D. H. Eberly, *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics*, Morgan Kaufmann, San Francisco, Calif, USA, 2001.

[2] J. Gregory, Ed., *Game Engine Architecture*, Taylor and Francis, 2009.

[3] R. Koster, *A Theory of Fun for Game Design*, Paraglyph Press, 2005.

[4] M. Csikszentmihalyi, *Flow: The Psychology of Optimal Experience*, Harper Perennial, 1991.

[5] T. Fullerton, *Game Design Workshop: A Playcentric Approach to Creating Innovative Games*, Elsevier, 2008.

[6] E. Zimmerman and K. Salen, *Rules of Play: Game Design Fundamentals*, The MIT Press, 2004.

[7] R. Bartle, *Designing Virtual Worlds*, New Riders Pub, 2003.

[8] I. Millington, *Game Physics Engine Development*, Morgan Kaufmann, San Francisco, Calif, USA, 2007.

[9] Gran Turismo, http://www.gran-turismo.com/.

[10] Havok, http://www.havok.com/.

[11] D. H. Eberly, *3D Game Engine Architecture: Engineering Real-Time Applications with Wild Magic*, Morgan Kaufmann, San Francisco, Calif, USA, 2005.

[12] A. Witkin and D. Baraff, "Physically based modeling: principles and practice," in *Proceedings of the Association for Computing Machinery Special Interest Group on Graphics (SIGGRAPH '97)*, Los Angeles, Calif, USA, 1997.

[13] T. Jakobsen, "Advanced Character Physics," 2003, http://www.pagines.ma1.upc.edu/~susin/files/AdvancedCharacterPhysics.pdf.

[14] M. Mcguire and O. C. Jenkins, *Creating Games: Mechanics, Content, and Technology*, A K Peters, 2008.

[15] L. Verlet, "Computer "experiments" on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules," *Physical Review*, vol. 159, no. 1, pp. 98–103, 1967.

[16] Escape from Quaoar, http://www.escapefromquaoar.com/.

[17] D. H. Eberly, *Game Physics*, Morgan Kaufmann, San Francisco, Calif, USA, 2003.

[18] D. Kodicek, *Mathematics and Physics for Game Programmers*, Charles River Media, Hingham, Mass, USA, 2005.

[19] G. H. Cottet and P. D. Koumoutsakos, *Vortex Methods. Theory and Practice*, Cambridge University Press, 2000.

[20] J. J. Monaghan, "Smoothed particle hydrodynamics," *Annual Review of Astronomy and Astrophysics*, vol. 30, no. 1, pp. 543–574, 1992.

[21] A. Hérault, G. Bilotta, A. Vicari, E. Rustico, and C. del Negro, "Numerical simulation of lava flow using a GPU SPH model," *Annals of Geophysics*, vol. 54, no. 5, pp. 600–620, 2011.

[22] E. Rustico, G. Bilotta, A. Hérault, C. Del Negro, and G. Gallo, "Smoothed particle hydrodynamics simulations on multi-GPU systems," in *Proceedings of the 20th Euromicro International Conference on Parallel, Distributed and Network-Based Computing, Special Session on GPU Computing and Hybrid Computing*, Munich, Germany, 2012.

[23] J. Bonet and S. Kulasegaram, "Correction and stabilization of smooth particle hydrodynamics methods with applications in metal forming simulations," *International Journal for Numerical Methods in Engineering*, vol. 47, no. 6, pp. 1189–1214, 2000.

[24] M. I. Herreros and M. Mabssout, "A two-steps time discretization scheme using the SPH method for shock wave propagation," *Computer Methods in Applied Mechanics and Engineering*, vol. 200, no. 21-22, pp. 1833–1845, 2011.

[25] T. Rabczuk and J. Eibl, "Simulation of high velocity concrete fragmentation using SPH/MLSPH," *International Journal for Numerical Methods in Engineering*, vol. 56, no. 10, pp. 1421–1444, 2003.

[26] M. Muller, D. Charypar, and M. Gross, "Particle-based fluid simulation for interactive applications," in *Proceeding of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pp. 154–159, Eurographics Association, 2003.

[27] Box2D, http://box2d.org/.

[28] Bullet, http://bulletphysics.org/wordpress.

[29] Cryengine, http://www.crytek.com/cryengine.

[30] "Digital Molecular Matter—Pixelux," http://www.pixelux.com/.

[31] I. Babuška, U. Banerjee, and J. E. Osborn, "Generalized finite element methods: main ideas, results, and perspective," *International Journal of Computational Methods*, vol. 1, no. 1, pp. 67–103, 2004.

[32] K. J. Bathe, *Finite Element Procedures*, Klaus-Jürgen Bathe, Cambridge, Mass, USA, 2006.

[33] J. N. Reddy, *An Introduction to the Finite Element Method*, McGraw-Hill, New York, NY, USA, 3rd edition, 2005.

[34] O. C. Zienkiewicz, R. L. Taylor, and J. Z. Zhu, *The Finite Element Method: Its Basis and Fundamentals*, Butterworth-Heinemann, 6th edition, 2005.

[35] Newton Game Dynamics, http://newtondynamics.com/forum/newton.php.

[36] Open Dynamics Engine—ODE, http://www.ode.org/.

[37] PhysX, http://developer.nvidia.com/object/physx.html.

[38] "Unreal Development Kit—UDK," http://www.unrealengine.com/en/udk/.

[39] Unity3D, http://unity3d.com/.

[40] I. Millington and J. Funge, *Artificial Intelligence for Games*, Morgan Kaufmann, 2nd edition, 2009.

[41] D. Baraff, *Dynamic simulation of non-penetrating rigid bodies [Ph.D. thesis]*, Computer Science Department, Cornell University, 1992.

[42] J. D. Cohen, M. C. Lin, D. Manocha, and M. Ponamgi, "I-COLLIDE: an interactive and exact collision detection system for large-scale environments," in *Proceedings of the Symposium on Interactive 3D Graphics*, pp. 189–196, Monterey, Calif, USA, April 1995.

[43] E. Christer, *Real-TimeCollision Detection*, Morgan Kaufmann series in interactive 3D technology, Elsevier, Amsterdam, The Netherlands, 2005.

[44] E. G. Gilbert, D. W. Johnson, and S. S. Keerthi, "A fast procedure for computing the distance between complex objects in three-dimensional space," *IEEE Journal of Robotics and Automation*, vol. 4, no. 2, pp. 193–203, 1988.

[45] Golshtein and E. G. Tretyakov, *Modified Lagrangians and Monotone Maps in Optimization*, Wiley, New York, NY, USA, 1996, translated by N. V. Tretyakov.

[46] J. Günther, S. Popov, H.-P. Seidel, and P. Slusallek, "Realtime ray tracing on GPU with BVH-based packet traversal," in *Proceedings of the IEEE/Eurographics Interactive Ray Tracing Symposium (IRT '07)*, pp. 113–118, September 2007.

[47] K. Shimizu, Y. Ishizuka, and J. F. Bard, *Nondifferentiable and Two-Level Mathematical Programming*, Kluwer Academic Publishers, Boston, Mass, USA, 1997.

[48] G. van den Bergen, *Collision Detection in Interactive 3D Environments*, Morgan Kaufmann, San Francisco, Calif, USA, 2003.

[49] U. Bartuschka, K. Mehlhorn, and S. Naher, "A robust and efficient implementation of a sweep line algorithm for the straight-line segment intersection problem," in *Proceedings of the Workshop on Algorithm Engineering*, S. Orlando, Ed., 1997.

[50] J. E. Bresenham, "Algorithm for computer control of a digital plotter," *IBM Systems Journal*, vol. 4, no. 1, pp. 25–30, 1965.

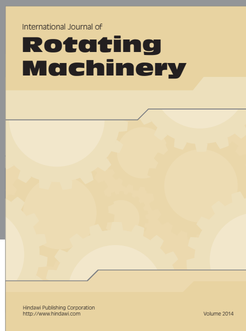
[51] wxWidgets, http://www.wxwidgets.org/.

[52] SDL, 2013, http://www.libsdl.org/.

[53] OpenGL, http://www.opengl.org/.

[54] Direct3D, http://www.microsoft.com/en-us/download/details.aspx?id=23803.

[55] "IGF—Independent Games Festival," http://www.igf.com/.



Hindawi

Submit your manuscripts at
http://www.hindawi.com