Unifying rigid and soft bodies representation: the Sulfur physics engine

Dario Maggiorini, Laura Anna Ripamonti, Federico Sauro

Dep. of Computer Science, Università di Milano

Via Comelico, 39 – 20135, Milano, Italy

dario@di.unimi.it, ripamonti@di.unimi.it, federico.sauro@gmail.com

Keywords: video game, game engine, physics simulation

Abstract

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 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, which are only rarely involved in the main game mechanics – such as those 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. [16], [25]). 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, movies, etc. 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 underling 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 [33], that is achieved not only through alluring game mechanics, but also by providing an environment that fosters *immersivity* [13], [21], [55]. 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 [3], [35]. 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, hyperdimensionality, etc. 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 [3]. At the

same time, the simulated physics – even if simplified – should guarantee 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 GPUs (Graphics Processing Units) 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 [23], 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 [27]. At the moment a quite significant number of other engines have entered the market, among which PhysX, Euphoria, Digital Molecular Matter, Bullet, ODE, Cryengine, UDK - Unreal Development Kit, Unity3D, and many others. Nonetheless, the interest in the improvement of engines is still high, because a good simulation both increases players' commitment in the game and the exploitation of completely new gameplays.

1.1 Structure of a physics engine for video games

Engines works on discrete intervals of time and are generally composed by 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 Fig. 1).

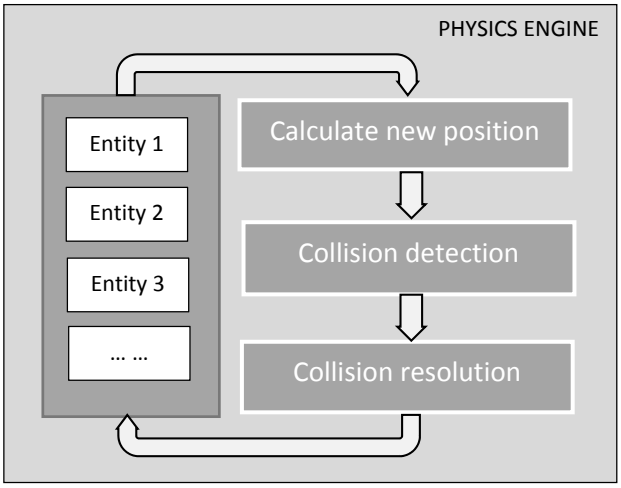

Fig. 1 – Structure of a general-purpose physics engine for video games

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 [16], [18]. 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. [52]).

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, both from an implementative and a mathematical point 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 offer the possibility to simulate in an easier way *destructible* objects (see Fig. 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 shall 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 §2).

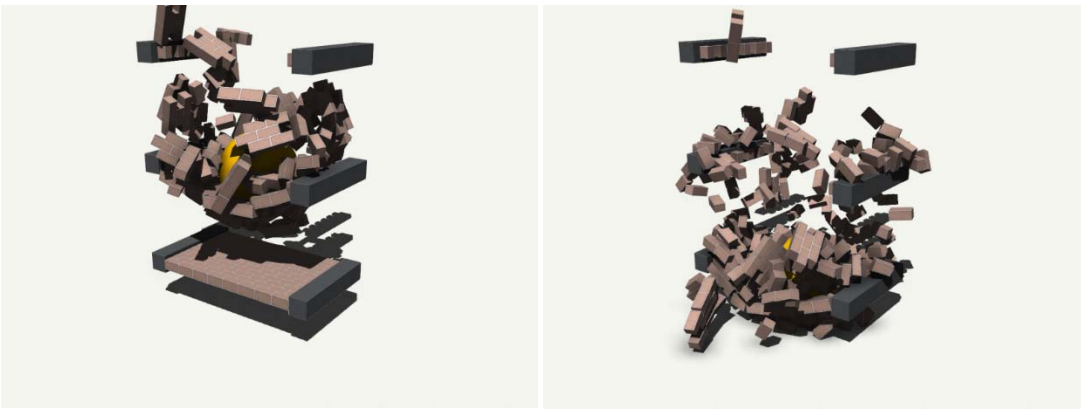

Fig. 2 – Example of destructible objects: a series of walls crumbling

The approach we propose provides two types of joints among particles: *elastic* and *semi-rigid* joints. The elastic joint is a classical spring joint with a sprig able to reach a good level of stiffness, thanks to an opportune integration method and a high refresh rate. The semi-rigid connection is assured by a maintaining constant the distance among the particles in the system and by adopting the Verlet approach to integration [31], [34], [51]. 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:

- **Atlax**: 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, 3D models, etc. Particular emphasis has been put on the design of the timing system, in order to guarantee the possibility to decouple rendering refresh rate from physics processing rate (that can rise up to 10KHz). 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 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.
- **Sulfur**: 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.

- **SulfurChamber**: a sandbox application, aimed at providing a Graphical User Interface (GUI) for Sulfur, where it is possible to create and to modify physic entities.
- **Escape from Quaoar** [20]: a video game based on mechanics aimed at exploiting the peculiarities of Sulfur.

The remaining 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 the subsequent Section 3 we will describe how the Sulfur game engine has been designed, while in the following 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 sendbox 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, 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. 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. [16], [17], [18], [25], [32], [35], etc.). Also, we want to point out that is out of the main scope of the present work to simulate the dynamics of fluids, since it would imply to address 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, for example, 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 to remember at least *Smoothed Particle Hydrodynamics* (SPH) and *Vortex Particle methods* (see e.g. [11]). In particular, SPH has been created to simulate nonaxisymmetric phenomena in astrophysics, and it was proposed for the first time by Monaghan [37]. Since its creation, SPH has evolved into other fields, among which fluid simulation (see e.g. [28], [45]) and solid mechanics (see e.g. [6], [29], [43], etc.). It has also been improved in order to support interactive simulations [38], 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 lays 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 easier to handle collisions with, e.g., 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 (§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 among *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-drive 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. 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 remaining of the engine.

The adoption of a physics engine may have some relevant implications, of which one should be well aware 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* [25]. As a matter of facts, 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 a pre-defined 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: 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 methods 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, for each of them, its main features. Numerical methods are all based on the 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 which 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, 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 NSV - *Newton–Størmer–Verlet* method) is another first order method, quick and simple, but – due to the fact the 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 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, soft-bodies can 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 constraint we have set, even the most renowned applications (such as Unity3D), as Tab 1 sums up (the requirements listed above have been put in columns). Here below we list their major characteristics.

Box2D [7] is Free to use, tailored for 2D, but lacking soft-body simulation.

Bullet [9] 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 n.1: efficient and native support to 2D soft-body simulation.

Cryengine 3 / BeamNG [12] 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 over-dimensioned and cumbersome to use for a 2D game.

DMM – Digital Molecular Matter [14] is different from the previous engines, because it is based on a simulation method called Finite Element Analysis [1], [5], [44], [54], which is far more accurate that 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 [27] 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 requirement 1 (there is a clear separation between rigid and soft bodies representation), nor requirement 2 (it can be applied to 3D environments only).

Newton Game Dynamics [39] is another open source library, not supporting soft bodies. A bit outdated also.

ODE – Open Dynamics Engine [40] is free to use, but it is outdated and only supports rigid bodies.

PhysX [42] 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 doesn't seem to cover 2D simulations. Moreover, the product is free only when used to develop on Windows operating systems.

UDK – Unreal Development Kit [48] 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 [49] is one of the most popular game development frameworks currently around. It supports every platform and provides a cheap license for indies. It provides a basic physics module that does not include any kind of soft-body simulation.

	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
DMM - Digital Molecular Matter	No	Yes	Yes	No
Havok	No	No	No	No
Newton Game Dynamics	Yes	No	No	Yes
ODE - Open Dynamics Engine	Yes	No	No	Yes
PhysX	No	Upcoming	Upcoming	Only Windows
UDK - Unreal Development Kit	No	Yes	Yes	Partial
Unity3D	Yes	No	No	Yes

Table 1 – Comparison of the more diffused game engines, based on the requirements we have defined

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 non-convex objects are involved – [36], 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 other 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 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 the Hooke's law corrected with the damping. The resulting equation is:

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

Where:

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

$F\big(t, \dot{x}(t)\big) = -b\dot{x}(t)$ 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 mitigating oscillations intensity and duration. Since it is our aim to simulate also rigid bodies using particle systems, it is of 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 refresh rate on a linear configuration composed by a system of 40 particles, that simulates a rope (dots in Fig. 3), connected by elastic joints (dashes in Fig. 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 to the length of the joint). When the simulation is started, both gravity and air friction start to affect the system (Fig. 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 (Fig. 5). Tab. 2 summarizes the maximum values for the elastic constant when the refresh rate is set to 500 Hz, while Tab. 3 summarizes the same values when the damping constant is set to 0.2. To notice that 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.

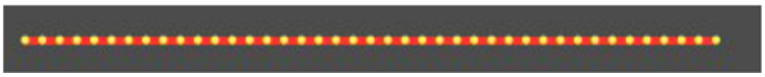

Figure 3 – The rope composed by 40 particles used to tune damping and refresh rate in its starting state.

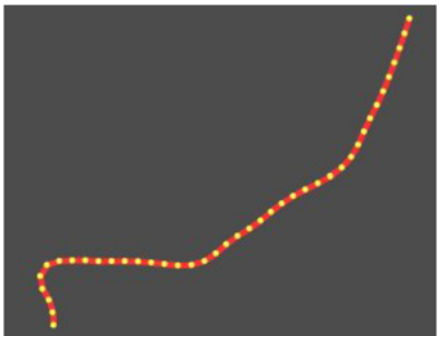

Figure 4 – The rope oscillates correctly under the effects of friction and gravity.

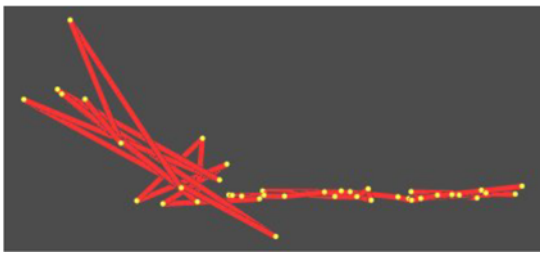

Figure 5 – The rope explodes because the system has become instable.

	DAMPING				
	0.0	***0.1***	***0.2***	***03***	***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 2 – Maximum values of the elastic constant with variable damping and fixed refresh rate (500 Hz)

	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

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

From Tab. 2 it is possible to notice that the accuracy of the integrator does not implies 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 refresh rate is taken into consideration (Tab. 3), it becomes clear that semi-implicit integrators behave by far better than explicit integrators: in the first case it is possible even to rise 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 refresh rate at 500 Hz, 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 e particles system, it 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 Fig. 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 the deformable square on the right has – correctly – bounced away after the collision.

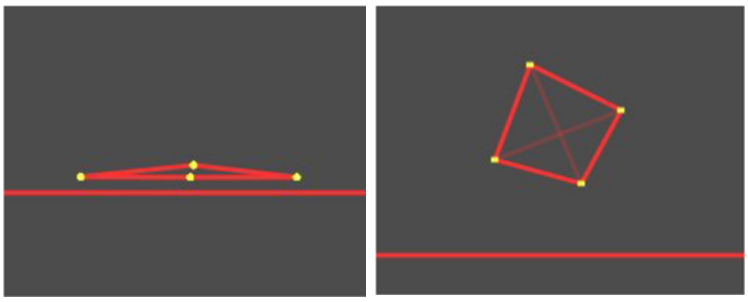

Figure 6 – A deformable soft square - with and without internal joint - collides with the floor

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 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 the 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 to highly inaccurate results.

A valid choice for an alternative integrator is the *Verlet* method [34], [51], a symplectic integrator of the second order, 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 the Taylor's theorem, applied two times: the first time forward (in time), and then backward (in time), as described by the following formulae:

$$x(t + \Delta t) = x(t) + \dot{x}(t)\Delta t + \frac{1}{2}\ddot{x}(t)\Delta t^2 + \frac{1}{6}x^{(3)}(t)\Delta t^3 + O(\Delta t^4) \qquad (2)$$

$$x(t - \Delta t) = x(t) - \dot{x}(t)\Delta t + \frac{1}{2}\ddot{x}(t)\Delta t^2 - \frac{1}{6}x^{(3)}(t)\Delta t^3 + O(\Delta t^4) \qquad (3)$$

Hence:

$$x(t + \Delta t) = 2x(t) - x(t - \Delta t) + \ddot{x}(t)\Delta t^2 + O(\Delta t^4) \qquad (4)$$

In the equation (4) velocity does not appear explicitly, if needed, it can be derived from the difference between the starting and final position 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* or *Leapfrog* integrator 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 creating 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 able to track even the smallest possible aggregate: two particles connected by a joint. 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 *BVH – Bounding Volume Hierarchy*, *SAP – Sweep And Prune*, *SAT – Separating Axis Theorem*, *GJK – Gilbert-Johnson-Keerthi distance algorithm* (see e.g. [2], [10], [19], [22], [24], [26], [47], [50]) is not a viable way. In the same vein, the *Bentely-Ottmann* algorithm, also called *Sweep-Plane* algorithm [4], which is aimed at finding intersections between segments on a plane, is not adaptable to our specific case. In particular, it considers only one-dimension segments, that cannot cope with some extreme case, such those depicted in Fig. 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.

Figure 7 – Two segments that should collide with the floor

Since no standard method to manage collision detection seems to be able to support our approach to rigid body representation, we have developed an original approach, which preserves the two “canonical” phase: *broad* and *narrow* collision detection.

The first – *broad* – phase of collision detection is aimed at detecting the objects potentially superimposing. To avoid the trap of one-dimension segments, we include each segment in a “capsule” (see. Fig. 8 - left), which simulate thickness. We then subdivide the plane with a uniform grid and keep track of the cells crossed by each segments in each moment (see Fig. 8 - right). 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 [8], which supplies good performances, with an execution time which depends linearly from the number of capsules present on the grid.

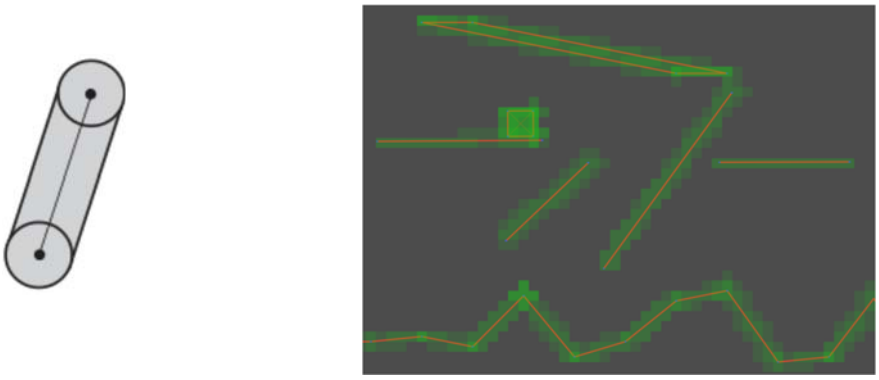

Figure 8 – A single capsule surrounding a segment (left) and the representation of capsules on the grid

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 friction. 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 from the operating system in order to provide the maximum possible portability. All the code we have developed has 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* [53], a cross-platform GUI (Graphical User Interface) library, and *Simple DirectMedia Layer* [46] a cross-platform library designed to provide low level access to audio, keyboard, mouse, joystick, and graphics hardware via OpenGL [41] and DirectX [15]. 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 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 shares some classes (notably to execute calculus and to dynamically manage particles and joints). It implements the approach we have summarized in the previous §3.

Basing 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 version. Its main purpose is to allow experimenting different configurations of particles with varying elasticity, gravity, sliding friction and air friction (Fig. 9, left). In particular, it is possible to create objects both starting from a pre-defined set (Fig. 9, center) or by designing them from scratch by combining sets of particles and joints (Fig. 9, right). 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, Intel Core –i7 930 with Linux-Ubuntu. The simulation has been run by simulating an increasing number of identical squared boxes (each of which composed by four particles, four main rigid joints and two internal structural joints). Couples of boxes were dropping from above continuously, hence no box was in quiet (this would have the consequence to exclude the box from the physics simulation). The values of friction, mass and gravity were set to the same values for the whole simulation. Tab. 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 is around 60 or less, depending on the combination 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.

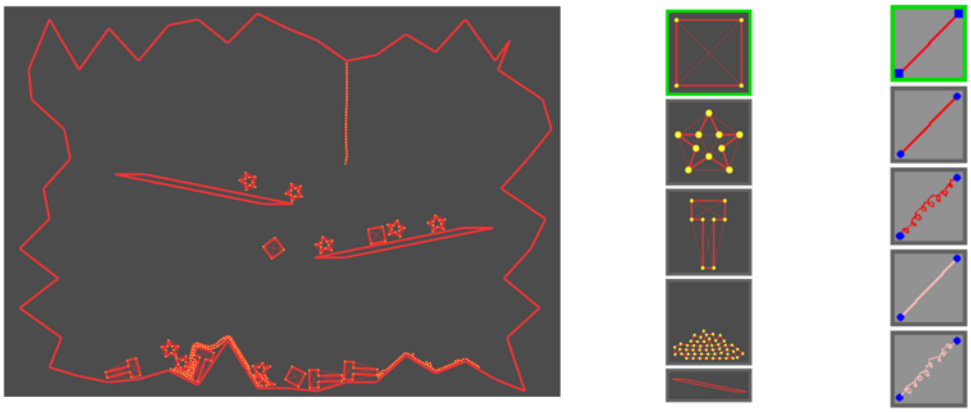

Figure 9 – Soft and rigid bodies in SulfurChambre

	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%

Table 4 – Performance of Sulfur with different hw-sw configurations.

Escape from Quaoar [20] 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 Fig. 10, right), that 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 Fig. 11, left). The overall behavior of the game is very satisfying, and the game has participated to the 13th Independent Game Festival (IGF) [30].

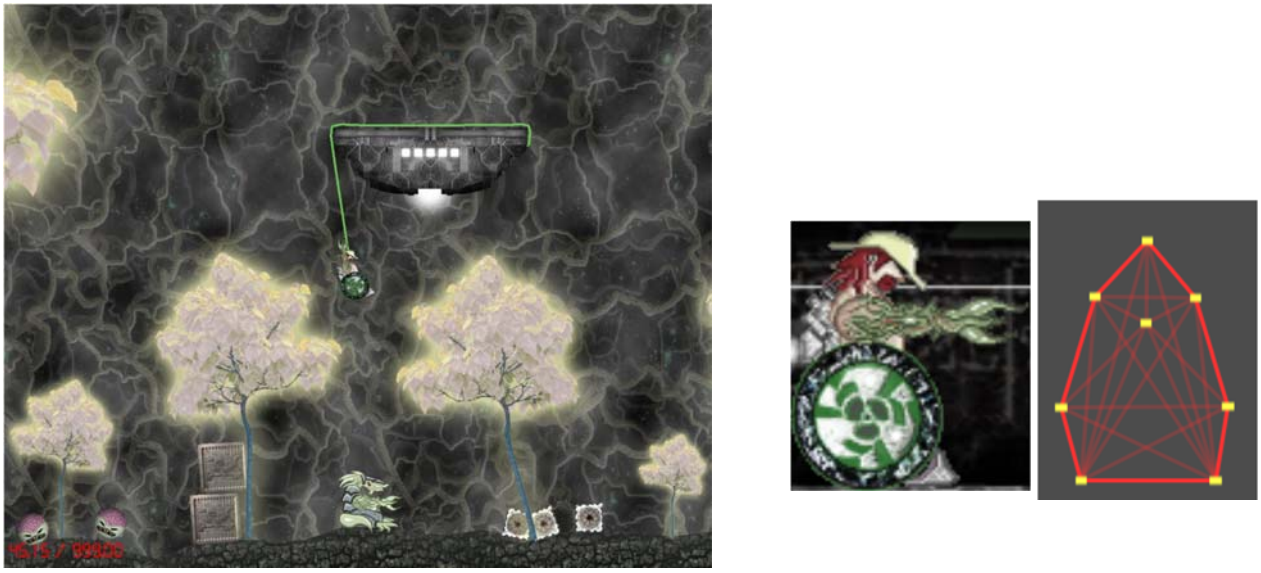

Figure 10 – Screenshots from Escape from Quaoar: example of a level (on the left) and the main character texture and underpinning particle system (on the right)

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

5. Conclusion and future work

The main goals of our work has been to design and create a physics simulator for video games able to unify rigid and soft bodies simulation, endowed with an 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 the interesting side effect to avoid messing up with angles and rotations. In the same vein, we have obtained a sound approach to collision management by coupling a static grid with Bresentham 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:

- Increasing accuracy of simulation for friction;
- 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);

- Optimizing calculation-intensive code section, by moving the workload to the GPU;
- Introducing angular joints, to decrease the number of particles used to represent a single object;
- Introducing a more sophisticated simulation of fractures (at the moment fractures are obtained by removing some joints);
- Extending Sulfur to 3D simulation;
- Extending Sulfur to simulate fluids motion;
- Finally, yet importantly, we are planning to reduce the refresh rate (whose value at present is around 500Hz), 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.

References

[1] Babuška, I., Banerjee, U., Osborn, J.E. 2004. Generalized Finite Element Methods: Main Ideas, Results, and Perspective. *International Journal of Computational Methods* 1 (1): 67–103.

[2] Baraff, D. 1992. *Dynamic Simulation of Non-Penetrating Rigid Bodies*, (Ph. D thesis), Computer Science Department, Cornell University, pp. 52–56.

[3] Bartle, R. 2003. *Designing virtual worlds*. New Riders Pub.

[4] Bartuschka, U., Mehlhorn, K. and Näher, S. 1997. *A robust and efficient implementation of a sweep line algorithm for the straight-line segment intersection problem*. Orlando, S., Proc. Worksh. Algorithm Engineering.

[5] Bathe, K.J. 2006. *Finite Element Procedures*. Cambridge, MA: Klaus-Jürgen Bathe.

[6] Bonet, J. and Kulasegaram, S. 2000. Correction and stabilization of smooth particle hydrodynamics methods with applications in metal forming simulations. *Int. J. Numer. Methods Eng*. 47: 1189–1214.

[7] Box2D http://www.box2d.org last visited Jan. 2014.

[8] Bresenham, J. E. 1965. Algorithm for computer control of a digital plotter. *IBM Systems Journal* 4 (1): 25–30.

[9] Bullet http://bulletphysics.org/wordpress last visited Jan. 2014.

[10] Cohen, J.D., Lin, M.C., Manocha, Ponamgi, M.K. 1995. *I–COLLIDE: an Interactive and Exact Collision Detection System for Large Scale Environments*. Proceedings of the 1995 Symposium on Interactive 3D Graphics (Monterey, CA), pp. 189–196.Christer E. 2005. Real-time collision detection, Morgan Kaufmann series in interactive 3D technology, Amsterdam: Elsevier, pp. 329–338.

[11] Cottet, G.H. and Koumoutsakos, P.D. 2000. *Vortex Methods. Theory and Practice*. Cambridge University Press, 2000. isbn: 0-521-62186-0.

[12] Cryengine http://www.crytek.com/cryengine last visited Jan. 2014.

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

[14] Digital Molecular Matter – Pixelux http://www.pixelux.com/ last visited Jan. 2014.

[15] Direct3D http://www.microsoft.com/en-us/download/details.aspx?id=23803 last visited Jan. 2014.

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

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

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

[19] Ericson, C. 2006. *Real-Time Collision Detection.* Morgan Kaufmann.

[20] Escape from Quaoar. http://www.escapefromquaoar.com last visited Jan. 2014

[21] Fullerton, T. 2008. *Game Design Workshop: a playcentric approach to creating innovative games*. Elsevier.

[22] Gilbert, E., Johnson D.W. and Keerthi S. 1998. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal of Robotics and Automation*, Vol.4, No.2, April 1998.

[23] Gran Turismo http://www.gran-turismo.com/ last visited Jan. 2014

[24] Golshtein, E. G.; Tretyakov, N.V.; translated by Tretyakov, N.V. 1996. *Modified Lagrangians and monotone maps in optimization*. New York: Wiley.

[25] Gregory J. 2009. *Game Engine Architecture.* Taylor and Francis.

[26] Günther, J., Popov, S., Seidel H-P. and Slusallek, P. 2007. *Realtime Ray Tracing on GPU with BVH-based Packet Traversal*. Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007, pag 113-118.

[27] Havok http://www.havok.com/ last visited Jan. 2014.

[28] Hérault, A., Bilotta, G., Rustico, E., Vicari, A. and Del Negro, C. 2011. Numerical Simulation of lava flow using a GPU SPH model. *Annals of Geophysics*, Vol. 54, n. 5.
[29] Herreros, M.I., and Mabssout, M. 2011. A two-steps time discretization scheme using the SPH method for shock wave propagation. *Comput. Methods Appl. Mech*. Engrg. 200: 1833–1845.
[30] IGF – Independent Games Festival http://www.igf.com/ last visited Jan. 2014.
[31] Jakobsen, T. 2003. *Advanced Character Physics.* http://www.pagines.ma1.upc.edu/~susin/files/AdvancedCharacterPhysics.pdf last visited Jan. 2014.
[32] Kodicek, D. 2005. *Mathematics and Physics for Game Programmers. Hingham,* MA: Charles River Media.
[33] Koster, R. 2005. *A Theory Of Fun For Game Design*. Paraglyph Press.
[34] Mcguire M., and Jenkins, O.C. 2008. *Creating Games: Mechanics, Content, and Technology*. A K Peters Ltd.
[35] Millington, I. 2007. *Game Physics Engine Development.* San Francisco, CA: Morgan Kaufmann.
[36] Millington, I., and Funge, J. 2009. *Artificial Intelligence for Games*, Second Edition. Morgan Kaufmann.
[37] Monaghan, J.J. 1977. Smoothed particle hydrodynamics. *Annual Review of Astronomy and Astrophysics*, 30(1):543-574.
[38] Muller, M., Charypar, D., and Gross, M. 2003. *Particle-based fluid simulation for interactive applications*. In Proceeding of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, pages 154-159. Eurographics Association.
[39] Newton Game Dynamics http://newtondynamics.com/forum/newton.php last visited Jan. 2014.
[40] Open Dynamics Engine - ODE http://www.ode.org last visited Jan. 2014.
[41] OpenGL http://www.opengl.org/ last visited Jan. 2014.
[42] PhysX http://developer.nvidia.com/object/physx.html last visited Jan. 2014.
[43] Rabczuk, T., Eibl, J. and Stempniewski, L. 2003. Simulation of high velocity concrete fragmentation using SPH/MLSPH. *Int. J. Numer. Methods Eng*.56: 1421–1444.
[44] Reddy, J.N. 2005. *An Introduction to the Finite Element Method* (Third ed.). McGraw-Hill.
[45] Rustico, E., Bilotta, G., Hérault, A., Del Negro, C. and Gallo, G. 2012. *Smoothed Particle Hydrodynamics simulations on multi-GPU systems*. 20th Euromicro International Conference on Parallel, Distributed and Network-Based Computing, Special Session on GPU Computing and Hybrid Computing, 2012, Garching/Munich (DE).
[46] SDL, 2013. http://www.libsdl.org/ last visited Jan. 2014.
[47] Shimizu, K., Ishizuka, Y. and Bard, J.F. 1997. *Nondifferentiable and two-level mathematical programming*. Boston: Kluwer Academic Publishers.
[48] Unreal Development Kit – UDK http://www.unrealengine.com/en/udk/ last visited Jan. 2014.
[49] Unity3D http://unity3d.com/ last visited Jan. 2014.
[50] van den Bergen, G. 2003. *Collision Detection in Interactive 3D Environments.* San Francisco, CA: Morgan Kaufmann.
[51] Verlet, L. 1967. *Computer experiments on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules,* Phys. Rev., 159, 98-103.
[52] Witkin, A. and Baraff, D. 1997. *Physically Based Modeling: Principles and Practice*. Proc. SIGGRAPH '97. Los Angeles: Association for Computing Machinery Special Interest Group on Graphics. 1997.
[53] wxWidgets http://www.wxwidgets.org/ last visited Jan. 2014.
[54] Zienkiewicz, O.C., Taylor, R.L., and Zhu, J.Z., 2005. *The Finite Element Method: Its Basis and Fundamentals* (Sixth ed.). Butterworth-Heinemann.
[55] Zimmerman, E., and Salen, K. 2004. *Rules of play: game design fundamentals*. The MIT Press.