Evaluation of Numerical Integration Methods for Simulation of Simple 3D Rigid Body Motion Within a Game Environment

Vladeta Stojanovic

University of Abertay Dundee
Institute of Arts, Media and Computer Games
May 2012

University of Abertay Dundee

Permission to copy

Author: Vladeta Stojanovic

Title: *Evaluation of Numerical Integration Methods for Simulation of Simple 3D Rigid Body Motion Within a Game Environment*

Degree: BSc (Hons) Computer Games Technology

Year: 4

(i) I certify that the above mentioned project is my original work
(ii) I agree that this dissertation may be reproduced, stored or transmitted, in any form and by any means without the written consent of the undersigned.

Signature

Date **03/05/2012**

Table of Contents

List of Figures

Abstract

The research topic covered in this dissertation addresses the importance of evaluating and selecting an appropriate numerical integration method for calculation of 3D rigid body motion within a 3D scene. The selected integration methods that are evaluated are the implicit Euler and Runge-Kutta Order Four (RK4) integration methods. The main problem often encountered when implementing a new physics system for an interactive 3D application is selecting an appropriate numerical integration method that provides reasonable numerical accuracy, along with reasonable processing and resource usage on the target platform.

The methodology used to obtain the results for the evaluation of the implicit Euler and RK4 integration methods is implemented in terms of a simple 3D interactive application developed with Microsoft XNA 4.0. This application allows the user to examine the characteristics of 3D rigid body motion using the two selected numerical integration methods. Based on this analysis, the results concerning the efficiency, performance impact and aesthetic quality of the selected integration methods can be assessed. The research undertaken also shows that implementing the empirical model of Newtonian dynamics for 3D rigid body motion is not a trivial matter, as various discrete mathematical modelling techniques need to be used in order to achieve the desired result.

The results obtained show that for a simple interactive 3D application, the use of implicit Euler integration for 3D rigid body motion suffices, while the use of RK4 numerical integration does not provide any significant improvements to the aesthetic qualities of the simulation.

1. Introduction

One of the most important aspects of an interactive 3D application is being able to allow the user to interact with various entities in a given 3D scene. Often, this is accomplished using various physics systems. These systems can be as simple as one dimensional linear velocity calculations to hardware accelerated soft body particle physics simulations. The middle ground most real time interactive applications settle on is using 3D rigid body systems to approximate the positional and rotational properties of 3D objects in a 3D scene where such objects are intended to behave "realistically " under various simulations of physical phenomena. A given 3D rigid body system will typically try to approximate the physical phenomena of an impulsive force striking a rigid body at a point from its centre of mass, making it move and spin accordingly in the direction of the impulse. The amount of movement and spinning the object does depends on various properties of the object in question, such as its size, mass and shape, as well as other simulated factors such as gravity.

Since a given 3D object in a scene can take the form of any shape imaginable, approximating its physical properties and behaviour can become very complex. The idea behind rigid bodies is to treat a given object as an enclosed volume. Within this enclosed volume, every point in the volume has a constant mass, size and position. Therefore a rigid body does not deform and its mass is said to be distributed evenly. This allows for the approximation of the rigid bodies transitional and rotational properties to be updated dynamically for all objects enclosed within a given rigid body volume. For efficiency the volume takes the shape of the simplest approximation of the bounding values of a given 3D shape that it encloses. As only the given volume is treated as a rigid body within a system, the physical properties such as translation and rotation of a given 3D object can be computed efficiently, regardless of how complex the actual mesh data of the 3D object is. This makes rigid body systems ideal for real time use.

The approximated transitional and rotational values for a given rigid body volume depend on what is called an inertia tensor. An inertia tensor can be thought of as a property matrix that defines how a given object should be translated in 3D space based on its volume properties and the resulting angular and linear velocities applied to it from an outside impulsive force. In a 3D rigid body system, all of these computations are approximated in a discrete differential manner with respect to time. The approximated values are summations of the angular and linear velocity values updated every frame. How these summations are computed determines the overall accuracy, stability, efficiency and aesthetic qualities of a given 3D rigid body system.

These summations are products of numerical integration methods. Such methods exist in order to provide numerical approximations for equations where the given answer or desired value cannot be obtained analytically (at least not very easily). In classical mechanics, much of the problems involving dynamics concerning the computation of general motion of objects in 3D space rely on discretised versions of Newton's laws of motion, referred to as the Newton-Euler equations of motion. In a 3D rigid body system, these equations can be implemented and approximated using various numerical integration methods. Such integration methods can be used to approximate the transitional and rotational values of a 3D rigid body, to a given degree of accuracy. Additionally, depending on the numerical integration method used, some methods tend to behave better than others in terms of their potential to converge to a correct approximated value of a desired degree of accuracy. The terms *implicit* and *explicit* are used to refer to numerical integration methods where the given method is said to be implicit if it can converge to an approximation of reasonable accuracy and explicit if after a given period of the approximated function or equation, the approximated values diverge.

The research presented in this dissertation aims to answer the question of: *what is the most efficient numerical integration method to use within a simple 3D rigid body system?* The dissertation aims to answer which of the selected numerical integration methods provide the best efficiency for use in a 3D rigid body system. The efficiency is measured in terms of the computational processing performance of the selected numerical integration systems. The two selected numerical integration algorithms that will be evaluated are the implicit Euler method and the Runge-Kutta Order Four Integration method (commonly abbreviated as RK4). Both the RK4 and the implicit Euler methods are used to replace the traditional analytic approach of solving 3D rigid body dynamics problems. The results used to support the findings are obtained through means of code profiling and performance testing the resulting demo application.

2. Literature Review

A rigid body is a non-naturally occurring volume of particles where each particle within the volume retains its position during the lifetime of the rigid body [DeLoura et al 2000]. Each of the particles in the given volume are also position-relative to the centre of mass of the given 3D rigid body. Therefore mass within a rigid body is evenly distributed. These properties allow for simplified approximations of motion of objects enclosed within a given volume, regardless of the enclosed object mesh complexity. The enclosing volume of a rigid body can be thought of as an inertial frame used to calculate the motion of an object relative to any given point within the mass volume. For most 3D rigid bodies, simple volume shapes can be used to approximate the closest bounding volume shape of that body. For example, a wheel of a 3D car model can be approximated using a cylinder whose radius and depth enclose the bounds of the 3D wheel mesh. Likewise, the rigid body volume of the body of the said 3D car model can be approximated using solid cube of a given width, height and depth that encloses the 3D mesh of the car body. For most 3D rigid bodies, a simple solid cube volume is a popular choice to use [DeLoura et al 2000].
A 3D rigid body being simulated is said to have three different motion states: stationary, non-rotational transformation and rotational transformation. Moving a 3D rigid body requires updating its acceleration and linear velocity, and rotating it quires updating its angular velocity. A stationary 3D rigid body is really of no use, as for collision checking purposes, bounding volume intersection testing is sufficient for most simple simulation scenarios [Eberly 2010].

The inertial frame of reference of a given 3D rigid body is represented visually by a volume, but the physical properties of the given volume need to be approximated in order to determine the transitional and rotational properties of the rigid body. This is achieved with the use of an inertia tensor. An inertia tensor can be thought of as a property matrix where the main diagonal elements are called the moments of inertia and the off-diagonal elements are called the products of inertia. These element properties are used to determine how much the given rigid body rotates around a given axes, relative to an angular force applied to it [Lengyel 2011], [DeLoura et al 2000]. An inertia tensor for a 3D solid cube volume can be defined in terms of its bounding width, height and depth as [Lengyel 2011], [DeLoura et al 2000] (For a default 3x3 symetrical inertia tenosor model, please see Appendix A):

$$I_{box} = \begin{bmatrix} \frac{1}{12} m(h^2 + d^2) & 0 & 0 \\ 0 & \frac{1}{12} m(w^2 + d^2) & 0 \\ 0 & 0 & \frac{1}{12} m(w^2 + d^2) \end{bmatrix}$$

Once the inertia tensor for a given 3D rigid body has been set up, the angular and linear motion values for the given 3D rigid body can be obtained by integrating each of the elements of the inertia tensor, with respect to the dimensions of the given rigid body volume and the vector operations performed using the vector values of the impulsive force that sets the given rigid body into motion. This is where the idea of numerical integration comes into play. Since the equations of motion that are applied to the given 3D rigid body and their derived values are obtained in an approximated form, these values need to be summed over a period of time. The summation method needs to approximate a given value for a certain moment in time. Depending on the numerical integration system used, these approximated values can be obtained for the current time value in the simulation and future time values [Verth et al 2008] (see Appendix A for more information).

As mentioned in the introduction, numerical integration methods are said to be *explicit* or *implicit*. The two selected numerical integration methods that were evaluated were the implicit Euler and the Runge-Kutta Order Four (RK4) numerical integration methods. Traditionally, numerical integration using the standard Euler integration method is an explicit method, as an exponential error is introduced to the approximation of a given function value [Kreyszig 2006], [Moler 2004].

[DeLoura et al 2000] describes that when such an explicit approximation method is applied to a rigid body system, it causes very significant stability errors. One way to increase stability is to use an implicit version of the Euler integration method, called the implicit Euler integration method (also known as the Backward Euler method). The implicit Euler integration method makes use of the new approximated values of the given function, rather than the previous values. It is defined as [Kreyszig 2006] [Moler 2004]:

$$y_{n+1} \approx y_n + hf(x_{n+1}, y_{n+1})$$

The RK4 numerical integration method is more accurate than the implicit Euler numerical integration method, and always converges to a close approximation of desired accuracy [Moler 2004]. The only disadvantage of using the RK4 method over the implicit Euler method is that it's computationally more expensive. The RK4 method is defined as [Kreyszig 2006] [Moler 2004][Press et al 1992] (for the complete model of the RK4 method, please see Appendix A):

$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$
$$x_{n+1} = x_n + h$$

The reason why the RK4 and implicit Euler methods were chosen is because they are both of the most popular numerical integration methods used in the simulation of 3D rigid body dynamics [Bourg 2002][Eberly 2010]. While Verlet integration is also another popular numerical integration method, it is best suited for more complex real time 3D dynamics problems, such as the simulation of soft body particles [Verth et al 2008]. With these systems in place, an investigation can be carried out see which of the two numerical integration methods are the most efficient.

It should be noted that the main focus of this dissertation is on numerical integration of rigid body motion. Therefore, focus on other aspects of physical rigid body simulation such as collision detection and response were limited to only the most basic implementations (as the field of real time 3D collision detection and response warrants its own research dissertation). On further note, it is important to mention that implementing a rigid body system based on the pre-set laws of motion as defined in classical mechanics textbooks is not a very easy task. Often, various hacks, tricks and optimizations need to be implemented in order for a rigid body to behave correctly in a simulation [Bourg 2002].

The result of this is that an implemented rigid body system that follows the correct always of physics may not always behave in an aesthetically pleasing manner (this is very important for interactive entrainment applications such as fast paced first person shooters or driving games – see Discussion section). For such applications, the most important thing is to make the physics look "good" while maintaining a given degree of realism [Bourg 2002]. As the main focus of this dissertation is rigid body systems for interactive applications such as games, the implemented rigid body system follows an *empirical model* of simulation. Such models of simulation do not guarantee scientific accuracy for the results obtained from such a system, but they do guarantee an aesthetic accuracy [Bourg 2002]. This is the other part of the research that this dissertation addresses along with answering the main question, and that is deciding which one of the selected numerical integration methods are not only optimal in terms of performance, but also how much they complement the aesthetics of the given empirical simulation within an interactive application.

3. Methodology

The development of the demo application accompanying the dissertation research was done using XNA 4.0 and programmed in C#. The reason why XNA 4.0 was chosen is because XNA allows for rapid application development of interactive 3D applications without need to write lots of "boiler plate" code (such as window creation, rendering context and input routines). This allowed for the development of the demo application to focus on creating a simple 3D rigid body simulation. The developed application allows the user to interact with a 3D object in the scene (the 3D object in question being a simple 3D model of a uniform cube), using rigid body based physics simulation. The user can control the motion of the 3D box model by applying impulsive forces to each of the six sides of the cube, along each of the world coordinate axes. Gravity is also simulated in the application as a constant vector impulse force where the y-coordinate element has a constant value of -9.8 m/s. The rotational motion of the box model is based on the implemented box inertia tensor (contained within the box entity class, see Appendix B for program implementation details), while transitional motion is based on the mass of the object and the size and direction of the impulsive force applied to it.

Another important reason for using XNA was that XNA comes with a very good set of 3D mathematics routines within its built-in auxiliary classes and components (such as a 3D vector class, various linear interpolation functions, amongst other functionality). It also has its own bounding box collision class and accompanying collision checking/intersection routines. However, XNA does have its shortcomings: The main problem was the high level of abstracted functionality of the XNA helper libraries and routines, particularly the bounding box collision detection routines. The default bounding box class featured in XNA makes use of axis-aligned bounding boxes (AABB) [Eberly 2010]. This means that the every time a 3D object that makes use of an AABB changes its position and/or orientation, the bounding box alignment becomes distorted. This makes the use of AABB's for dynamic object not very ideal. Instead, for dynamic objects, orientated bounding boxes need to be implemented (OBB's). OBB's are much like AABB's, except that their coordinates are aligned to the world transformation matrix, rather than the objects local transformation matrix. This means that their position, size and rotation properties are updated along with the 3D mesh they are assigned to [Eberly 2010].

Unfortunately, XNA does not feature default support for OBB's, and therefore a solution had to be developed for updating the present AABB properties of the box object dynamically. This was accomplished by assigning new bounding box dimension values every time the box object in the scene is transformed to a new position.

However, this does not account for the orientation of the box, and since assigning a new bounding box to fit the rotated axes of the box model every frame is considered to be computationally costly, a new method was devised: The method involves resizing the coordinates of the bounding box every frame based on the position and the size of the box (though the size does not change), and multiplying these coordinates by the world matrix, so that the bounding box encloses the box model not matter how the box is rotated. It should be noted that the bounding box itself is not rotated, but is simply rescaled to fit the size of the rotated box model it encloses.

The above process provides satisfactory collision detection results. The reason why the collision results are only satisfactory is because collision checking happens only between the bounding box faces enclosing the box model (regardless of 3D meshes rotation within the bounding box), and the top face of the ground plane model bounding box. This means that if the box mesh is rotated within the bounding box, the edges of the box mesh may end up going through the ground plane before the collision intersection result is returned and the box object is snapped back outside of the intersection (see Figure 1):

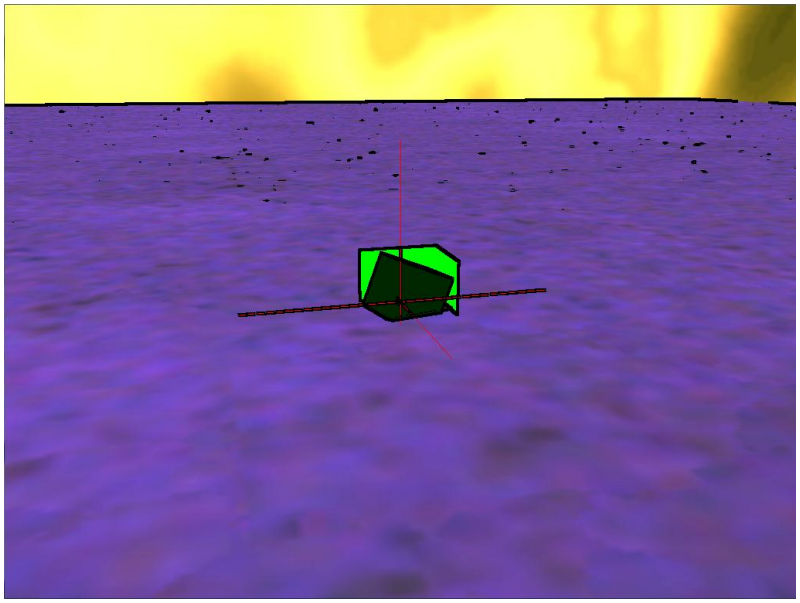

Figure 1: The box model intersecting with the ground plane.

This was the main drawback of using XNA to handle collision detection. Since the implementation of fully precise collision detection was not a priority task for the development of the demo application, the current collision detection system is sufficient for demonstrative purposes.
The next important part of the dissertation application to discuss is the implementation of the numerical integration algorithms. The implemented implicit Euler and RK4 numerical integration algorithms (see Appendix B for full code listing of implementation), numerically integrate the computed linear and angular velocities of the box object every frame during the run-time of the application. Based on these values, the transformational and rotational properties of the box object in the scene are updated accordingly. The implemented numerical integration algorithms both make use of the current number of elapsed milliseconds as the time differential variable used to sum to the velocity components. One interesting thing to observe about both methods is that they are both fairly similar in their structure and operation. The main differences between the implementations are the four extra computational steps implemented by the RK4 method. In essence, the implicit Euler method can be thought of as a first-order Runge-Kutta numerical integration method. Both methods focus on stabilizing the Taylor series expansions of given order so that the approximation error based on the sampled stepping size for a given function is within reasonable limits [Bourg 2002][Eberly 2010].

The visual simulation of the state of equilibrium is actually a rather difficult physical phenomenon to simulate in real time. Early 3D physics engines often used hacks to prevent objects "jittering" while in resting contact [Eberly 2010]. In more complex scenarios, this problem is solved by numerically integrating the collision detection and response values along with the equations of motion for the rigid body in question. However, for the developed demo application a simple system was used that checked to see if the bounding box is in contact with the ground plane, then if it is, a linear interpolation function is used to interpolate the current linear and angular velocities of the box object to zero over a given period of time, based on value of a constant friction coefficient variable (see Appendix B for implementation details).

In order to simulate motion for demonstration purposes, the demo application makes use of acceleration-based constrained motion, specifically impulse based motion. [Eberly 2010] describes acceleration-based motion as asset of methods used to compute motion interaction between two or more objects, where either of the objects are resting or in motion. When the demo application was first being developed, the user could control and drive a car object around the scene. A system was developed that allowed the box object in the scene to be accelerated in the opposing direction of its contact with the car object. However, because the collision system at the time was not sufficient enough to handle proper contact based collision checking and response, the car object was removed from the demo application. The box object remained, but instead of the user driving a car into the box, the user of the demo application now applies impulses along each of the local box object axes via keyboard input. This system works sufficiently, as the user is able to apply an impulse to the box object along all five of its sides (except the top face of the box object, as there is a gravity constant being applied to the box object along in the negative y-axis direction, if the box object is off the ground). However, the computation of the change of the object linear and angular velocity, based on the applied impulse, is *exclusive* to that instance. This means that if the user applies an impulse to the object that accelerates the object in a given direction, and then while the object is moving another impulse is applied, the derived results of the linear and angular velocity of the object becomes exclusive to the last applied impulse while the previous velocity values become *discontinuous* [Eberly 2010]. While this is not entirely realistic, within an empirical simulation model, this can be used in place of more complex change of velocity computation methods (see Figure 2 for an example of the implemented motion modelling solution).

The final important aspect of the dissertation demo application to talk about is how the results used for the comparison of the implicit Euler and RK4 methods were obtained. This is where the concept of code profiling is very useful. Code profiling is essential when prototyping any performance critical component of a given application and can be thought of as the process of extracting data that can be used for evaluating critical components of the application. Both of the evaluated numerical integration methods are deemed as critical components of the dissertation demo application. The two main performance areas that were evaluated for each method were the number of milliseconds taken to process each integration cycle per frame, and the overall CPU usage of the application when running either of the two integration methods. The obtainment of the results of the number of milliseconds taken to complete each integrations cycle per frame was made possible with the implementation of a standard high-resolution timer that samples the number of elapsed milliseconds as high-precision 64-bit double floating point values. This allows for the evaluation of the elapsed time results to be viewed to a very high fractional degree (see Appendix B for implementation listing and details). However, even though this profiling data was being computed, the demo application had to be able to somehow log this data. To provide a solution for this, a simple text file logging system was implemented that would log the elapsed number of milliseconds for each integration cycle per frame to a text file that could be viewed externally once the application was terminated. This data was then analysed and plotted in Microsoft Excel to provide the obtained results.

The other important profiled data that was obtained was the overall CPU usage during the run-time of the application using either of the selected numerical integration methods. Since the demo application as developed using Visual Studio 2010 Professional edition, the performance analysis and code profiling tools that are featured in the Enterprise and Ultimate editions of the IDE were not available. Therefore an open-source IDE called Sharp Develop was used, as it is a popular IDE and development environment used for the development of managed .Net applications. It also allows for profiling of stand-alone managed applications that make use of the .Net framework, such as those developed with XNA. The dissertation demo application CPU processing performance was then analysed using Sharp Develop to obtain the desired data. The profiling tools in Sharp Develop were also used to obtain the visualisation of the CPU performance via a screen-capture of the visualised real-time CPU usage graph for the dissertation demo application (see Figures 4 and 6).
Overall, both of the profiling methods combined provided satisfactory data that was used to arrive to the predicted conclusion of the dissertation research topic.

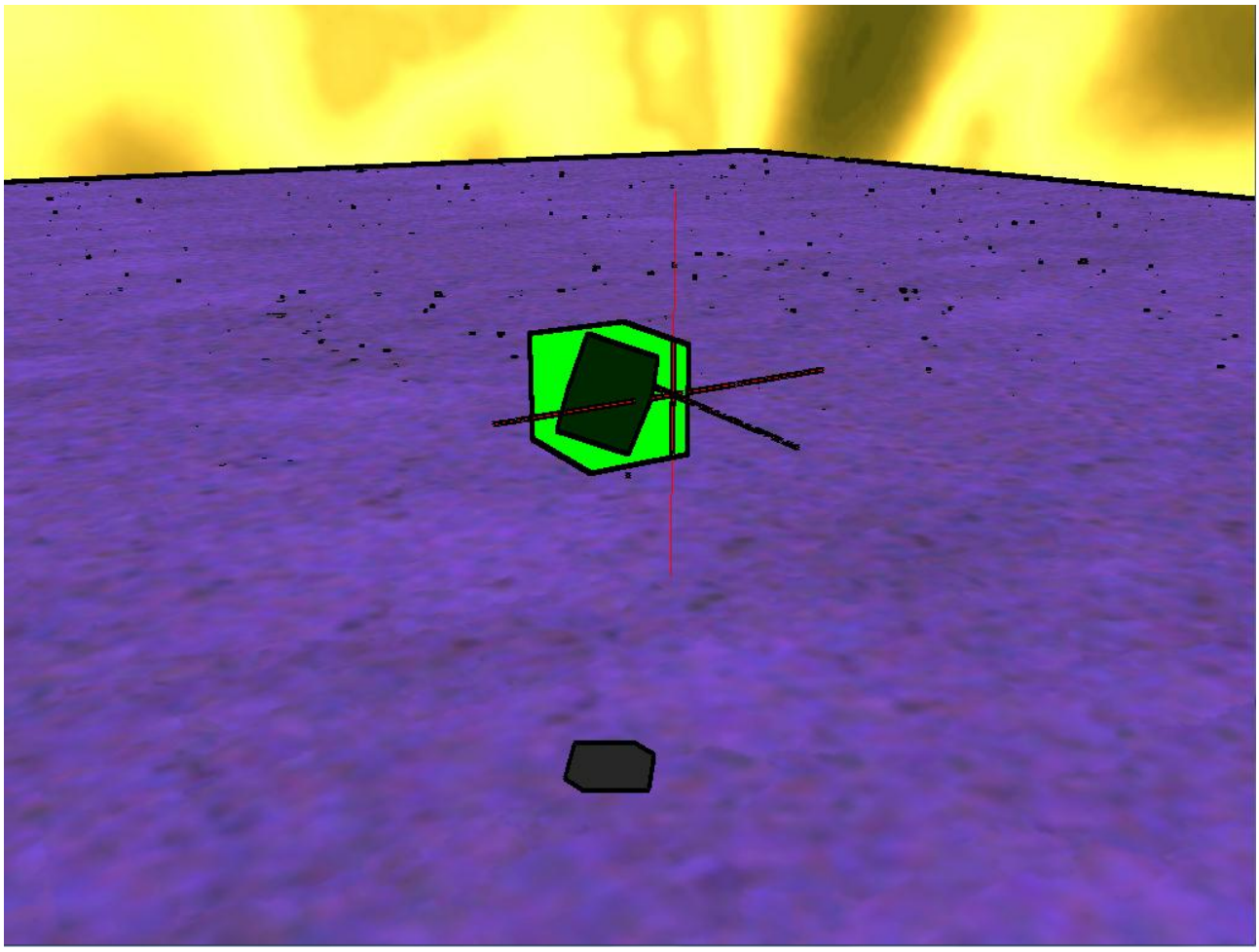

Figure 2: The box rigid body in motion. Notice that the bounding box is re-calculated every frame.

4. Results

The obtained results show that the most efficient numerical integration method to use for the simulation of simple 3D rigid body motion is the implicit Euler numerical integration method.

The stance taken on the topic of selecting the appropriate numerical integration method is: ***The aim of this dissertation was to show which method was more efficient, without causing any visible deterioration to the aesthetic quality of the simulation. Based on these criteria, the implicit Euler method is the recommended method. Implicit Euler provides satisfactory results for non-complex simulation of 3D rigid body motion dynamics. The use of RK4 numerical integration does not provide any greater aesthetic improvements to the simulation, and therefore, it makes more sense to use implicit Euler numerical integration since it requires less processing time and power.***

This answers the main research question of the topic. However, this answer is satisfactory in only the most common cases where the simulation of 3D rigid body motion is concerned. For more complex simulation, such as those that may require greater numerical precision, the RK4 numerical integration method should be used. Implicit Euler numerical integration is also best suited in the use of empirical simulation of 3D rigid body dynamics, such as those where the aesthetic result is more important than the scientific accuracy of the simulation. For example, using implicit Euler for the simulation of particles moving in an unnatural gravity setting for simulation of space flight would not be ideal, and something more complex like a higher order RK or Verlet integration methods would be used. But for purposes of entertainment, such as video games and interactive 3D applications such as training simulations, where the aesthetic quality is more important that scientific accuracy, the implicit Euler numerical integration method is sufficient for use in the simulation of 3D rigid body motion in such scenarios. It should also be noted that the use of implicit Euler numerical integration for time-stepping simulation of *simple* 3D rigid body motion is also recommended by [DeLoura et al 2000] and [Bourg 2002].
However, the stability of any numerical integration method is dependent on the chosen stepping size [Eberly 2010]. A numerical integration method is considered stable if it does not diverge too much from the initial approximated result. But how much is "too much"? The relationship of stability concerning numerical integration and the simulated physical stability of a given system can be evaluated in terms of the difference of the approximated results as the time variable of the simulation increases [Eberly 2010]. Both the implicit Euler and the RK4 numerical integration methods are considered to be stable for the simulation of "stiff ODE's" by [DeLoura et al 2000], [Bourg 2002], [Verth et al 2008] (also see Discussion section), and are considered to be stable for most other numerical analysis application as well [Kreyzig 2006], [Moler 2004],[Press et al 1992] (see Appendix for more information). The stepping size used for testing both of the numerical methods was set to a value of 0.005 (see the discussion section for the reason behind this). This ensured that both methods were being used to approximate results to a good degree of accuracy. The main testing criteria for the efficiency of the two selected numerical integration methods was the amount of time taken to complete each integration cycle during the run-time of the application, and the impact each of the selected numerical integration methods had on the processing usage of the CPU running the application***. The obtained results show that the RK4 numerical integration method takes slightly longer to compute than the implicit Euler method (a few more fractions of a millisecond), and that neither of the methods have any significant impact on the processing usage of the CPU. Please see Figures 3 – 6 for evidence of this.***
It should be noted that the aim of this dissertation was not to evaluate the numerical accuracy and stability of either of the numerical integration methods, since this has already been widely covered in various academic literature [Kreyzig 2006], [Moler 2004],[Press et al 1992] and both methods are accepted to provide results of reasonable accuracy based on the chosen stepping size variable used in the computation for each of the methods.

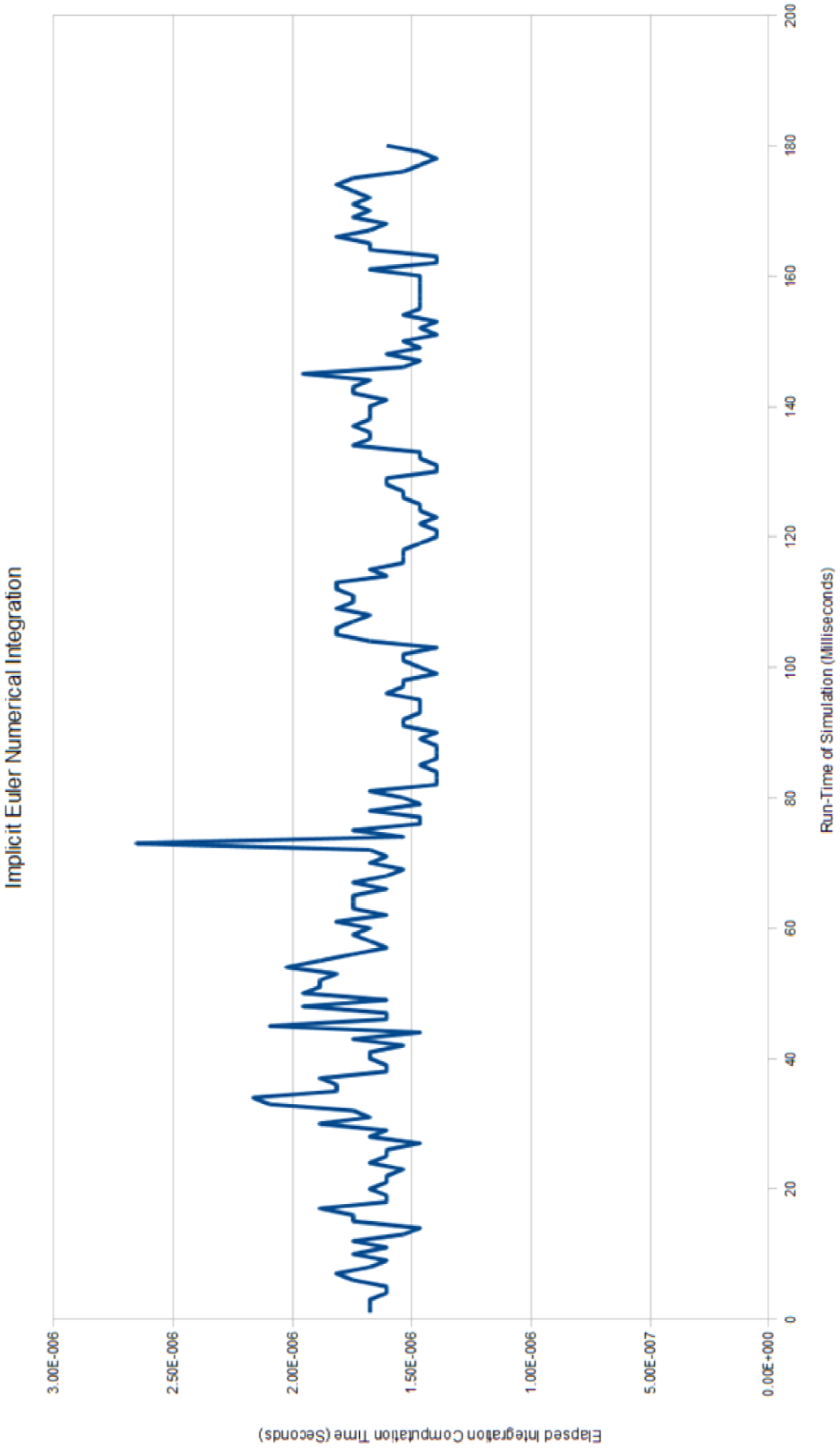

Figure 3: Implicit Euler numerical integration cycle evaluation graph.

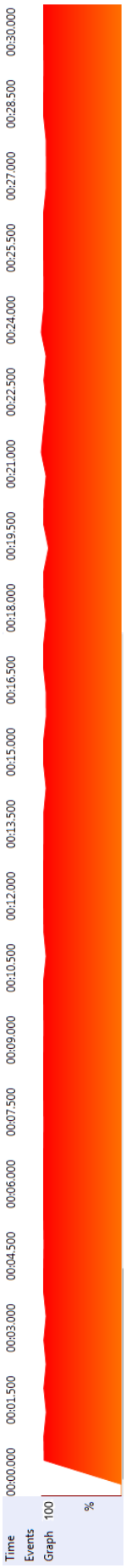

Figure 4: CPU usage graph with demo application running simulation using Implicit Euler.

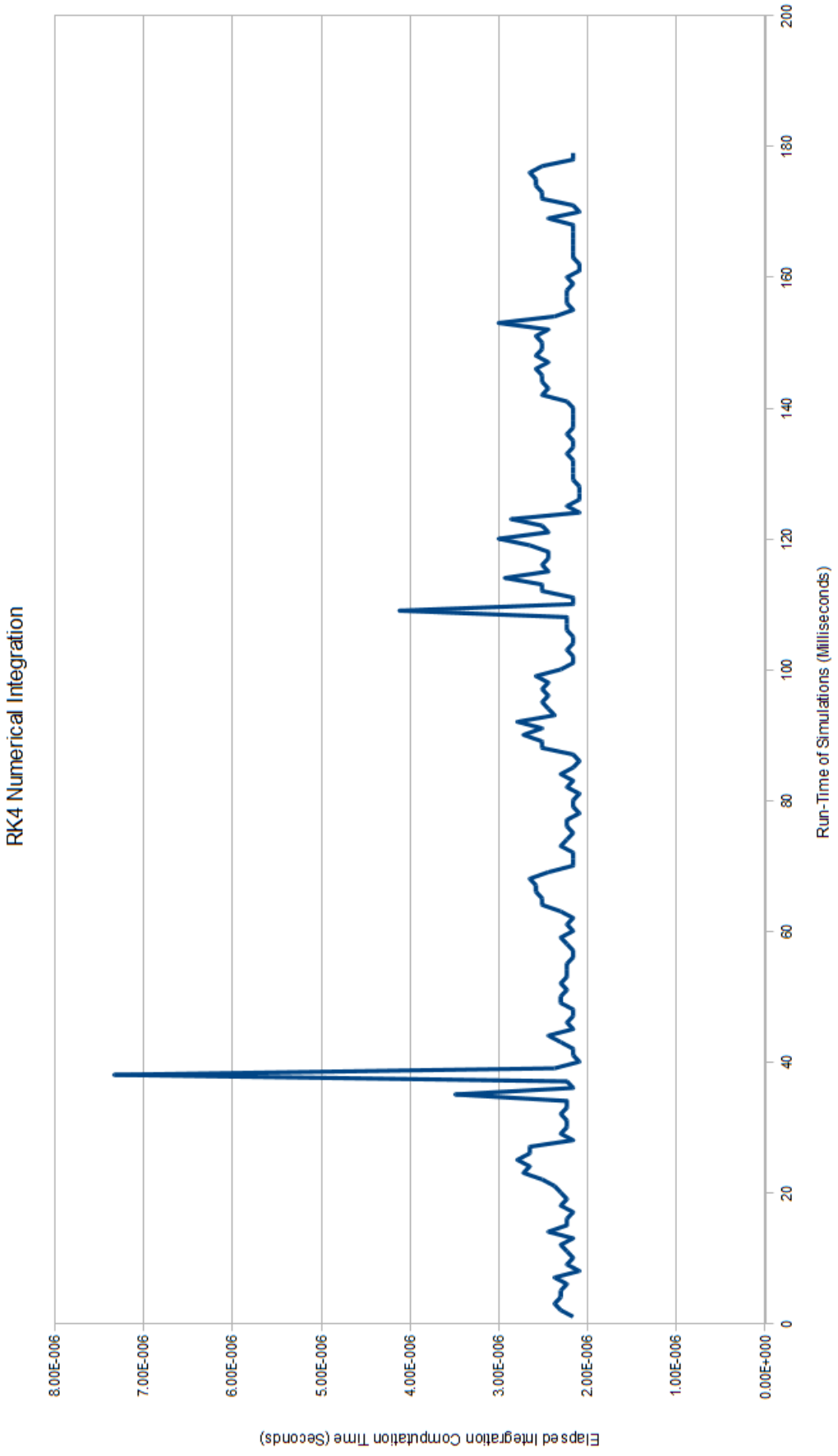

Figure 5: RK4 numerical integration cycle evaluation graph.

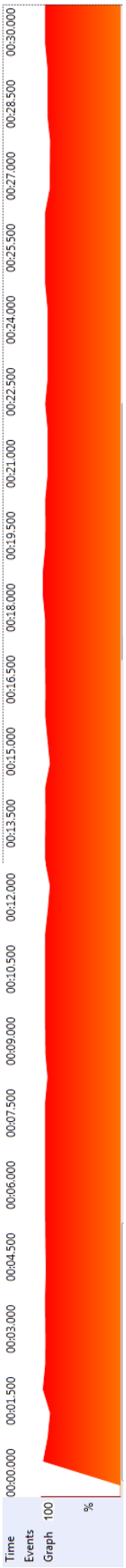

Figure 6: CPU usage graph with demo application running simulation using RK4.

5. Discussion

The use of numerical integration methods for rigid body simulation within a 3D environment does not always guarantee that the computed result will be aesthetically pleasing. Whether or not the accuracy of the chosen numerical integration method is high enough is often considered to be important only if it has any significant stability issues (as is the case with explicit Euler integration, see [DeLoura et al 2000]). Most of the time, for applications where scientific accuracy comes second to the aesthetic quality, the advantages of a numerical integration method are based solely on its performance criteria [McShaffrey et al 2009] (in most cases, this is either the amount of time or the number of CPU cycles taken to complete an integration cycle). In most interactive simulations, the use of analytic methods falls short because the application of Newton's laws of motion does not always provide a direct solution [Bourg 2002]. This is because the applications of Newton's laws of motion are being computed discretely. Thus numerical analysis methods have to be used.

It is often the case, especially for interactive entertainment applications such as video games, where the issue of performance is the main concern, that various "hacks" are used to achieve predictable results. The simulation of 3D rigid body motion in these scenarios is no exception to these practices, and often the case is that an inferior numerical computation method (often some variant of a predictor-corrector method) will be used in favour of its reduced processing requirements. [DeLoura et al 2000] makes this evident by stating that even explicit Euler integration can be used, if the approximated result is multiplied by a constant fractional value to keep it from diverging. The use of empirical modelling of simulation aspects in various 3D game scenarios is the main reason why numerical methods which flatter in comparison to those methods used in the field of scientific computing are used. One of the most advanced cross-overs from scientific computing to the realm of games programming in terms of numerical analysis, is with the use of the Verlet numerical integration method, and even that method is used sparsely and reserved for the most advanced real-time simulations (such as the dynamics of cloth particles, having its roots in the scientific simulation of molecular dynamics) [Verth et al 2008] , [Eberly 2010] .

Equations used to model the dynamics of rigid body motion in 3D space fall into the category of "stiff" ordinary differential equations – ODE's (as is the case with the use of the Newton-Euler equations of motion – see Appendix A). [Kreyszig 2006] describes the implicit Euler method as being best suited for approximating "stiff" ODE models. This is because generally, stiff ODEs require the stepping size to be a small value (usually no larger than 0.01, with the stepping size used in the demo application being set to 0.005) in order for them to be approximated accurately. The method used to select the correct stepping size for approximating stiff ODE models of rigid body motion using the selected numerical integration methods is explained by [Eberly 2010]. [Verth et al 2008] also mentions a very important behaviour for the implicit Euler method, which is that the approximation of a given stiff ODE using an implicit method actually dampens the energy of the given system. In terms of applying this to 3D rigid bodies, it changes the behaviour from having the rigid body have its linear and angular momentum increase to infinity to having it decrease (dampen) to a state of equilibrium.

The aesthetics of a given visual simulation are based on the viewer's perception of "realism". While many 3D physics engines and middleware solutions strive to meet the demand of cutting edge interaction in the form of interactive entertainment, such as video games, a more humble approach can be taken at the same time when developing a physics system for certain scenarios. For example, an inverse kinematics based character animation system would have very little use in a flight simulation application. Thus in most cases a physics system developed for a game or simulation will have specific functionality, depending on the scenarios being simulated [McShaffrey et al 2009].

However, all physics systems need to be able to handle the computation of general motion, whether in 2D or 3D, and this almost always requires the use of a stable numerical integration system [Bourg 2002].

While the mathematical and technical concepts of these topics have already been discussed, the aesthetics of the simulation play in important role in application where the main purpose is for entertainment. Imagine a football being kicked against a wall in a 3D game. Based on the collision detection and response implemented in the game system, the player of the game will expect the football to bounce against the wall in the opposing direction of its impact. But what if the ball penetrates the wall slightly? Or what if it gets stuck in the wall? Or what if the computation of the collision response is too realistic, and the ball just merely bounces slightly off the wall without any dramatic impact that the player would perhaps expect? This would surely break the aesthetic aspect of the game. The same concepts can be applied to the process of choosing an appropriate numerical integration system to handle the computation of motion of 3D rigid bodies in a game scene. If the player was playing a driving game and bumped into a box while driving his car, would the player care that the equating linear and angular velocity components of the box, as it is spinning in the air and colliding against the surroundings, are being enabled to do so using either the implicit Euler or RK4 numerical integration methods? Unless the player has technical knowledge of the underlying game physics system, the chances are he would not care what numerical integration system was being used. In this context, the programmers of the said game scenario will use the numerical integration system that is the fastest to compute. ***The general rule is that if something looks realistic and behaves realistic, thus can be visualised empirically, then it doesn't matter what numerical integration method is used. Therefore the cheapest method to accomplish the task should be used.*** [McShaffrey et al 2009] re-enforces this school of thought by stating how based on the psychology of the player, as long as the player interacts with the game environment in a way that they find pleasing, the underlying details of the simulation computation are not important.

[McShaffrey et al 2009] also mentions a very important fact about the simulation of realistic physics in video games and interactive applications: Such simulations are very expensive to compute accurately, therefore compromises and optimizations have to be made. This is a generally accepted fact when developing any real-time 3D applications on today's hardware platforms. While deciding on the use between implicit Euler and RK4 numerical integration methods for use in an application developed on today's consumer desktop computer hardware may not warrant worrying about the RK4 method being a few split milliseconds slower that the implicit Euler method - making the same decision when developing an interactive simulation on hardware platforms with limited resources carries much bigger consequences. The implementation, evaluation and/or maintenance of numerical integration methods may be a requirement for a programmer developing a new physics system for a new hardware or software platform. Though most of the time developers will rely on existing physics middleware [McShaffrey et al 2009], understanding the basic concepts of and being able to implement a 3D physics system with a focus on assessing a crucial component of such a system/ middleware solution (that being the appropriate numerical integration method) is beneficial.

The results from the research have thus answered the dissertation question, with the answer being that the implicit Euler method is the recommended method to use based on the provided evidence in its favour. As long as a fairly small stepping size is used with the implicit Euler method, a stable result will be approximated every time (see previous page). For more advanced features such as higher numerical precision and being able to approximate the velocity of the object backwards and forwards in time, during the run-time of the simulation, other numerical integration methods such as RK4 and Verlet integration should be used.

6. Conclusions and Future Work

The main principle of thought that can be taken away from this research is that physics modelling in real time 3D simulations is not always straightforward to implement. The transfer of Newton's Laws of Motion, from theory into practice, almost always requires a complete rethinking of the operational and theoretical aspects of those principals. Interactive 3D applications, such as games, thus rely on empirical modelling in order to simulate the required level of realism for a given scenario. However, with the advance in modern computing hardware accelerating at such a fast pace, it may come to no surprise that in a few years interactive 3D simulations and games will allows players to interact with the game environment in ways one only could have imagined (or only rendered off-line) a few years ago. Parallel processing is becoming increasingly popular, and thus GPU based computation will inevitably merge with CPU computing, to create a hybrid processing platform [1][2][3][4]. Such computing platforms will be able to perform vast amounts of very precise floating point operations, making them ideal for the implementation of various scientific computation models. Technologies like CUDA [5] and OpenCL [6] are already being used widely in the software development industry, finding use in fields as diverse as molecular dynamics and petroleum research, to more common use like GPU accelerated digital movie encoding and decoding.

But how will the new generation physics system be used in interactive applications and simulations? The advancement in parallel processing has naturally led the way for computer graphics applications to make use of real-time 3D rendering on even average consumer graphics processing hardware, where rendering methods that once could only be computed off-line or on high-end workstations, can now being computed in real-time on average consumer hardware. One such recent advancement in the field of interactive 3D simulation using modern hardware is with the use of finite element modelling and analysis techniques in real time. Such techniques could only be computed off-line up until a few years ago, and are widely used in the automotive and construction industries to test the integral structure of components before they are manufactured [7]. A recent technology called DMM (short for Digital Molecular Matter), developed by a company called Pixelux [8], is being used in real time applications, from video games [9] to military simulations [10]. The DMM technology allows for real-time finite-element level analysis and simulation of molecular matter of 3D objects. This allows for the very realistic simulation of material penetration, destruction, and very low level – high precision collision detection and response [7][10].

This then asks the question: How will the use of 3D rigid body simulation fit into the future frame of interactive computer simulations? [Bender et al 2012] answers this question by stating various examples and areas of computation where real-time simulation of 3D rigid body motion is still very applicable and has wide spread use. Furthermore, [Bender et al 2012] states that the use of middleware solutions like Nvidia's PhysX [11] and the Bullet physics engine [12] is making the use of 3D rigid body more easily accessible to developers. Thus, the use of 3D rigid body motion in real time physics simulations will be around for quite some time, even if new technologies like DMM, aimed at replacing the traditional model of physics computation, are emerging. The other reason for the popularity of using 3D rigid body motion instead of some sort of molecular dynamics model is that even if the later method can be implemented to run at decent speeds, implementing such system is far more difficult than implementing a 3D rigid body simulation system. This goes back to the iteration of the idea mentioned in the previous section, and that is that if something is simple to implement and provides the end-user with a desirable experience, then that is the technology/method that should be used.

However, the uncanny valley rule of computer simulation still applies to the principles of physics simulation, and new and advanced methods will replace the traditional 3D rigid body simulation model one day.

In conclusion, to summarise the findings of this dissertation research paper; the following points can be made:

- The implicit Euler method is the preferable numerical integration method to use when developing a simple 3D rigid body system.

- The RK4 method is a little bit slower (by few fractions of a millisecond – see Figure 5) than the implicit Euler method (see Figure 3), but does not provide any significant aesthetic improvements over the implicit Euler method.

- Both methods require a small stepping size in order to approximate the derived values of the equations of motion accurately every frame. This value should be somewhere in the fractional floating point range of 0.005 to 0.01 (see Discussion page).

- The aesthetics of a given 3D rigid body simulation system depends on the desired response from the user. This also depends on the nature of the application that the physics system is being developed for.

- When modelling something empirically for a real-time application, scientific accuracy comes second to the processing requirements of the implemented method. In turn, the processing requirements often come second to the aesthetic requirements of the real-time interactive simulation.

These five points summarize the findings of this dissertation. While there are many numerical integration methods available to use when developing 3D rigid body systems, it is often the case that the methods which are simple to implement, test and maintain, and provide a reasonable degree of accuracy to their approximations, are the preferred methods to use.

In terms of future work on this topic, various other methods could be investigated. Verlet integration is another popular method, and can be used for more advanced simulation aspects such as the simulation of cloth particles and being able to reverse the trajectory of a moving particle backwards in time. Collision detection and response is also another topic that can be researched and developed further. Finally, the idea of having multiple rigid bodies in the scene moving and colliding with each other requires further work in the field acceleration-based constrained motion, and is something that should be investigated in order to facilitate the development of a fully functional 3D rigid body system.

In conclusion, the realistic simulation of physic is a difficult thing to achieve in real time 3D. Often, programmers have to rely on various tricks to make something look “good” or “realistic”. What is good or realistic depends on the opinion of the end user of the application. While numerical integration methods facilitate the approximations of Newton’s laws of motion and aid in the simulation of real-life physically occurring phenomena, how the end user of the application experiences their interactions of the virtual environment presented to them, determines whether or not a given numerical approximation method does its job correctly.

Appendix A – Overview of Additional Mathematical Topics

Implementation of a 3D Rigid Body Inertia Tensor

The bounding box volume of a rigid body is relatively simple to model [DeLoura 200]. Thus an appropriate inertia tensor for a bounding box volume will need to be implemented. In classical mechanics, dm is the differential mass at a given point in the body, and in case of 3D rigid body, this point is the centre of the object. r is the distance from the centre of mass of the given object to the axis of rotation, and V is the integral volume representing the shape of the given rigid body. The moment of inertia around a particular axis can be modelled using the following formula [Lengyel 2011]:

$$I = \int_V r^2 \, dm$$

The density p of the rigid body is treated as a constant that is multiplied with each computed sum of the moments of inertia within the relative objects inertia tensor matrix. The moment of inertia of a rigid body is the sum of the total angular momentum about its centre of mass relative to any outside forces affecting it. With respect to the centre of the mass of the given 3D rigid body, the mass of the object can be defined as:

$$m = p \int_V dV$$

An inertia tensor is used to represent the computational model for three moments of inertia around each of the three vector axes of the objects coordinate frame and the three products of the inertia. An inertia tensor can be presented in the form of a 3x3 matrix as:

$$I = \sum_i m_i \begin{bmatrix} {y_i}^2 + {z_i}^2 & -x_i y_i & -x_i z_i \\ -x_i y_i & {x_i}^2 + {z_i}^2 & -y_i z_i \\ -x_i z_i & -y_i z_i & {x_i}^2 + {y_i}^2 \end{bmatrix}$$

[DeLoura et al 2000] states that the rigid bodies initial reference coordinate frame (vector frame), is relative to the world coordinate matrix. This requires the computation of the off-diagonal matrix elements. Since it is more common to model 3D rigid body motion based on its local transformations and rotations, the inertia tensor matrix can be diagonalized. This means setting the values of all off-diagonal elements to zero. This then reduces the inertia tensor matrix to contain only the eigenvectors of the diagonalized inertia tensor matrix. Based on these eigenvectors, the eigenvalues of a specified inertia tensor for a rigid body can be computed, relative to the objects local transformation and rotation coordinate frame.

The eigenvectors present the principal axes, while the eigenvalues present the principal moments of inertia of the rigid body. In practice, the principal axes are used to apply the rotational motion of a given 3D rigid body, where the rotations are described in the form of yawing, pitching and rolling, while the eigenvalues determine how much the body rotates in a given direction with respect to its local reference frame:

$$I = \sum_i m_i \begin{bmatrix} {y_i}^2 + {z_i}^2 & 0 & 0 \\ 0 & {x_i}^2 + {z_i}^2 & 0 \\ 0 & 0 & {x_i}^2 + {y_i}^2 \end{bmatrix}$$

Using the above matrix, the three moments of inertia for each of the axes of the given objects centre of mass can be calculated as [Lengyel 2011], [DeLoura et al 2000]:

$$I_{11} = \lim_{m_i \to 0} \sum m_i \, ({y_i}^2 + {z_i}^2) \to \; p \int_V \; (y^2 + z^2) \, dV$$

$$I_{22} = \lim_{m_i \to 0} \sum m_i \, ({x_i}^2 + {z_i}^2) \to \; p \int_V \; (x^2 + z^2) \, dV$$

$$I_{33} = \lim_{m_i \to 0} \sum m_i \, ({x_i}^2 + {y_i}^2) \to \; p \int_V \; (x^2 + y^2) \, dV$$

Now in order to make the newly diagonalized inertia tensor relative to the local reference frame of the given 3D rigid body, it needs to be transformed from the world coordinates to local coordinates of the given 3D rigid body reference frame. This is done by finding the inverse of the inertia tensor, where the matrix *R* is used to represent the rotation of the 3D rigid body in world space. Then the inverse can be defined as [Baraff97a] , [Lengyel 2011]:

$$I^{-1} = R . I^{-1} . R^T$$

The previous inertia tensor [inverse] becomes:

$$I = \sum_i m_i \begin{bmatrix} \dfrac{1}{{y_i}^2 + {z_i}^2} & 0 & 0 \\ 0 & \dfrac{1}{{x_i}^2 + {z_i}^2} & 0 \\ 0 & 0 & \dfrac{1}{{x_i}^2 + {y_i}^2} \end{bmatrix}$$

As mentioned above, the optimal bounding volume model to use in terms of simplicity is the bounding box volume. $f(l), f(w)$ and $f(h)$ represent the principal axes approximations of a rectangular box volume. The edges of the box volume are parallel, so the triple volume integral is evaluated over the domain of $l_0 \leq x \leq l_1$, $w_0 \leq y \leq w_1, h_0 \leq z \leq h_1$, where l_0 to l_1 goes from $-\frac{l}{2}$ to $\frac{l}{2}$, w_0 to w_1 goes from $-\frac{w}{2}$ to $\frac{w}{2}$ and h_0 to h_1 goes from $-\frac{h}{2}$ to $\frac{h}{2}$ [Garity 1996].
Note that the box volume has a constant density (p), and the total mass of the box volume is calculated as $m = plwh$. The volume integral of a box can be defined as [Lengyel 2011]:

$$Box\ Volume = p \iiint_V f(l, w, h)\, dx\, dy\, dz$$

$$\rightarrow f(l) = I_{11} = y^2 + z^2$$
$$\rightarrow f(w) = I_{22} = x^2 + z^2$$
$$\rightarrow f(h) = I_{33} = x^2 + y^2$$

$$\rightarrow p \int_{-\frac{h}{2}}^{\frac{h}{2}} \int_{-\frac{w}{2}}^{\frac{w}{2}} \int_{-\frac{l}{2}}^{\frac{l}{2}} f(l, w, h)\, dx\, dy\, dz$$

$$\rightarrow f(l) = p \int_{-\frac{h}{2}}^{\frac{h}{2}} \int_{-\frac{w}{2}}^{\frac{w}{2}} \int_{-\frac{l}{2}}^{\frac{l}{2}} (y^2 + z^2) dx\, dy\, dz = \frac{1}{12}\, plwh(w^2 + h^2)$$
$$\rightarrow f(w) = p \int_{-\frac{h}{2}}^{\frac{h}{2}} \int_{-\frac{w}{2}}^{\frac{w}{2}} \int_{-\frac{l}{2}}^{\frac{l}{2}} (x^2 + z^2) dx\, dy\, dz = \frac{1}{12}\, plwh(l^2 + h^2)$$
$$\rightarrow f(h) = p \int_{-\frac{h}{2}}^{\frac{h}{2}} \int_{-\frac{w}{2}}^{\frac{w}{2}} \int_{-\frac{l}{2}}^{\frac{l}{2}} (x^2 + y^2) dx\, dy\, dz = \frac{1}{12}\, plwh(l^2 + w^2)$$

This in turn gives the final definition of the [inverse] box inertia tensor [Lengyel 2011], [DeLoura et al 2000]:

$$I_{box} = \begin{bmatrix} \frac{1}{12}\, m(h^2 + d^2) & 0 & 0 \\ 0 & \frac{1}{12} m(w^2 + d^2) & 0 \\ 0 & 0 & \frac{1}{12} m(w^2 + d^2) \end{bmatrix}$$

Calculation of Forces Impacting Each Affected Rigid Body

The mass properties of a rigid body can be broken down into: total body mass, the centre of the mass and the moment of inertia; they are referred to as mass properties of a rigid body [Bourg 2001]. The simulation of 3D rigid body motion is essentially concerned with the modelling of linear and angular motion of a rigid body. The modelling of linear and angular motion of a given rigid body is essentially the application of the principals of motion described in Newton's Second Law of Motion:

The change of momentum of a given body is proportional to the resultant force acting on the body, where the resulting momentum of the body moves in the same direction as the resultant force.

This is commonly expressed in the form of the equation [Meriam et al 2008]:

$$\boldsymbol{F} = m\mathbf{a}$$

Where $\boldsymbol{F}$ is the impulse force acting on the object (in vector form), m is the mass of the object and $\boldsymbol{a}$ is acceleration (also in vector form). The sum of all the forces taken as the resultant force needs to be applied for all the coordinate axes in 3D space, both for the force vectors and the corresponding acceleration vectors [Bourg 2001]:

$$\sum \boldsymbol{F_x} = m\mathbf{a_x}$$

$$\sum \boldsymbol{F_y} = m\mathbf{a_y}$$

$$\sum \boldsymbol{F_z} = m\mathbf{a_z}$$

Since the sum of all the forces acting on the body is equal to the rate of change of the body's momentum over time, the linear momentum of a rigid body can be defined as [Bourg 2001]:

$$\boldsymbol{G} = m\boldsymbol{V}$$

Where $\boldsymbol{G}$ is the linear momentum that is defined as the product of the rigid bodies mass, multiplied by the bodies current velocity and relative to the centre of its mass. Rigid bodies do not have shifting mass as the mass of the rigid body can be said to be of constant value. Therefore acceleration of a rigid can be defined as the rate of change of velocity relative to time [Bourg 2001]:

$$\frac{d\boldsymbol{G}}{dt} = m\,\frac{d\boldsymbol{V}}{dt}$$

The linear momentum of a rigid body can be defined as the sum of all the forces acting on the body, which are the product of the mass of the body multiplied by the change in velocity of the body, relative to its centre of mass [Bourg 2001]:

$$\sum \boldsymbol{F} = \frac{d\boldsymbol{G}}{dt} = m\mathbf{a}$$

In order to model the rotational force acting on a given rigid body, angular momentum needs to be considered. The angular momentum of a rigid body is the rotation of the body relative to its local reference frame (the eigenvectors described in the previous section), and is calculated as a product of a vector quantity (the eigenvalues also described in the previous section). The change of a given rigid bodies angular momentum caused by a force acting on it is called a torque, or the sum of moments. The sum of all the moments for a rigid body, relative to its centre of mass (denoted as $\mathbf{M}_{cm}$), is computed as the rate of change in the body's angular momentum over time and where $\boldsymbol{L}$ is computed as the cross product between the force acting on the object and the displacement vector **r** of the force (the distance from the current position of the force vector **F** to its origin), referred to as the angular momentum [Bourg 2001]:

$$\sum \mathbf{M}_{cm} = \frac{d\boldsymbol{L}}{dt}$$

$$\boldsymbol{L} = \boldsymbol{r} \times \boldsymbol{F}$$

In turn, the angular momentum of a rigid body is computed as the sum of moments about the body's centre of mass (which is the body's default rotation axis), where $\boldsymbol{F_i}$ is defined as $m_i(\boldsymbol{\omega} \times \boldsymbol{r}_i)$ and $\boldsymbol{L}_{cm}$ is defined as [Bourg 2001]:

$$\boldsymbol{L}_{cm} = \sum \boldsymbol{r}_i \times \boldsymbol{F}_i$$

The angular velocity of the rigid body denoted as ω, relative to its axis of rotation and $(\boldsymbol{\omega} \times \boldsymbol{r}_i)$ is the angular momentum of the ***i***th particle of the rigid body. The angular velocity can then be applied to a given axis of the rigid body (via multiplying it with the corresponding eigenvector element in the inertia tensor matrix). The angular velocity is calculated for all volume particles making up the rigid body, where **r** is the distance from the given particle of the rigid body to the rigid body's centre of mass ($\boldsymbol{L}_{cm}$), and is defined as [Bourg 2001]:

$$\boldsymbol{L}_{cm} = I\boldsymbol{\omega}$$

Applying this to the original inertia tensor definition above, the angular velocity of a rigid body about a given axis can be calculated as [DeLoura et al 2000]:

$$\boldsymbol{L}_{cm} = \sum_i m_i \begin{bmatrix} {y_i}^2 + {z_i}^2 & -x_i y_i & -x_i z_i \\ -x_i y_i & {x_i}^2 + {z_i}^2 & -y_i z_i \\ -x_i z_i & -y_i z_i & {x_i}^2 + {y_i}^2 \end{bmatrix} \boldsymbol{\omega}$$

As with the linear momentum where the derivative of the body's momentum with respect to time is its acceleration, the derivative of the angular momentum is the angular acceleration of the rigid body about a given axis, where $\frac{d\omega}{dt}$ is calculated as the vector $\boldsymbol{\alpha}$. This allows angular momentum of a rigid body to be defined as [Bourg 2001]:

$$\sum \boldsymbol{M}_{cm} = I\boldsymbol{a}$$

Since the coordinates of the rigid body inertia tensor are relative to its local frame of reference, the sum of moments and the angular velocity relative to the non-inverse inertia tensor can be defined as [Bourg 2001]:

$$\sum \boldsymbol{M}_{cm} = \frac{d\boldsymbol{L}_{cm}}{dt} = I\left(\frac{d\boldsymbol{\omega}}{dt}\right) + (\boldsymbol{\omega} \times (I\boldsymbol{\omega}))$$

The linear velocity of a given rigid body is defined as $\frac{d\boldsymbol{V}}{dt}$ and the relationship with the angular velocity time derivate is defined as $\frac{d\boldsymbol{V}}{dt}$ + ($\boldsymbol{\omega} \times \boldsymbol{V}$). The difference between the linear velocity and the angular velocity is that the angular velocity derivative has an additive component consisting of the cross product difference between the angular velocity and the linear velocity of the rigid body.

In classical mechanics, the time-relative derivatives of the linear and angular velocities of the rigid bodies are called the *Newton-Euler equations of rigid body motion*, and can be summarized as [DeLoura et al 2000]:

For linear motion:

$$\frac{d\boldsymbol{r}}{dt} = \mathbf{V}$$

$$\frac{d\boldsymbol{V}}{dt} = \sum \frac{F}{m}$$

For angular motion:

$$\frac{d\boldsymbol{R}}{dt} = \boldsymbol{\omega} * \boldsymbol{R}$$

$$\frac{d\omega}{dt} = I^{-1}\left[\boldsymbol{M}_{cm} - \left(\boldsymbol{\omega} \times (I\boldsymbol{\omega})\right)\right]$$

Where $\boldsymbol{R}$ is the eigenvector of the inertia tensor representing the local XYZ axes (diagonal elements) of the body's inertia tensor.

Overview of Selected Integration Methods

Numerical integration is used in a 3D rigid body system to approximate the future position of the rigid body in the game scene, given the programmer defined time-stepping parameters. Based on the initial values of each of the four differential definitions (stated in the above section), the linear and angular motion of a rigid body can be approximated using numerical integration techniques. The four variables that need to be approximated are:

$\boldsymbol{r_{i+1}}$ *- The next vector coordinate position of the body (transformed from local to world coordinates)*

$\boldsymbol{V_{i+1}}$ *– The next linear momentum values of the body*

$\boldsymbol{R_{i+1}}$ *- The next angular momentum values of the body*

$\boldsymbol{\omega_{i+1}}$ *– The next angular acceleration values of the body*

Given the initial values of each of these variables, the next incremented value set can be approximated. For example, to approximate the next vector position coordinates of the body ($\boldsymbol{r_{i+1}}$), using the standard method of Euler integration, would be approximated as [Verth et al 2008]:

$$\boldsymbol{r_{i+1}} \approx \boldsymbol{r_i} + h_i \boldsymbol{r'_i}$$
$$\boldsymbol{r'_i} = \boldsymbol{V_i}$$
$$\boldsymbol{r_{i+1}} \approx \boldsymbol{r_i} + h_i \boldsymbol{V_i}$$
$$\boldsymbol{V_{i+1}} \approx \boldsymbol{V_i} + h_i \boldsymbol{V'_i}$$
$$\boldsymbol{V'_i} = \sum \frac{\boldsymbol{F}}{m}$$
$$\boldsymbol{V_{i+1}} \approx \boldsymbol{V_i} + h_i \left(\sum \frac{F}{m}\right)$$

The same method as above would be used to approximate the values of the other variables in the rigid body system. However, this method of numerical integration introduces an exponential error to the approximation of the results. This makes the standard Euler integration method unstable [Kreyszig 2006], [Moler 2004].

[DeLoura et al 2000] describes that when such an explicit approximation method is applied to a rigid body system, it causes very significant stability errors. The general case is that the larger the volume of the body is, the smaller the stepping size must be in order for accurate approximations to be compute

One way to increase stability is to use an implicit version of the Euler integration method, called the Implicit Euler Integration method (also known as the Backward Euler method). The Implicit Euler Integration method makes use of the new approximated values of the given function, rather than the previous values. It is defined as [Kreyszig 2006]:

$$y_{n+1} \approx y_n + hf(X_{n+1}, y_{n+1})$$

Given the initial value of the function, the new approximated value can be calculated. The new values X_{n+1} and y_{n+1} can be approximated using the Midpoint Method. [Moler 2004] describes the Midpoint Method as an extension to the standard Euler integration method, where a second function evolution is added with two additional delta functions. This in fact makes the Implicit Euler method a Runge-Kutta Order Two integration method, as most numerical integration methods are branches of Runge-Kutta methods.

The Midpoint Method can be defined as:

$$k_1 = f(x_n, y_n)$$
$$k_2 = f\left(x_n + \frac{h}{2}, y_n + \frac{h}{2}\ k_1\right)$$
$$y_{n+1} = y_n + hk_2$$
$$x_{n+1} = x_n + h$$

The Runge-Kutta Order Four (RK4) integration method is the other method this project aims to evaluate. The RK4 method is more computationally costly, taking 4 calculation sets, but is said to provide a better degree of accuracy when it comes to approximating stiff ODE's [Verth et al 2008]. Given an initial value problem in the form of:

$$y' = f(x_n, y_n), \qquad y(x_n) = y_n$$

The approximation using the RK4 method is defined as [Kreyszig 2006] [Moler 2004][Press et al 1992]:

$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$
$$x_{n+1} = x_n + h$$

, where each of the four delta functions is defined as:

$$k_1 = hf(x_n, y_n)$$
$$k_2 = hf\left(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1\right)$$
$$k_3 = hf\left(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_2\right)$$
$$k_4 = hf(x_n + h, y_n + k_3)$$

The RK4 method makes use of the weighted average of the four delta functions to approximate the result of the given ODE. The four delta functions are used to approximate four delta gradients of the numerical function interval, where k_1 approximates the delta gradient at the beginning of the interval, k_2 and k_3 at the mid- point of the interval and k_4 approximates the end of the interval. The mid-points are weighed higher due to the predictor-corrector nature of the RK4 algorithm [Moler 2004].

Appendix B – Overview of Program Implementation

This section will provide code extracts to important functions that were used in the calculation and obtainment of the research results. The demo application was programmed using C# and the following code examples were based on code examples, demo and articles by [Bourg 2002], [Verth et al 2008], [DeLoura et al 2000] and [Eberly 2010]. Some additionally code functionality was based on tutorials from [13], [14] and [15].

The core of the functionality is contained in the Box.cs interface, which is used as the main interface for the implementation of the box inertia tensor and dynamics calculations affecting its rigid body state in the 3D scene.

```
//////////////////////////////////////////////////////////////////////////////////////////
///////////////
// Simple 3d box shape rigid body interface.
// Written by Vladeta Stojanovic (0602920@live.abertay.ac.uk)
//
// Version: 130412_01
//////////////////////////////////////////////////////////////////////////////////////////

using System;
using System.Diagnostics;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;

namespace simple_3D_1_
{
    class Box
    {
        //Primary varaibles
        public Matrix worldMatrix;
        public Quaternion orientation = Quaternion.Identity;
        public Model boxModel;

        //OOBB variables
        public BoundingBox oobb;
        public Vector3 OOBMin;
        public Vector3 OOBMax;
        public Ray posXRay;
        public Ray negXRay;
        public Ray posZRay;
        public Ray negZRay;
        public Ray posYRay;
        public Ray negYRay;

        //force physics states
        public Vector3 forcePosition;
        public Vector3 forceVector;
        public Vector3 rotationForcePosition;
        public Vector3 rotationForceVector;
        public bool activeFromStart;

        //box tensor variabels
        public float momentOfInteriaX;
        public float momentOfInteriaY;
        public float momentOfInteriaZ;

        public Matrix rotationMatrix;
        float XrotLerp;
        float YrotLerp;
        float ZrotLerp;
```

```
        //rotation variables
        private float RadiusAroundX = 0;
        private float ForceAroundX;
        private float RadiusAroundY = 0;
        private float ForceAroundY;
        private float RadiusAroundZ = 0;
        private float ForceAroundZ;
        private float AlphaX;
        private float BetaX;
        private float GammaX;
        private float AlphaY;
        private float BetaY;
        private float GammaY;
        private float AlphaZ;
        private float BetaZ;
        private float GammaZ;

        //object physics states
        public Vector3 position;
        public Vector3 velocity;
        public Vector3 angularVelocity;
        public Vector3 acceleration;
        public Vector3 angularAcceleration;
        public Vector3 currentThrust;
        public float friction;
        public float size;
        public float mass;

        ////////////////////
        //Diagnostics
        ////////////////////

        //processing time
        public HiPerfTimer timer;
        public double rk4IntegrationTime;
        public double eulerIntegrationTime;

        public BoundingBox CalculateBoundingBox(Model m_model)
        {
             Vector3    modelMax    =    new    Vector3(float.MinValue,    float.MinValue,
             float.MinValue);
            Vector3    modelMin    =    new    Vector3(float.MaxValue,    float.MaxValue,
            float.MaxValue);
            Matrix[] m_transforms = new Matrix[m_model.Bones.Count];

            foreach (ModelMesh mesh in m_model.Meshes)
            {
               Vector3    meshMax    =    new    Vector3(float.MinValue,    float.MinValue,
      float.MinValue);
                Vector3    meshMin    =    new    Vector3(float.MaxValue,    float.MaxValue,
      float.MaxValue);

                foreach (ModelMeshPart part in mesh.MeshParts)
                {

                    int stride = part.VertexBuffer.VertexDeclaration.VertexStride;

                    byte[] vertexData = new byte[stride * part.NumVertices];
                    part.VertexBuffer.GetData(part.VertexOffset * stride, vertexData, 0,
part.NumVertices, 1);

                    Vector3 vertPosition = new Vector3();
                    for (int ndx = 0; ndx < vertexData.Length; ndx += stride)
                    {
                        vertPosition.X = BitConverter.ToSingle(vertexData, ndx);
                        vertPosition.Y    =    BitConverter.ToSingle(vertexData,    ndx    +
sizeof(float));
                        vertPosition.Z    =    BitConverter.ToSingle(vertexData,    ndx    +
sizeof(float) * 2);
```

```
                        meshMin = Vector3.Min(meshMin, vertPosition);
                        meshMax = Vector3.Max(meshMax, vertPosition);
                    }
                }

                meshMin                    =                    Vector3.Transform(meshMin,
m_transforms[mesh.ParentBone.Index]);
                meshMax                    =                    Vector3.Transform(meshMax,
m_transforms[mesh.ParentBone.Index]);

                modelMin = Vector3.Min(modelMin, meshMin);
                modelMax = Vector3.Max(modelMax, meshMax);


            }

            return new BoundingBox(modelMin, modelMax);
        }

        //get rid of all this
        public void SetVariables()
        {
            timer = new HiPerfTimer();
            oobb = new BoundingBox();

            //setup the physics properties here
            size = 1.0f;
            mass = 10.0f;
            angularVelocity = Vector3.Zero;
            acceleration = Vector3.Zero;
            angularAcceleration = Vector3.Zero;
            orientation = Quaternion.Identity;

            //setup the inertia box tensor
            momentOfInteriaX  =  0.83f  *  mass  *  ((float)Math.Pow(9.46f,  2.0f)  +
(float)Math.Pow(9.46f, 2.0f));
            momentOfInteriaY  =  0.83f  *  mass  *  ((float)Math.Pow(9.36f,  2.0f)  +
(float)Math.Pow(9.36f, 2.0f));
            momentOfInteriaZ  =  0.83f  *  mass  *  ((float)Math.Pow(9.46f,  2.0f)  +
(float)Math.Pow(9.46f, 2.0f));

            rotationMatrix = Matrix.CreateFromQuaternion(orientation);
            friction = 0.004f;
        }

        public void CalculateBoundingBox()
        {
            oobb = CalculateBoundingBox(boxModel);
        }

        public void ApplyImpulse(Vector3 impulseForce, Vector3 position)
        {
            ApplyForce(impulseForce);
            ApplyAngularForce(impulseForce, position);
        }

        public void ApplyGravity(GameTime gameTime, Vector3 gravity)
        {
            if (position.Y > 0.0f)
            {
                position += gravity;
            }
        }

        public void ApplyCollisionImpulse(BoundingBox colModel, GameTime gameTime)
        {
            if (oobb.Intersects(colModel) && position.Y < 0.0f)
            {
                Trace.WriteLine("Collision with floor");
```

```
                position.Y = 0.0f;

                angularAcceleration.X    =    MathHelper.Lerp(angularAcceleration.X,    0,
friction * (float)gameTime.ElapsedGameTime.Milliseconds);
                angularAcceleration.Y    =    MathHelper.Lerp(angularAcceleration.Y,    0,
friction * (float)gameTime.ElapsedGameTime.Milliseconds);
                angularAcceleration.Z    =    MathHelper.Lerp(angularAcceleration.Z,    0,
friction * (float)gameTime.ElapsedGameTime.Milliseconds);

                XrotLerp     =     MathHelper.Lerp(orientation.X,     0,     0.00005f     *
(float)gameTime.ElapsedGameTime.Milliseconds);
                YrotLerp     =     MathHelper.Lerp(orientation.Y,     0,     0.00005f     *
(float)gameTime.ElapsedGameTime.Milliseconds);
                ZrotLerp     =     MathHelper.Lerp(orientation.Z,     0,     0.00005f     *
(float)gameTime.ElapsedGameTime.Milliseconds);

                orientation   =   Quaternion.CreateFromYawPitchRoll(XrotLerp,   YrotLerp,
ZrotLerp);

            }
        }

        public void ApplyPlaneCollisionResponse(Vector3 impulse, GameTime gameTime)
        {
            float delta = (1 - (0.005f * gameTime.ElapsedGameTime.Milliseconds));

            Vector3 reactionForce = (new Vector3(impulse.X * mass, -impulse.Y * mass,
impulse.Z * mass)) * delta;
            Vector3 planeNormal = Vector3.Cross(position, impulse);
            Vector3.Normalize(planeNormal);

            ApplyForce(reactionForce);
            ApplyAngularForce(reactionForce, planeNormal);

            position.Y     =     MathHelper.Lerp(position.Y,     0,     friction     *
gameTime.ElapsedGameTime.Milliseconds);
        }

        public void ApplyFriction(GameTime gameTime)
        {
            acceleration.X    =    MathHelper.Lerp(acceleration.X,    0,    friction    *
(float)gameTime.ElapsedGameTime.Milliseconds);
            acceleration.Y    =    MathHelper.Lerp(acceleration.Y,    0,    friction    *
(float)gameTime.ElapsedGameTime.Milliseconds);
            acceleration.Z    =    MathHelper.Lerp(acceleration.Z,    0,    friction    *
(float)gameTime.ElapsedGameTime.Milliseconds);
        }

        public void ApplyForce(Vector3 force)
        {
            acceleration = force / mass;
        }

        public void ApplyAngularForce(Vector3 force, Vector3 pos)
        {
            forceVector = force;
            forcePosition = pos;
            rotationForceVector             =             Vector3.Transform(forceVector,
Matrix.Invert(rotationMatrix));
            rotationForcePosition           =           Vector3.Transform(forcePosition,
Matrix.Invert(rotationMatrix));

            //rotation calculations around the x, y and z axes
            ForceAroundX    =    (float)Math.Sqrt(Math.Pow(rotationForceVector.Y,    2)    +
Math.Pow(rotationForceVector.Z, 2));
            ForceAroundY    =    (float)Math.Sqrt(Math.Pow(rotationForceVector.X,    2)    +
Math.Pow(rotationForceVector.Z, 2));
            ForceAroundZ    =    (float)Math.Sqrt(Math.Pow(rotationForceVector.X,    2)    +
Math.Pow(rotationForceVector.Y, 2));
```

```
            //X axis
            if (ForceAroundX != 0.0f)
            {
                RadiusAroundX = (float)Math.Sqrt(Math.Pow(rotationForcePosition.Y, 2) +
Math.Pow(rotationForcePosition.Z, 2));

                AlphaX            =            (float)Math.Atan2(rotationForcePosition.Y,
rotationForcePosition.Z);

                BetaX = (float)Math.Atan2(rotationForceVector.Y, rotationForceVector.Z);

                GammaX = ((float)Math.PI - BetaX) + AlphaX;

                RadiusAroundX = (float)Math.Sin(GammaX) * RadiusAroundX;

                angularAcceleration.X   =   (float)(ForceAroundX   *   RadiusAroundX)   /
momentOfInteriaX; //this should perhaps be negative...
            }

            //Y axis
            if (ForceAroundY != 0.0f)
            {
                RadiusAroundY = (float)Math.Sqrt(Math.Pow(rotationForcePosition.X, 2) +
Math.Pow(rotationForcePosition.Z, 2));

                AlphaY            =            (float)Math.Atan2(rotationForcePosition.X,
rotationForcePosition.Z);

                BetaY = (float)Math.Atan2(rotationForceVector.X, rotationForceVector.Z);

                GammaY = ((float)Math.PI - BetaY) + AlphaY;

                RadiusAroundY = (float)Math.Sin(GammaY) * RadiusAroundY;

                angularAcceleration.Y   =   (float)(ForceAroundY   *   RadiusAroundY)   /
momentOfInteriaY;
            }

            //Z axis
            if (ForceAroundZ != 0.0f)
            {
                RadiusAroundZ = (float)Math.Sqrt(Math.Pow(rotationForcePosition.X, 2) +
Math.Pow(rotationForcePosition.Y, 2));

                AlphaZ            =            (float)Math.Atan2(rotationForcePosition.Y,
rotationForcePosition.X);

                BetaZ = (float)Math.Atan2(rotationForceVector.Y, rotationForceVector.X);

                GammaZ = ((float)Math.PI - BetaZ) + AlphaZ;

                RadiusAroundZ = (float)Math.Sin(GammaZ) * RadiusAroundZ;

                angularAcceleration.Z   =   (float)(ForceAroundZ   *   RadiusAroundZ)   /
momentOfInteriaZ;
            }
        }
```

```
        /// ////////////////////////////
        /// Intergration methods
        /// ////////////////////////////
        public void ImplicitEulerStep(float dt)
        {
            timer.Start();

            Vector3 F;
            Vector3 A;
            Vector3 VNew;
            Vector3 SNew;
            Vector3 k1, k2;

            F = (currentThrust - (new Vector3(friction, friction, friction)) * velocity);
            A = F / mass;
            k1 = dt * A / 1000.0f;

            F = (currentThrust - (new Vector3(friction, friction, friction)) * (velocity
+ k1));
            A = F / mass;
            k2 = dt * A / 1000.0f;

            VNew = velocity + (k1 + k2) / 2;
            SNew = position + VNew * dt;

            angularVelocity += angularAcceleration / 60.0f;
            velocity = VNew;
            position = SNew;

            velocity *= (1 - (0.005f * dt));
            angularVelocity *= (1 - (0.005f * dt));

            timer.Stop();
            eulerIntegrationTime = timer.Duration;
        }

        public void Rk4Step(float dt)
        {
            timer.Start();

            Vector3 F;
            Vector3 A;
            Vector3 VNew;
            Vector3 SNew;
            Vector3 k1, k2, k3, k4;

            F = (currentThrust - (new Vector3(friction, friction, friction)) * velocity);
            A = F / mass;
            k1 = dt * A / 1000.0f;

            F = (currentThrust - (new Vector3(friction, friction, friction)) * (velocity
+ k1/2));
            A = F / mass;
            k2 = dt * A / 1000.0f;

            F = (currentThrust - (new Vector3(friction, friction, friction)) * (velocity
+ k2/2));
            A = F / mass;
            k3 = dt * A / 1000.0f;

            F = (currentThrust - (new Vector3(friction, friction, friction)) * (velocity
+ k3));
            A = F / mass;
            k4 = dt * A / 1000.0f;

            VNew = velocity + (k1 + 2*k2 + 2*k3 + k4) / 6;
            SNew = position + VNew * dt;

            angularVelocity += angularAcceleration / 60.0f;
            velocity = VNew;
```

```
            position = SNew;

            velocity *= (1 - (0.005f * dt));
            angularVelocity *= (1 - (0.005f * dt));

            timer.Stop();
            rk4IntegrationTime = timer.Duration;
        }

        public void StepExplicitEuler(float dt)
        {
            velocity += (acceleration / 60.0f);

            angularVelocity += angularAcceleration / 60.0f;
            position += velocity * dt;

            velocity *= (1 - (0.005f * dt));
            angularVelocity *= (1 - (0.005f * dt));
        }


        public void Update(GameTime gameTime)
        {
            Matrix m = Matrix.CreateFromQuaternion(orientation);
            m = m
                * Matrix.CreateFromAxisAngle(m.Right, angularVelocity.X)
                * Matrix.CreateFromAxisAngle(m.Forward, angularVelocity.Z)
                * Matrix.CreateFromAxisAngle(m.Up, angularVelocity.Y);

            orientation = Quaternion.Normalize(Quaternion.CreateFromRotationMatrix(m));

            worldMatrix        =        Matrix.CreateFromQuaternion(orientation)        *
Matrix.CreateTranslation(position);
        }

        public void Draw(Effect boxEffect, Model boxModel_)
        {
            Matrix[] transforms = new Matrix[boxModel_.Bones.Count];
            boxModel_.CopyAbsoluteBoneTransformsTo(transforms);

            foreach (ModelMesh mesh in boxModel_.Meshes)
            {
                //Apply the desired specEffect to rendered geometry
                foreach (ModelMeshPart part in mesh.MeshParts)
                {
                    part.Effect = boxEffect;
                    part.Effect.Parameters["World"].SetValue(worldMatrix);

                    //now update everything for the oobb...
                    OOBMin = new Vector3(float.MaxValue, float.MaxValue, float.MaxValue);
                    OOBMax = new Vector3(float.MinValue, float.MinValue, float.MinValue);
                    int vertexStride = part.VertexBuffer.VertexDeclaration.VertexStride;
                    int vertexBufferSize = part.NumVertices * vertexStride;
                    float[] vertexData = new float[vertexBufferSize / sizeof(float)];
                    part.VertexBuffer.GetData<float>(vertexData);

                    for  (int  i  =  0;  i  <  vertexBufferSize  /  sizeof(float);  i  +=
vertexStride / sizeof(float))
                    {
                        Vector3     transformedPosition     =     Vector3.Transform(new
Vector3(vertexData[i],  vertexData[i  +  1],  vertexData[i  +  2]),  Matrix.Identity  *
worldMatrix);
                        OOBMin = Vector3.Min(OOBMin, transformedPosition);
                        OOBMax = Vector3.Max(OOBMax, transformedPosition);
                    }
                }

                mesh.Draw();
```

```
                //Update OOB
                oobb = new BoundingBox(OOBMin, OOBMax);
                Vector3 rayTransform = Vector3.Transform(position, rotationMatrix);

                posXRay = new Ray(rayTransform, Vector3.Right);
                posYRay = new Ray(rayTransform, Vector3.Up);
                posZRay = new Ray(rayTransform, -Vector3.Forward);

                negXRay = new Ray(rayTransform, -Vector3.Right);
                negYRay = new Ray(rayTransform, -Vector3.Up);
                negZRay = new Ray(rayTransform, Vector3.Forward);
            }
        }
    }
}
```

The next two important classes are the high precision timer class (Timer.cs – HiPerTimer class) and the logging system (Log.cs – Logger class):

```
using System;
using System.Runtime.InteropServices;
using System.ComponentModel;
using System.Threading;

namespace simple_3D_1_
{
    class HiPerfTimer
    {
        [DllImport("Kernel32.dll")]
        private static extern bool QueryPerformanceCounter(
            out long lpPerformanceCount);

        [DllImport("Kernel32.dll")]
        private static extern bool QueryPerformanceFrequency(
            out long lpFrequency);

        private long startTime, stopTime;
        private long freq;

        // Constructor
        public HiPerfTimer()
        {
            startTime = 0;
            stopTime = 0;

            if (QueryPerformanceFrequency(out freq) == false)
            {
                // high-performance counter not supported
                throw new Win32Exception();
            }
        }

        public void Start()
        {
            Thread.Sleep(0);

            QueryPerformanceCounter(out startTime);
        }

        public void Stop()
        {
            QueryPerformanceCounter(out stopTime);
        }
```

```
        public double Duration
        {
            get
            {
                return ((double)(stopTime - startTime) / (double)freq);
            }
        }
    }
}

using System;
using System.Diagnostics;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;

namespace simple_3D_1_
{
    public enum Log_Type
    {
        ERROR = 0,
        WARNING = 1,
        INFO = 2
    }

    class Logger
    {
                static protected bool  _active;

        public void Init()
        {
            _active = true;
#if WINDOWS

            StreamWriter textOut = new StreamWriter(new FileStream("simple_3D_log.txt",
FileMode.Create, FileAccess.Write));
            textOut.WriteLine("Log File");
            textOut.WriteLine("");
            textOut.WriteLine("Log           started           at           "           +
System.DateTime.Now.ToLongTimeString());
            textOut.Close();

#endif
        }

        public bool Active
        {
            get { return _active; }
            set { _active = value; }
        }

        public void Log(Log_Type type, string text)
        {
#if WINDOWS
            if (!_active) return;
            string begin = " ";
            switch (type)
            {
                case Log_Type.ERROR: begin = "Error: "; break;
                case Log_Type.INFO: begin = "Info: "; break;
                case Log_Type.WARNING: begin = "Warning: "; break;
            }
            text = begin + System.DateTime.Now.ToLongTimeString() + " : " + text;
            Output(text);
#endif
```

```
        }

        private void Output(string text)
        {
#if WINDOWS
            try
            {
                StreamWriter        textOut        =        new        StreamWriter(new
FileStream("simple_3D_log.txt", FileMode.Append, FileAccess.Write));
                textOut.WriteLine(text);
                textOut.WriteLine(textOut.NewLine);
                textOut.Close();
            }
            catch (System.Exception e)
            {
                string error = e.Message;
            }

#endif
        }
    }
}
```

This provides the conclusion to the code overview of the core functionality of the demo application developed for the purpose of obtaining the results for this dissertation. Please refer to the project source code files for further info.

References & Bibliography

Baraff, D. 1997. *An Introduction to Physically Based Modeling: Rigid Body Simulation 1 – Unconstrained Rigid Body Dynamics.* Available online at: www.cs.cmu.edu/~baraff/pbm/pbm.html.

Bender et al. 2012. *Interactive Simulation of Rigid Body Dynamics in Computer Graphics.* Available online at: http://image.diku.dk/kenny/download/bender.erleben.ea12.pdf

Bourg M, D. 2001. *Physics for Game Developers.* O'Reilly.

Conger, D. 2004. *Physics Modeling for Game Programmers.* Premier Press.

Croft. A, Davison. R. 2006. *Mathematics for Engineers: A Modern Interactive Approach, Third Edition.* Prentice Hall.

DeLoura et al. 2000. *Game Programming Gems*. Charles River Media.

Eberly, H. D. 2010. *Game Physics, Second Edition.* Morgan Kaufmann

Garity, D. 1996. *Web Study Guide for Vector Calculus.* Available online at: http://www.math.oregonstate.edu/home/programs/undergrad/CalculusQuestStudyGuides/vcalc/vcalc.html

Gray et al. 2007. *Improving a 3D Rigid Body Simulator*. Available online at: http://thanosmich.com/files/cactus/cactus_report.pdf

Kreyszig, E. 2006. *Advanced Engineering Mathematics, 9th Edition.* Wiley

Lengyel, E et al. 2011. *Game Engine Gems 1.* Jones and Bartlett.

McShaffry et al. 2009. *Game Coding Complete, 3rd Edition.* Delmar.

Mendre, I. 2008. *Rigid body dynamics using Euler's equations, Runge-Kutta and quaternions.* Available online at: www.mare.ee/indrek/varphi/vardyn.pdf

Meriam, J. L. Kraige, L. G. 2008. *Engineering Mechanics: Dynamics, 6th Edition.* Wiley.

Moler, C. 2004. Numerical Computing with MATLAB. SIAM.

Press et al. 1992. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press.

Süli, E. 2010. *Numerical Solution of Ordinary Differential Equations.* Available online at: http://people.maths.ox.ac.uk/suli/nsodes.pdf.

Van Verth, M. J. Bishop, M. L. 2008. *Essential Mathematics for Games & Interactive Applications: A Programmer's Guide, Second Edition.* Morgan Kaufmann.

Additional Online References

[1] Willington, R. 2010. *AMD Aims To Merge GPU/CPU With Fusion APU.* Available online at: http://hothardware.com/News/AMD-Aims-To-Merge-GPUCPU-With-Fusion-APU/

[2] Patrizio, A. 2009. *Intel Merges GPU, CPU for Netbooks.* Available online at: http://www.internetnews.com/hardware/article.php/3854621/Intel+Merges+GPU+CPU+for+Netbooks.htm

[3] Freeman, V. 2010. *AMD Intel Race To Merge CPU and GPU.* Available online at: http://www.hardwarecentral.com/features/article.php/54011_3882846_/AMD-Intel-Race-To-Merge-CPU-and-GPU.htm

[4] Anderson, T. 2011. *NVIDIA plans to merge CPU and GPU – eventually.* Available online at: http://www.itwriting.com/blog/5239-nvidia-plans-to-merge-cpu-and-gpu-eventually.html

[5] Nvidia. 2012. *Nvidia Cuda Overview*. Available online at: http://www.nvidia.com/object/cuda_home_new.html.

[6] Khronos Group. 2012. OpenCL - *The open standard for parallel programming of heterogeneous systems.* Available online at: http://www.khronos.org/opencl/

[7] Griffith, S. 2008. *Digital Molecular Matter: Realistic material damage for military training simulations using real-time Finite Element Analysis*. Available online at: http://mil-embedded.com/articles/digital-finite-element-analysis/

[8] Pixelux. 2012. *Pixelux Homepage*. Available online at: http://www.pixelux.com/

[9] Pixelux. 2012. *DMM Engine*. Available online at: http://www.pixelux.com/DMMengine.html

[10] Sohal, V. 2008. *Military Simulations Get Real*. Available online at: http://www.cotsjournalonline.com/articles/view/100781

[11] Nvidia. 2012. PhysX Developer Zone. Available online at: http://developer.nvidia.com/physx

[12] Game Physics Simulation. 2012. Bullet Physics Engine. Available online at: http://bulletphysics.org/wordpress/

[13] Microsoft. 2012. *XNA Developrs Hub*. Available online at: http://create.msdn.com/en-us/education/gamedevelopment

[14] XNA Wiki Community. 2012. *XNA Wiki Tutorials*. Avaialble online at: http://www.xnawiki.com/index.php/Main_Page

[15] Strigl. D. 2002. *High-Performance Timer in C#*. Available online at: http://www.codeproject.com/Articles/2635/High-Performance-Timer-in-C