Pós-Graduação em Ciência da Computação

MARCO TÚLIO CARACIOLO FERREIRA ALBUQUERQUE

Dunas: A framework for Deformable Bézier Surfaces and Curves

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

Recife
2008

MARCO TÚLIO CARACIOLO FERREIRA ALBUQUERQUE

Dunas: A framework for Deformable Bézier Surfaces and Curves

Dissertação de Mestrado apresentado a Pós-graduação em Ciências da Computação, como parte dos requisitos necessários à obtenção do título de Mestre em Ciências da Computação.

Orientador: Silvio de Barros Melo
Coorientador: Co-orientador

Recife
2008

Catalogação na fonte
Bibliotecário Jefferson Luiz Alves Nazareno CRB 4-1758

A345d Albuquerque, Marco Túlio Caraciolo Ferreira.
Dunas: a framework for deformable bézier surfaces and curves / Marco Túlio Caraciolo Ferreira Albuquerque. – 2008.
86f.: fig., tab.

Orientador: Silvio Barros Melo.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn. Ciência da Computação, Recife, 2008.
Inclui referências e apêndice.

1. Computação gráfica. 2. Jogos por computador. 3. Modelagem física. I. Melo, Silvio Barros. (Orientador). II. Titulo.

794.8 CDD (22. ed.) UFPE-MEI 2018-09

Dissertação de Mestrado apresentada por **Marco Túlio Caraciolo Ferreira Albuquerque** à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título "**Dunas: A framework for Deformable Bézier Surfaces and Curves",** orientada pelo **Prof. Silvio de Barros Melo** e aprovada pela Banca Examinadora formada pelos professores:

_____

Prof. Marcelo Walter
Centro de Informática / UFPE

_____

Prof. Emerson Alexandre de Oliveira Lima
Departamento de Informática e Estatística / UNICAP

_____

Prof. Silvio de Barros Melo
Centro de Informática / UFPE

Visto e permitida a impressão.
Recife, 25 de agosto de 2008.

_____

Prof. Francisco de Assis Tenório de Carvalho
Coordenador da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.

Dedico esta dissertação, primeiramente a meus pais que me deram todo o apoio e incentivo possíveis, não só durante este trabalho, mas durante toda a minha vida. Ao Centro de Informática que frequento há oito anos, por toda a infraestrutura e suporte oferecidos. E, por fim, ao Professor Doutor Silvio de Barro Melo cuja valiosa orientação tornou possível a concretização deste trabalho.

AGRADECIMENTOS

Primeiramente a minha mãe, Ligia por ter sido sempre tão compreensiva e dado todo tipo de apoio necessário durante todos esses anos. Agradecer também ao meu pai, Marco Túlio, por criar as todas as oportunidades que tive em minha vida e pela preocupação com meu futuro. Sem esquecer os meus irmãos, Adriana e Bruno por estarem sempre ao meu lado.

À Carol França, amiga e namorada por seis anos. Sempre companheira e que sempre falou o que eu precisava ouvir sem se importar como eu iria reagir. Sempre entendeu, mais do que a maioria, os horários e a dificuldade de estudar no Centro de Informática. Além de ter revisado, jornalista que é, diversos relatórios, textos e artigos meus.

Agradeço aos Joeys: Felipe, Thiago e Duda. Sempre foram melhor forma de diversão e distração de todo tipo de problema que enfrentei, de sempre trazerem alegria e serem amigos em todos os momentos, bons ou ruins. Enfim, por terem dividido todas as experiências que vivemos e rachado todos aqueles churrasquinhos.

A todos os meus amigos de faculdade, principalmente os colegas mais próximos, o grupo de projetos: André, Afonso, Vicente, Cabelinho, Vilmar e Ives. Pois afinal foram as pessoas que estavam comigo nas aulas, nos projetos, nas madrugadas sem dormir, nos estágios, convivendo quase que diariamente durante toda a graduação. Agradecer especialmente a Vicente, amigo desde os tempos de colégio e sócio a mais de três anos que não só apoiou o trabalho como ajudou em diversos momentos na criação do mesmo.

Ao professor e amigo Silvio Melo pelo suporte. Por estar sempre disposto a ajudar no que for preciso, mesmo quando ainda fazia parte do Departamento de Matemática. Além de suprir a necessidade de um orientador na área da computação gráfica deficiente no centro em particular e no estado como um todo.

Gostaria de agradecer a diversos professores do Centro de Informática, especialmente Chico, George e Geber. Pela formação recebida e pela orientação nos diversos projetos durante cadeiras na graduação e na pós e também iniciação científica.

Gostaria de agradecer a todos aqueles que participaram de forma positiva na minha formação. E todos que fizeram parte da minha vida no CPI, no Centro de Informática, no CESAR, na Manifesto e na Jynx pela amizade e companheirismo. Desses gostaria de destacar a especialmente Carol Gondim, grande amiga, principalmente nesses últimos meses de trabalho quando deu o apoio necessário em tempos bastante complicados.

RESUMO

Os poderosos processadores da nova linha de computadores e consoles de vídeo games fazem da utilização de superfícies de Bézier uma boa idéia devido a sua potência de processamento, no entanto, a criação de modelos com superfícies de Bézier ainda parece ser muito mais do campo de projeto auxiliado por computador e manufatura auxiliada por computador do que no campo de desenvolvimento de jogos segundo Dave Eberly. Esse trabalho tenciona popularizar o uso das técnicas de modelagem com curvas e superfícies de Bézier. Para isso, introduziremos técnicas de modelagem física para a simulação de corpos deformáveis com o intuito de automatizar a animação de tais superfícies popularizando a abordagem no meio dos desenvolvedores de jogos. O método criado torna possível a utilização de modelagem física em plataformas de baixo poder computacional, como Flash e J2ME, e ao mesmo tempo possibilita a utilização de cenas mais complexas em plataformas mais poderosas que utilizam placas gráficas dedicadas.

Palavras-chave: Computação Gráfica. Jogos de Computador. Modelagem Física. Curvas de Bézier. Superfícies de Bézier.

ABSTRACT

Powerful graphics processors on personal computers and game consoles make surface representation a good choice due to their processing capabilities. However, creating surface models still appears to be in the realm of the CAD/CAM and not game development acoording to Dave Eberly. This work unify both fields, physics and geometry, exploring the interface between geometry and physically based animation by incorporating well established physic models into Bézier surfaces for the simulation of deformable objects. By doing so, we create a simple yet powerful representation of the model, with Bézier Curves and Surfaces, and incorporate some of the most used methods of automated shape changing animation. Our approach makes possible to use real physics simulations in limited platforms, like Desktop Flash and mobile J2ME. At the same time, enables the use of even more complicated scenes on powerful platforms that makes use of dedicated graphics cards like Desktop Personal Computers.

Keywords: Computer Graphics. Computer Games,.Physically Based Modeling. Bézier Curves. Bézier Surfaces.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

2D	Two dimensions
3D	Three dimensions
CAD/CAM	Computer Aided Design/ Computer Aided Manufacturing
CPU	Central Processing Unit
FEM	Finite Element Methods
LOD	Level of Detail
PC	Personal Computer
RAM	Random Access Memory
UML	Unified Modeling Language)
XML	EXtensible Markup Language

SUMÁRIO

1 INTRODUCTION

This chapter presents the main motivation for this work, enumerates a list of research goals, does a little introduction to the subjects involved and gives a preview of the whole document and how it is organized.

1.1 Motivation

Historically, deformable models appeared in computer graphics to create and edit complex curves, surfaces and solids [1]. Today, deformable bodies are used in real time character animation, especially for computer games, for the realistic simulation of skin [2], clothing [3], Figure 1, and human or animal characters. Generally speaking, existing modeling approaches can be categorized, by technique, into one of two major groups: physically-based techniques and purely geometric techniques [12]. Models based on solving continuum mechanics problem, under consideration of material properties and other environmental constraints, are called physically-based techniques. Non-physical techniques are purely geometric techniques used to deform visual objects. Each of the techniques has advantages that could be strengthened and disadvantages that could be weakened by mixing the methods in a smart way.

Figura 1 – Cloth Simulation using string physics

Source: [3]

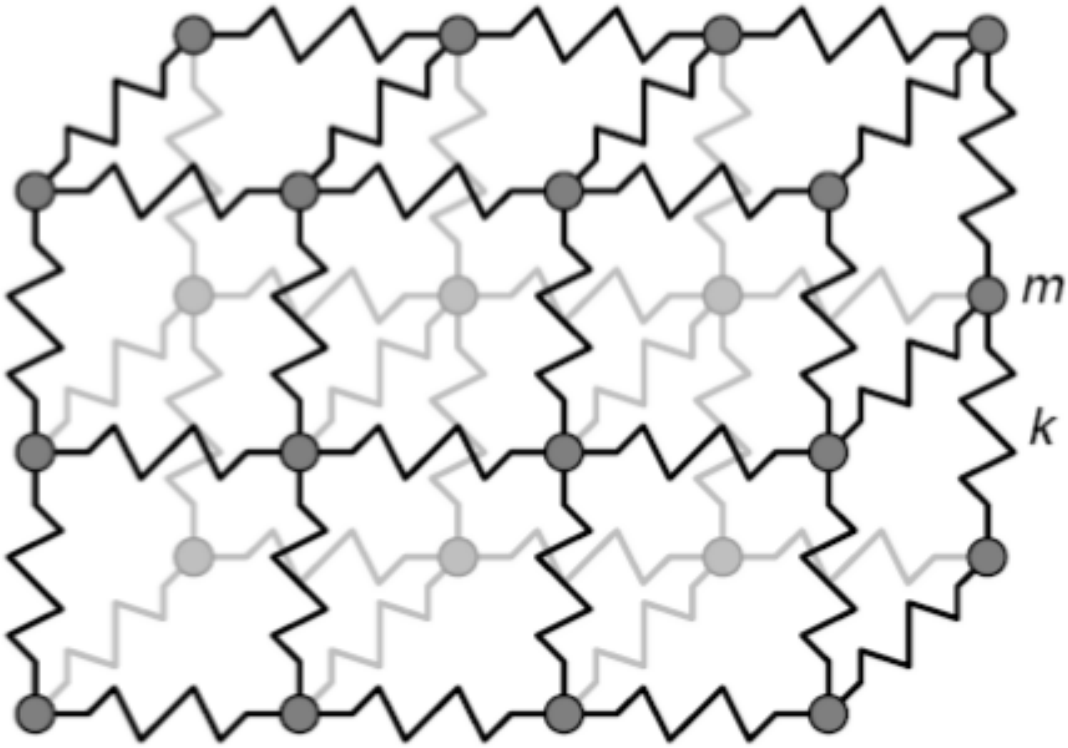
To explore the interface between geometry and physically based animation, in a smart way, we created the Dunas Framework that diminishes the processing needed by physical methods by using geometric methods. Dunas incorporates a well established

physics model, mass-spring, into Bézier curves and surfaces [11] for the simulation of deformable objects in real-time interactive applications, such as video games. The whole work consists of how these two, at a first glance, antagonic methods can be mixed together to create a powerful representation of deformable models that can be effectively used in different scenarios of realtime applications.

Purely geometric deformation techniques are faster and are generally simpler to implement than their physically based competitors. In such cases, physical accuracy is sacrificed for computational efficiency and the system has no knowledge about the material of the object being deformed. Usually, these techniques are computationally efficient, but rely on the skill of a designer rather than on real physics principles. Created by the French engineer Paul Bézier, Bézier curves and surfaces are among the most popular and intuitive geometric methods for deformable models, a whole chapter is dedicated to the Bézier theory on this work as an introductory text.

Sophisticated physically based models, although necessary for simulating the dynamics of realistic interactions, are not yet suitable for fully interactive real-time simulation of multiple objects in virtual environments due to the current limitations of computational power. MassSpring Damper systems (MSD) appear to be the most widely used deformable models at present. It is likely that their popularity originally stemmed from the simplicity of obtaining equations and programming them. MSD models thus evolved rapidly and implementations for nearly every conceivable type of interaction have been developed. This, unsurprisingly, only increased their popularity. The main advantage of the MSD model over its competitors today is its ability to approximate physical realism at real-time rates. Using MSD, an object is modeled as a collection of point masses, called particles, connected by springs in a lattice structure - Figure 2.

Figura 2 – A portion of a mass-spring model



Source: [12]

Dunas dramatically reduces the number of particles used in the simulation of mass-spring systems to improve performance. However, lowering the number of particles by itself might lead to models with a very few number of polygons that, when rendered, leads to very poor images for modern computer games standards. The use of Bézier models are especially motivated by this fact since it can generate smooth curves and surfaces with an arbitrary level of detail, thus generating smoother models with any number of polygons as desired - rendering the best model possible for a given platform.

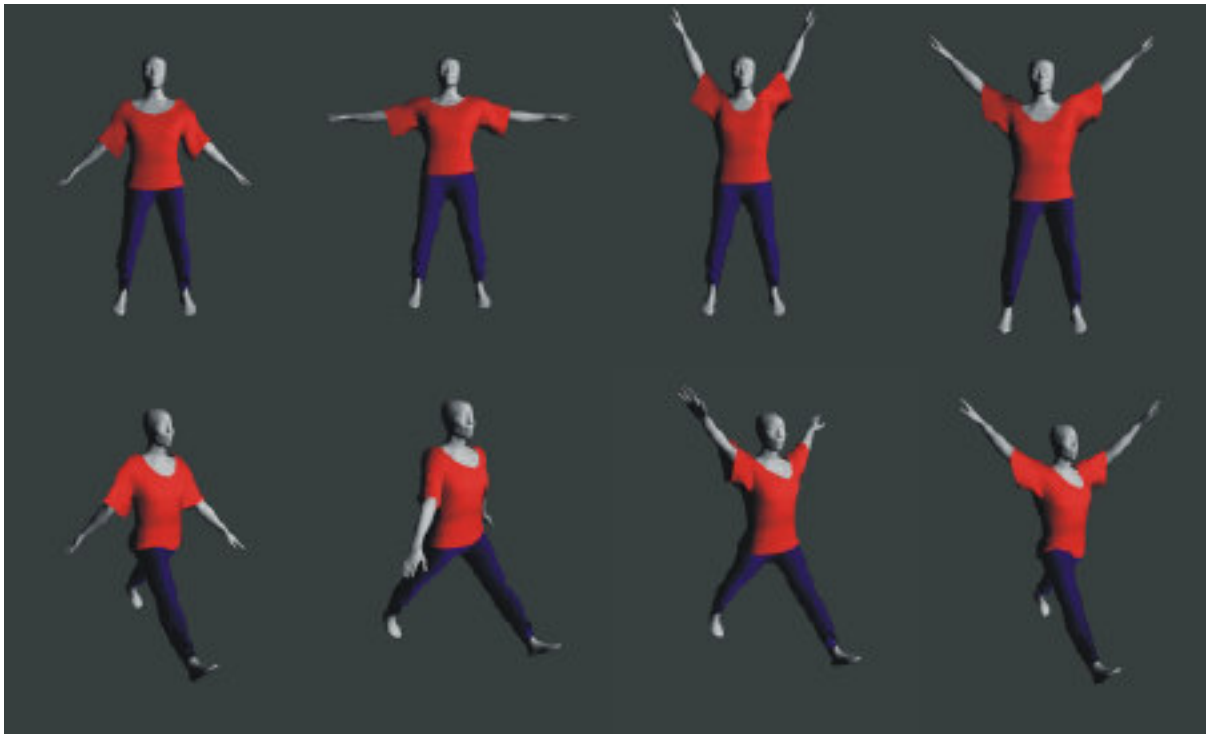
Despite the popularization of deformation methods in the academic field, very little is being used in commercial projects, especially casual games. Casual games are targeted to ordinary people that usually don't have a powerful computer with dedicated graphics card. With this in mind, we developed a mechanism that can bring more realism to this kind of game by using a simplified yet powerful approach to model mass-spring deformable bodies and at the same time still benefits from high-end machines and is used on hardcore computer games.

The trial and buy business model is currently used by the most popular casual game portals around the web [13][14][15]. In this model, the user can play the game for a limited period of time and then the game is locked until the user pays for it. Usually the user isn't willing to download a large piece of software to install. So, a web version of the game, developed in the Flash platform [16], is usually created to give him a taste

of the game. This way, the Flash version of the game must maintain the most crucial game features so the user can fully experience the gameplay involved.

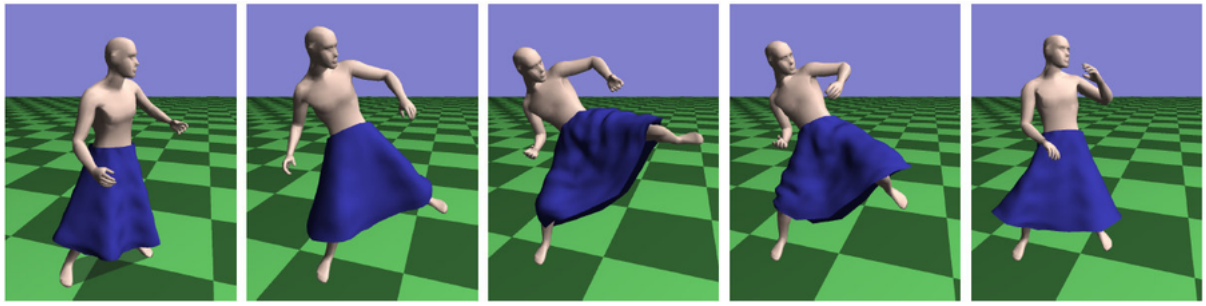
Besides, some of the articles in the field that claims to be real-time can't be used in commercial products. Today's games have a high polygon budget and most of the images shown in these papers are toy problems. For instance, most of the scenes presented on articles [17] from the field trespass the real-time frame per second barrier (at least 30) with a small margin, not more than 32 frames per second. However, the scenes are very simple with no more than one piece of deformable body being simulated at real-time like in the Figure 3 and Figure 4. A better configuration is achieved by Dunas, since it was projected to beat these numbers, as shown in the last chapter of this document.

Figura 3 – Single Model



Source: [18]

Figura 4 – Single piece of cloth



Source: [17]

1.2 Goals and Contributions

The main purpose of this research is to create a simple deformable simulation model, and framework that supports the use of this model, which can be used in interactive real-time applications for limited platforms, such as Flash and J2ME[20], and at the same time be scalable enough to take advantage of cutting edge graphics technology. Such a goal is made concrete by some tasks, including:

- Research of deformable bodies that uses physics methods and could benefit from the mixture with a geometrical method, like cloth;
- Design and development of a physics framework that supports the most common formulation of the mass-spring system for real-time simulation of deformable models;
- Efficient implementation of the Bézier algorithms to create smooth curves and surfaces from a set of points that can be effectively used in real-time commercial applications;
- Integration of both methods in a pipeline that simulate a deformable model using physics rules and Bézier models;
- Evaluation of the framework created regarding the quality of the generated images, if they are visually good enough for today's standards, and at the same time giving the required performance for real-time applications in platforms of low processing power such as Flash and J2ME
- Improvement of the current state of the art by incorporating a new hybrid method for the simulation deformable models at real-time frame rates even in limited platforms.

Each of these tasks was made in this work and is presented in the appropriate context in the following chapters. In summary, we ended up developing a framework that uses mass-spring system to simulate real physics for deformable bodies. However, we can drop the number of particles, or masses, needed on the simulation by smoothing the resulting model using the deCasteljau algorithm at the price of a small penalty on the reality of the physics simulation. In other words, we use a Bézier model in which the control points are not fixed but a function of time that changes the control points based dynamics theory according to the external forces acting on them.

1.3 Challenges

The cloth simulation in real-time has to meet several requirements that greatly limit the development or even the existence of such systems [18]. Among these, we could mention stability, robustness and, of course, speed. These same requirements hold true for all kinds of physically simulation of deformable models.

Stability is needed to deal with the fact that whatever are the conditions present during the simulation the system must behave in a correct and predictable manner. Unnatural or erratic model appearance and movement must be avoided at all costs. In other words, we need to use the simplest model possible and avoid numerical instability.

Robustness is an even more difficult requirement, especially when the model is placed in an interactive environment, i.e. a virtual reality application or a video game, where interaction or the fast-changing conditions must be handled properly.

Speed is obviously the most important aspect of any real time system. There are two major obstacles that make these requirements difficult to fulfill: the model itself and the detection of collisions. Most highly accurate physically based simulation systems are just too slow to be used in interactive or real time applications. They are computationally heavy and difficult to optimize to a level that would permit them to be significantly accelerated. This is especially important for Dunas.

Adding to the challenges of all deformable model articles, in this work the biggest challenge is to create a way of using deformable bodies in 2D computer games for low processing power platforms, like Flash and J2ME, and at the same time the technique must provide good visual results to be used in good machines with dedicated graphics cards. From one side, we need to create a simple and fast method that can be used in this kind of platform and on the other side it must be scalable, in terms of performance and the realism of the simulation, to create good images on better computers.

1.4 Dissertation Organization

The contents of this document are organized in six chapters. Chapter 2 discusses the state of the art on deformable model with an emphasis in the hybrid methods that mix geometric methods and physically-based models for deformable bodies. A discussion about the recent use of massspring system to model ropes and cloth models is also provided.

Chapter 3 presents an introductory text on geometric methods for deformable bodies with a special emphasis on the Bézier theory.

Chapter 4 details the Dunas framework architecture using the Unified Modeling Language (UML) [21] diagrams, the process involved in creating the software component

and the design decisions that were made along the development.

Chapter 5 presents two complete case studies developed using the proposed software component and compares the results obtained with the state of art techniques for the simulation of mass-spring systems. At the end of the chapter there is a section discussing the results and suggests future works.

2 STATE OF THE ART

This chapter introduces the physically-based modeling of deformable bodies' field and how it is used in modern computer games. Three of the most popular physics simulation methods are explained and the numerical integration methods used.

2.1 Deformable Bodies

Physics-based animation is a highly-interdisciplinary field, which is based on theories from engineering, from physics, and from mathematics. The field of physics-based animation was first named in a course in the 1987 ACM SIGGRAPH (the Association for Computing Machinery's Special Interest Group on Computer Graphics) conference, "Topics in Physically-Based Modeling" organized by Alan H. Barr. In recent years the emphasis in physics-based animation on computationally efficient algorithms has spawned a new field: plausible simulation.

The plausible simulation term is a primary concept in our work since we are trying to achieve a result that looks good visually but not necessarily obeying any kind of physical model based on the real phenomena.

Deformable Objects, or Bodies, covers all object types that are not rigid, and thus span a wide range of different object types with different physical properties, such as jelly, cloth, biological tissue, water, wind, and fire. Our work is concerned with certain types of deformable objects simulations that would benefit from being modeled as Bézier curves or surfaces such as ropes, clothes, water or terrain. Our basic approach is using the physics simulation techniques on the control points of the curve or surface rather than making them act on the object points. This approach, if it yields good visual results, drops the amount of computation required to achieve the same results and yet uses a fixed amount of computation even using dynamic LOD techniques, because the number of control points is fixed at build time.

With that in mind, this chapter presents what we consider to be the state of art on the field of deformable bodies that has a direct impact on our work. The following section presents the field of deformable bodies and where this work is placed among others.

The deformable model field is basically divided into two major groups [15] called the physics-based technique and geometric methods. Mass-spring system is the most popular physically-based deformation technique for real-time applications. On the other hand, Bézier models are one of the most intuitive and popular method for modeling deformation on curves and surfaces.

The basic idea behind our technique is to make the control points of a Bézier curve or surface a function of time updating the point position based on physics rules

according to the external forces acting on the model. This approach drop the amount of computation required by traditional methods to achieve similar results, because we update only the control points instead of object points, and yet uses a fixed amount of computation even using dynamic Level of Details techniques, because the number of control points is fixed at build time and the degree of evaluation can go until the desired resolution is achieved. The use of physics rules is to automate the animation process, usually done by hand which requires a skilled and experienced artist, and at the same time get realistic, or at least plausible, results.

An important fact to note is that these hybrid methods, despite the fact that some good articles can be found in the literature, aren't yet fully explored neither popular in the physics simulation field. Also, most of it is used to improve the interaction of the artist during the modeling phase and not for simulation purposes. This is easily noticed since most of the surveys on the field [12][23][28] don't even mention any of these methods and explicitly split the area in two big groups: geometric methods and physically-based methods. However, the improvement in performance they can achieve, with a small degrade on quality, makes them a reasonable solution for computer games since we are not trying to get realistic results but plausible results that can provide a good experience for the user while playing the game for a cheaper computational cost.

In this chapter, we present some methods that make the combination of geometric and physically-based models showing how they approach the problem and how our technique is based on them and also how our approach differs from theirs. Then we show popular techniques used to simulate rope and cloth with a physically-based approach that is modified in the last chapter, to use our method, and show how it can benefit from a mix with geometrical model, in this case Bézier curves for the rope and Bézier surfaces for the cloth.

2.2 Hybrid Methods

Many methods combines dynamics simulations and geometric techniques for deformable bodies to reduce the number of particles involved in simulation thus creating a new technique that can benefit from both, the realistic models from physically-based modeling, like mass-spring systems, and the high computational performance of geometrical methods like Free-Form Deformations (FFD)[26] and B-Splines[11]. Some of these methods and the pros and cons of each are discussed in this section providing the motivations that made us take a different path using a bit of each. The first two articles aim to create a better interaction process so the designer can have more flexibility and better feedback while creating a model, specially animating that model and despite the fact that they try to mix physically-based method with geometric models, they are not actually related to our work.

Before hand we must say that most of these methods aim an accurate model, instead of a good trade-off between performance and accuracy, which dramatically reduces the performance of such methods. Our goal is to create a powerful model that can at the same time be used in platforms with low computational power, like Flash[16] and J2ME[20], and at the same time sufficiently scalable to benefit from high-end machines like modern Personal Computers with dedicated graphics card and video game consoles like XboX360[7] and PlayStation 3[8]. Besides that, as we discuss along this chapter, some of the articles claim to get real-time frame rates. However, the scenes they use as benchmark are pretty simple and not suitable for a commercial game with a limited triangles budget.

Since there are very few articles using hybrids approaches, the following pages discuss each one of them individually and how they affect our job. As said in the introduction of this chapter, [12][23][28] gives no information about hybrids methods, a sign that, although very effective in some applications these approaches to the field are not explored as it should be. Unfortunately, this is not an isolated case since [24], also talks about the benefits from using machine learning in computer graphics that is little explored because very few people actually have the proper contact with both fields to fill the gap - physicists usually don't bother with computer tricks to generated plausible simulations and computer scientist don't have the required mathematical background to actually understand the simulations.

D-NURBS [25] is the first effort to combine geometric and physic methods. D-NURBS are physics based models that incorporate mass distributions, internal deformations energies, forces, and other physical quantities into the NURBS geometric substrate. The intent is to create an interactive process to sculpt NURBS much in the same way artists do with mud. The process is intended to be fast and accurate but not necessarily real-time. Again, their objective is highly different from ours since they aim to improve the creation process of a model and not animating it during real-time using physically plausible methods.

The basic idea, same as we use, is to make the control points a function of time instead of a fixed position tupple. The function updates the control point's position according to the laws of dynamics and the external forces acting on them. The Finite Element Method (FEM) [27] technique is used to update the control points on four different kinds of NURBS.

Using D-NURBS, during the interactive process, the user must define a large set of parameters that fits their need of interaction. This conflict can lead the designer to use parameters that favors the interaction over realistic properties of the real object being modeled. Which is good for their purposes since would make the creating process easier and more straightforward. However, our model tries to create a plausible approach for

simulations purposes in real-time during runtime and these feature wouldn't give us better results. It's important to notice that both works could be used together if the interaction process is used in a Bézier model instead of a NURBS. Then during runtime, our method could apply physics rules to change the model according to the external forces being applied on it.

A curious point in the article it's their comment on the idiosyncrasy generated by using negative weights. This reveals the author's lack of familiarity with the theory of NURBS since this can be easily explained using the projective geometry [11]. This fact also reveals that the deformable models field could benefit from works like ours with people that have experience with both of the methods, since Terzopolous, which coined the term deformable bodies and is recognized as one of the major researchers in the area, coming from a physics background shows very basic knowledge about geometric methods. This fact shows how [24] affects the deformable bodies field much in the same way it does for computer graphics and machine learning.

NURBS have a mathematical theory more flexible, thus much more complex, then Bézier models. This might be a good property in some applications since it gives more flexibility of manipulation and therefore the ability to create better models. However, in game companies artists are not math experts and sometimes don't even have a good educational formation to actually comprehend the mathematical background involved in NURBS. Bézier models, on the other hand, are very intuitive and the deCasteljau algorithm have a geometric representation [11] that is easy to understand. A simple pool, but that gives a feeling of what we are saying, made with designers working in some of the companies in Recife is shown in Appendix A of this document. The main information we can retrieve is that most designers feel more comfortable using Bézier models then NURBS when both are available.

D-NURBS uses to calculate the change in positions of the particle, the FEM-based method. FEM is very accurate and used in lots of realistic simulations from car security systems [30] to reconstruction of human faces[29]. However, the computational cost involved makes it prohibitive for real-time interactive applications targeted to platforms like Flash and J2ME and still very limited applications on personal computers and game consoles. Thus far, FEM techniques have proven to be impractical for real-time applications [28].

Compared to this D-NURBS's project, we take a more simplified approach using techniques that requires less computational power: mass-spring system, as the dynamic model, and Bézier curves and surfaces instead of NURBS for the geometric model. The use of rational surfaces instead of polynomials, our case, reveals their concern with a realistic representation of certain kinds of surfaces, such as quadrics and revolution surfaces, which is not the case for us that are looking for physically plausible methods

especially for ropes and clothes.

Yet, dedicated graphics cards implement interpolation schemes, required by OpenGL[9], ready to create Bézier curves and surfaces, which improves the performance of these methods and makes it extremely simple to implement compared to NURBS. This usually isn't an issue in an academic research. However, in commercial products, the time to implement a new feature is extremely crucial since it can dramatically increase the time to market of the game - designers that knows how to model NURBS, programmers that knows NURBS theory and can implement it efficiently, etc.

Some other methods followed D-NURBS [31] but all of them aiming the same goal, to provide more flexibility while designing a model using NURBS and never actually a framework to simulate the deformations of an object. In [32] a method using Splines and FEM-based methods is presented but also "to improve the efficiency of interactive design".

Dunas uses Bézier as the geometric methods in the hybrid approach. A common question is why use Bézier instead of NURBS and the answer is very simple: Bézier is good enough to solve the problem being much simpler to implement, understand and use it. At the same time, Bézier is a simplification of NURBS since it is a NURBS with weights equals one. That being said, every successful application of Bézier is also a successful application of NURBS and can be easily extended to be tested with other weights. So we are using the approach that is simple as possible, however not simpler than required by the problem.

The most effective and one of the most recent works on the use of geometric methods and physically-based techniques to give proper feedback in haptic interfaces[33] . Adding haptic feedback in deformable objects is attractive in many applications, such as computer games, interactive cartoon design, and virtual prototyping. This paper, proposes an interactive haptic deformation approach incorporating the dynamic simulation of mass-spring systems and flexible control of free-form deformation in the touch-enabled modeling of soft-object deformation.

Just as D-NURBS, this works focus on giving more flexibility during the creation process of a model. There is no concern at all on the performance of the method or how it could be used to handle real-time simulations, such as a game. Imagine that you can create a fixed preanimated response to a certain force - like wind. However, if you change the force in real-time during the simulation this new effects won't be taken into account. For collision detection response this is especially awkward, since none of the collisions wouldn´t be treated effectively.

There aren't a lot of projects using mass-spring systems as the physics simulation method when accuracy is necessary. However, it is the usual choice for interactive

applications and it is used in this work to create proper feedback for haptic input. The idea is create the correct behavior of a haptic input system to the user that is interacting with the applications.

The work uses a cubic Bézier, third degree patch, instead of curves or surfaces. Since they are providing feedback in a volumetric input device it makes a lot of sense. However, for our purposes it would be a waste of computation and non-intuitive since we would need to deform a whole volume to obtain an effect that visually affects only a surface or curve.

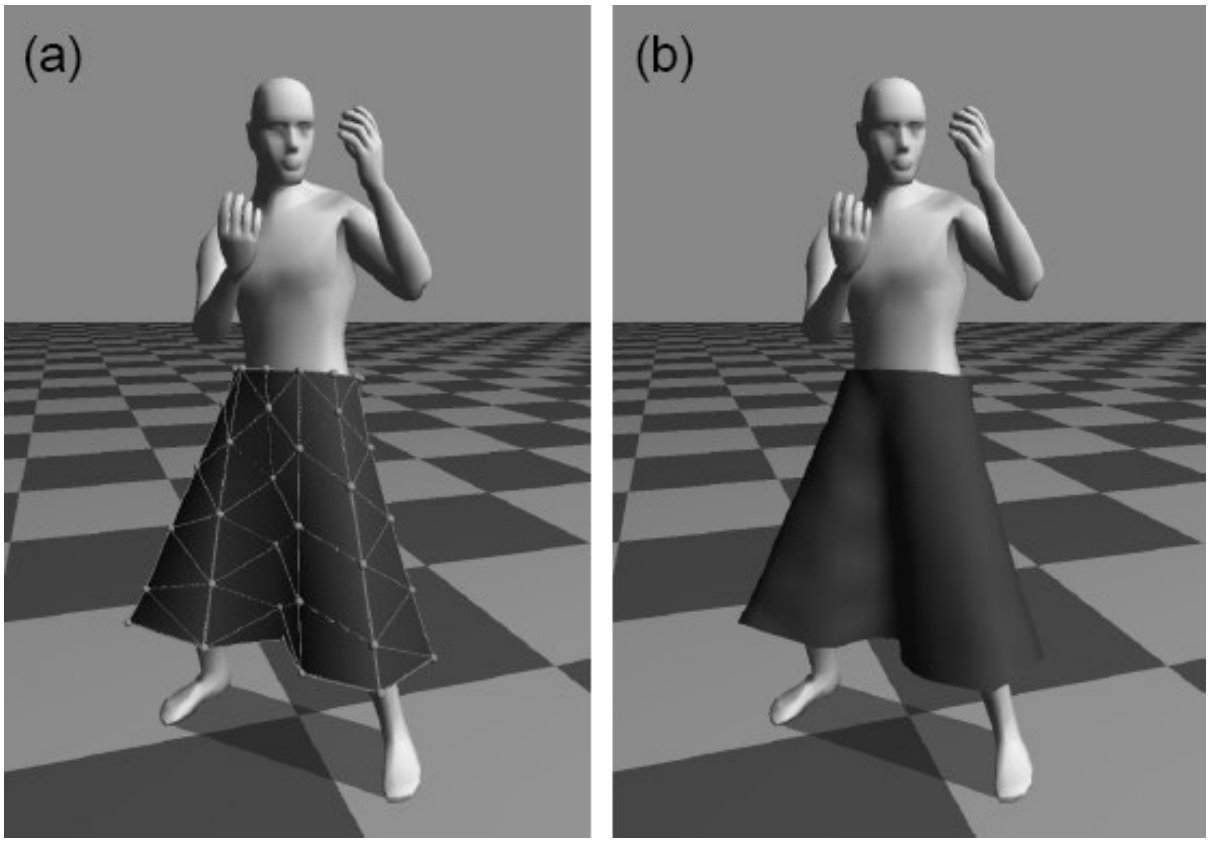
A technique for real-time cloth simulation combines dynamic simulation and geometric techniques [17] . Only a small number of particles (a few hundred at maximum) are controlled using dynamic simulation to simulate global cloth behaviors such as waving and bending.

Cloth can be simulated using mass-spring systems as shown in the last section of this chapter. A mass-spring system is based on a set of particles, or masses, that are connected using springs that obeys Hooke's law. The main problem with this approach is the huge number of vertices used to model a piece of cloth with the direct translation of a vertex to a particle. As our method, this one drops the number of particles used and creates the rendering model using a Bézier method translating control vertices in particles instead of mesh vertices. This gives us not just the low number of particle but also a fixed number of particles even when level of detail techniques aren't necessary. This is desirable because it's possible to know the number of particles in the simulation at compile-time. Using level of details technique and using the vertices as particles this is impossible since we can't foresee the number of particle that will be used in the scene for rendering.

Oshita makes use of "only a small number of particles (a few hundred at maximum) are controlled using dynamic simulation to simulate global cloth behaviors such as waving and bending. The cloth surface is then smoothed based on the elastic forces applied to each particle and the distance between each pair of adjacent particles". We take the same path lowering the number of particles needed for the simulation but we don't use PN Triangles [34] to smooth the surface.

The principle in this method is pretty much the same used in ours. However, instead of using the particles as control points of a Bézier patch, it uses PN Triangles to create the subdivision schema resulting in a smoother surface. Smoothing the surface is desirable since there is drastically reduction on the number of points in the model, eliminating some of the details. As shown in Figure 5, a cloth modeled using only 66 particles don't generate a satisfactory result. However, the same 66 particles smoothed, which leaded to 1944 faces rendered, yields better results.

Figura 5 – (a) shows the particles of the simulation. (b) shows the rendered skirt only



Source: [34]

The use of PN Triangles gives us the smoothing feature but on the other hand brings the problem of shortened edges. The problem is alleviated using more control points to increase the size of the spring. However, the analog problem of elongated edges is apparently solved dynamically. The problem is still open since both of the issues can't be totally solved by the techniques presented. However, the results are still very good visually as shown in Figure 4.

For Bézier tensor patches, as we use, we don't need to calculate the normal of each triangle for the smoothing process, which eliminates one of the biggest performance bottlenecks of this article. The usual way of calculating a triangle's normal, using the vector product of two edges of the triangle, is extremely inefficient and unsuitable for real-time applications. Due to this fact, they have a special, inaccurate method, to calculate the normal of each of the particles. That partially solves the problem since the normal is still needed to be calculated.

This method achieves very good results for cloth simulation, but it's still heavy for real commercial applications. For a single piece of cloth, in this case a skirt, they achieved in best cases a little more the 30 frames per second. A naive observer can think that it's an outstanding result since he can get really interesting animations, which meets today's standards for real-time simulation of deformable bodies, but the scene in

the paper - as shown in Figure 4 - is very simple with no texture and a very cheap triangle cost. So, as we can attest, in an application where you need to use let's say 10 pieces of cloth we'll get a lower frame rate which is unacceptable for games nowadays. Imagine a more complicated scene and besides the graphics we still need processing power for game logic's, artificial simulation for non-player characters and network processing [5][6].

Clothes can be animated, in real time, placed over an articulated character [18][19]. The articulated character is approximated using a hierarchy of ellipsoids. The cloth, as usual, is represented by a mass-spring particle system simulation. First the particles move following the ellipsoids; this is followed by the application of dynamic forces. Finally, penetration of the character's ellipsoids by any particle is corrected. This method is fast enough to deliver real-time performance on mid-range PC and workstations. However, the technique is limited to articulated characters and clothes. Dunas, on the other hand, can be used in any mass-spring system and even be used in this method to use fewer particles on the rope simulation and then smooth it using the deCasteljau algorithm. In fact, one of the improvements suggested in the future works of the paper is using smoothing techniques. The integration is straight-forward as we just need to use the deCasteljau algorithm to smooth the model created by the particles on the mass-spring system after the correction stage.

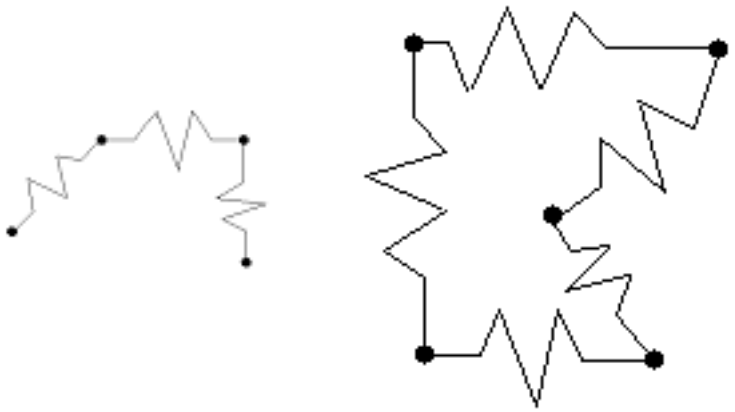
2.3 Mass-spring systems

Mass-spring systems can be divided into three subcategories: curve masses, surface masses and volume masses. This section discuss each of one separately since we will implement two samples of curve mass and one of surface masses.

2.3.1 Curve Masses

Curve Masses or deformable curves, is represented as a polyline of vertices, open with two end points or closed with no end points. Each vertex of the polyline represents a mass. Each edge represents a spring connecting the two masses at the end points of the edge. Figure 6 shows both configurations (open and closed).

Figura 6 – Two curve mass objects represented as mass-spring systems



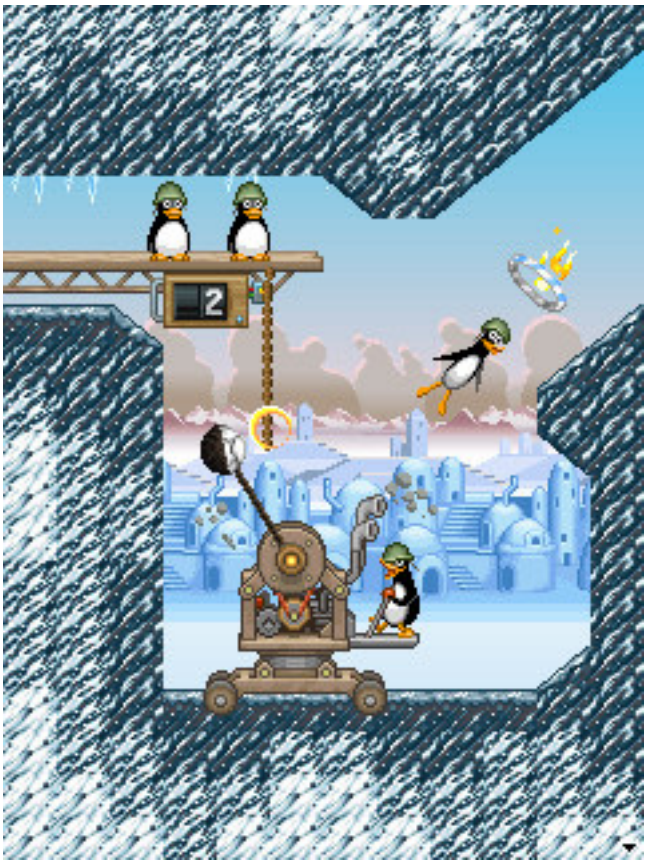
Source: The Author

This is the simplest topology of a mass-spring system. The masses mi are located at positions Xi for 1 i p; spring i connects mi and mi+1.At an interior point I, two spring forces are applied, one from the spring shared with point i-1 and one from the spring shared with point i+1.

Each spring can have its own spring constant and rest length. However, usually the whole chain represents a single object and it's desired that the whole set of spring presents the same properties. The spring constant may use real world object constants. But the rest length is usually set by an artist and exhaustively changed until the desired behavior is achieved. Note that is not unique to curve masses but for all mass-spring systems.

Figure 7 shows how a curve mass can be used as a rope on computer games. The Crazy Penguin Catapult from Digital Chocolate uses a rope to swing the penguin that is going to be catapulted to battle against the bears.

Figura 7 – Crazy Penguin Catapult game from Digital Chocolate [36]



Source: [35]

2.3.2 Surface Masses

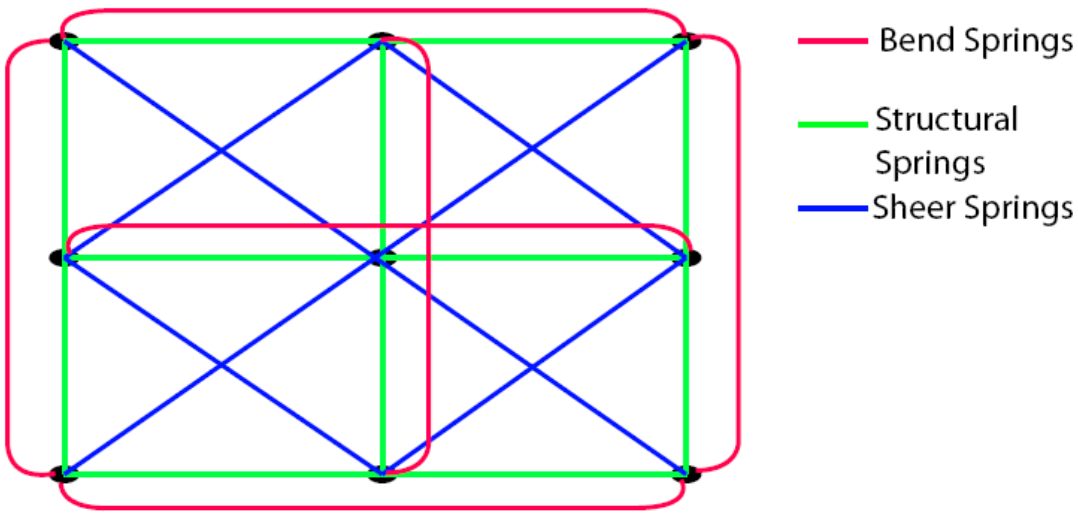
Classically, cloth was modeled by using mass-spring systems [37]. The cloth is modeled as a regular 2D grid where the grid nodes correspond to particles, or masses. The cloth can be animated by moving the particles, which will deform the regular grid. We are going to talk about cloth modeling, as presented in Figure 8, and may extend the same reasoning to other types of objects that are commonly modeled using the same 2D grid, such as water surfaces.

Figura 8 – Cloth Simulation using mass-spring system method

Source: [37]

Springs are not created at random between the cloth particles, but a certain structure is used to mimic cloth properties. Three types of springs are commonly used: structural springs, shearing springs and bending springs. These springs are illustrated in Figure 9.

Figura 9 – Types of spring in a cloth model



Source: The Author

Structural Springs are created between each pair of horizontal and vertical particle neighbors in the regular 2D springs. The purpose of structural springs is to resist stretching and compression of the cloth. This kind of spring is usually initialized with very high spring coefficients and with a rest length equal to the interparticle distance. Keep in mind that, not only for the structural spring but for all types, the values for spring coefficient and rest length have no real world meaning or counter-part. This is a huge simplification from the real cloth physics and the best numbers are the ones that give the best visual results.

Shearing springs are used to make sure that the cloth does not change shape. Spring coefficient for this type of spring is usually smaller than the ones for structural springs. Shearing springs are created between diagonal directions in the regular grid. Thus a particle at a grid location (i, j) will have shearing springs to particles at grid locations: (i+1, j+1), (i+1, j-1), (i-1, j+1) and (i-1, j-1).

Bending springs are inserted to make sure the cloth will not have sharp ridges when folded. These springs are usually not very strong, since natural cloth does not strongly resist bending. These springs are created along horizontal and vertical grid lines, but only between every second particle. Again, for a particle at a grid location (i, j) will have bending springs to particles at grid locations: (i+2, j), (i, j+2), (i-2, j) and (i, j-2).

Varying spring coefficients can be used to model different types of cloths [38]: cotton, polyester, etc. Mass-spring system can also be used to create cloth animation for realtime applications [39] like in Figure 10.

Figura 10 – Blue dress modeled using mass-spring systems

Source: [39]

The problem with this approach is that only rectangular pieces of cloth can be modeled efficiently – like flags. However, these concepts can also be used on non-regular grids with the generalizations made in [29], which can be applied to unstructured meshes such as arbitrary triangle meshes. Such as the blue dress in Figure 10.

The generalization is simple and is based that in an unstructured mesh we can define the type of connection between two points, or masses, based on their neighbor distance. Every edge in the mesh has a cost of 1 and the connection between two points is given by the number of edges in the shortest path between them, for instance 1-neighborhood for structural springs, 2neighborhood for shear and bending springs.

Generally speaking, the larger the neighborhood size used, the more rigid the resulting object appears to be. In fact, if the neighborhood size is large enough, springs will be created between every pair of particles in the mesh, and if the spring constants are large, then the object will appear completely rigid. Thus we refer to the neighborhood size as the rigidity factor. The spring creation method was named surface springs in [29].

This type of representation is particularly good to use with Bézier surfaces. Rectangular grids of control points are widely used to represent all kinds of objects, especially on terrain modeling.

2.3.3 Volume Masses

All the ideas presented before can be extended from surfaces to solids. For instance, given a 3D rectilinear grid, structural, shearing, and bending springs can be extended straightforwardly to the 3D case. However, one needs to consider spatial diagonals as well as shearing diagonals. Shearing diagonals will prevent area loss; spatial diagonals will counterattack volume loss. Keep in mind that the objects have a rest state they try to preserve; analog to Newton's second law of dynamics and any disturbance on this state will create internal forces to annulated it.

Unfortunately, in practice a rectilinear grid or a volume mesh of the solid is not always available, but instead a 2D surface mesh of the object is given. To make a solid mesh, a voxelization of the interior of the surface could be performed and the resulting voxels could be used as a basis for creating a rectilinear grid structure resembling the solid object. The surface mesh can be coupled to the grid-structure by initially mapping each vertex into the grid cube containing it. During deformation, the vertex position can be updated when the rectilinear grid deforms by using trilinear interpolation of the deformed positions of the grid nodes of the containing grid cube. The mesh coupling idea is a well-known technique for deformable objects (17).

Although a lot of work has been done in this area, we decided not to explore it in

our work. The use of Bézier surfaces already fits with the use of surface meshes. There is no straightforward analogy to use Bézier with this kind of simulations and adapting it would just insert new elements that not necessarily yield better visual results.

3 BÉZIER CURVES AND SURFACES

Originally conceived to aid in the representation of car body parts, Bézier curves and surfaces have been widely used in computer graphics applications, including computer games. In this chapter we will present the basic math needed to understand these geometric tools and how they will be used in our work. For the sake of simplicity almost all concepts will be presented using Bézier curves. Keep in mind that all also apply to Bézier surfaces with little or no effort.

A parametric curve defined in three dimensions is given by three univariate functions:

Q(u) = (X(u), Y(u), Z(u)), where 0 u 1

A parametric curve in Computer Graphics is usually a polynomial:

Q(u) = p_0 + $p_1 u$ + $p_2 u^2$ + ... + $p_n u^n$, where p_i ^{3}

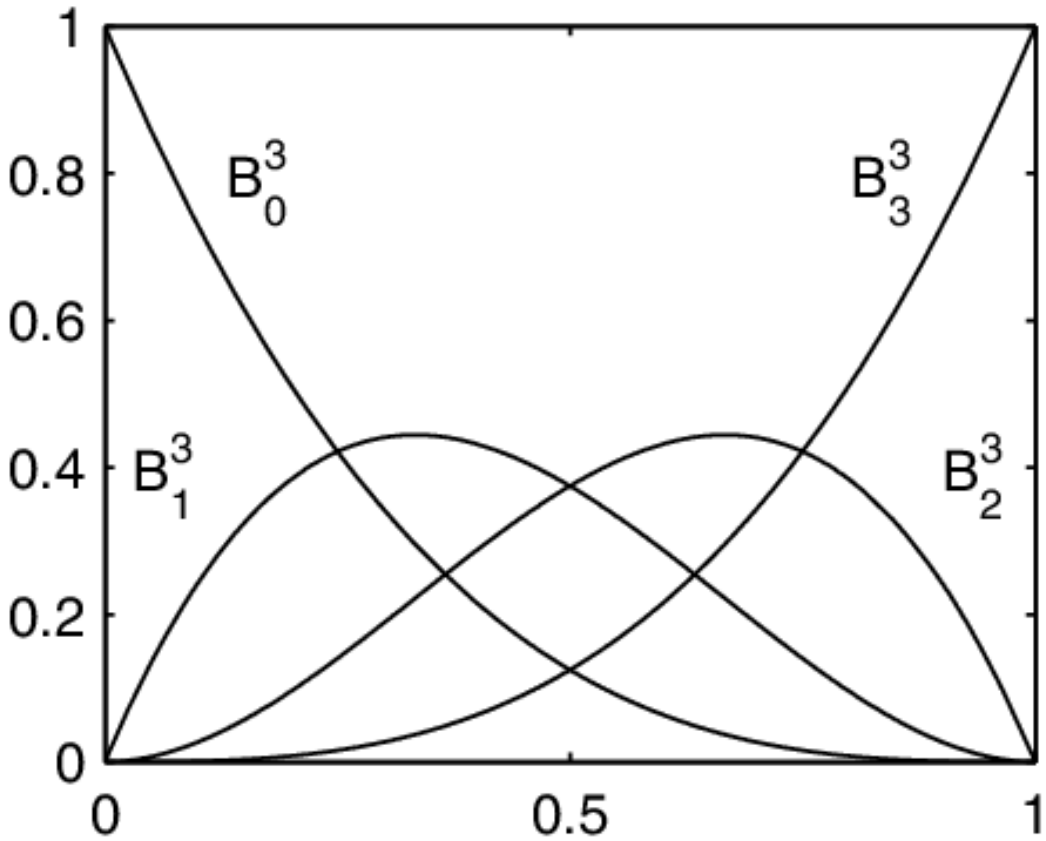
In computer graphics, especially for interactive real-time applications such as computer games, we normally use cubic polynomials. For degrees greater than 3 there is a tradeoff between curve flexibility and descriptions that are more cumbersome to work with. However, there are a number of ways to overcome the limitations of using higher degrees on curves by using techniques to connect lower degree curves, B-Splines being the most used for Bézier curves.

The relationship between the shape of the curve and the polynomial coefficients are not very intuitive. Instead of having to manipulate these coefficients directly, Bézier provided us with a very intuitive and efficient way to represent the curve using the notion of control points and basis functions. Taking into account the parametric equation, the control points are represented by the coefficients and the basis function represented by the u factor. In Bézier curves and surfaces we use Bernstein polynomials as basis functions. Bernstein polynomials are defined as below:

$B_{i,n}$ (u) = $C_i^n u^i (1 - u)^{n-i}$, where C_i^n = n! / (i!(n-i)!)

If n=3, this generates the basis function for the quadratic Bézier curve as shown in Figure 12. Interestingly, when n is 1, this generates the blending functions for the parametric line equation, so we can think of the Bézier curve as a generalization of a line to higher orders.

Figura 11 – The Bernstein polynomials of degree 3



https://www.researchgate.net/figure/221612648_fig1_Fig-1-The-Bernstein-polynomials-of-degree-3

For a cubic Bézier curve the basis functions are, shown in Figure 11:

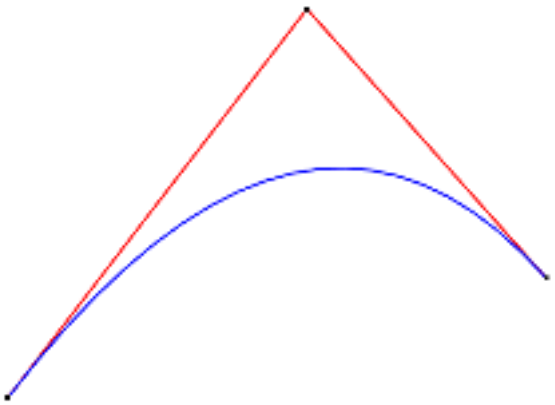
$$B_{0,3}(u) = (1-u)^3$$

$$B_{1,3}(u) = 3u(1-u)^2$$

$$B_{2,3}(u) = 3u^2(1-u)$$

$$B_{3,3}(u) = u^3$$

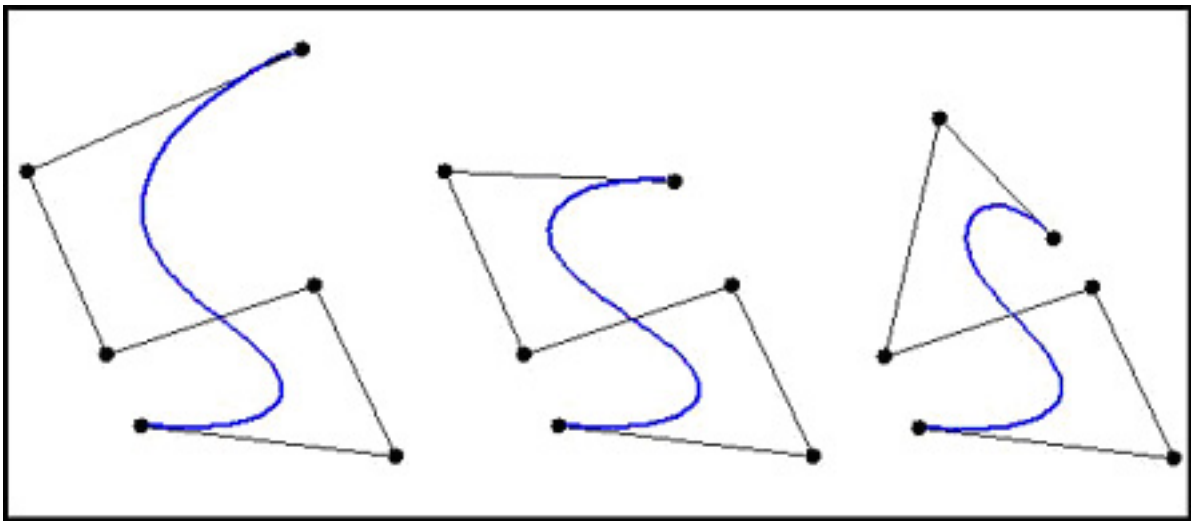
Figura 12 – Quadratic Bézier Curve



Source: The Author

These curves show the influence that each control point has on the final curve form. When u=0 the basis function B0, 3 = 1 while the others are 0. When u=1 the basis function B3, 3 = 1 while the others are 0. From this we know that when u=0, p0 will have the most influence and when u=1, p3 will have the most influence. The control points p1 and p2 have the most effect when u=1/3 and u=2/3 respectively. The manner in which the basis functions affect the shape of the curve is the reason they are called blending functions. This leads to a very intuitive description of a curve and is easy for artists to manipulate them. Figure 13 shows a sequence of Bézier curves so the reader can attest how intuitive the curve is given the control points.

Figura 13 – Different Bézier Curves



http://www.dankalman.net/AUhome/dofiles/doss_e0512bezier.html

3.1 The deCasteljau Algorithm

The way Bézier defined the curves is really good for mathematical purposes such as proving properties and getting curve derivatives.On the other hand, they have a high computational cost and their uses on real-time application are restrictive. To make Bézier curves and surfaces viable for our goal, computer games, we make extensive use of the deCasteljau algorithm. Historically, it is with this algorithm that the work of deCasteljau started in 1959. The only written evidence is two technical reports with confidential material property of Citroen. deCasteljau's work went unnoticed until 1975, when W. Boehm obtained copies of the reports in 1975. To demonstrate the algorithm, we will use linear interpolation for the quadratic case and show how it extends to high order curves. For the most of the text we will restrict the discussion to quadratic and cubic cases, but everything can be extended to higher degree curves, though cubic is the highest order ever used in practice. Let $\mathbf{b}_0$, $\mathbf{b}_1$, $\mathbf{b}_2$ be any three points in ^{3}, and let t . Construct

$b_0{}^1(t) = (1-t)\ \mathbf{b}_0 + t\mathbf{b}_1$

$b_1{}^1(t) = (1-t)\ \mathbf{b}_1 + t\mathbf{b}_2$

$b_0{}^2(t) = (1-t)\ \mathbf{b}_0{}'(t) + t\mathbf{b}_1{}^1(t)$

Inserting the two first equations on the third one, we get

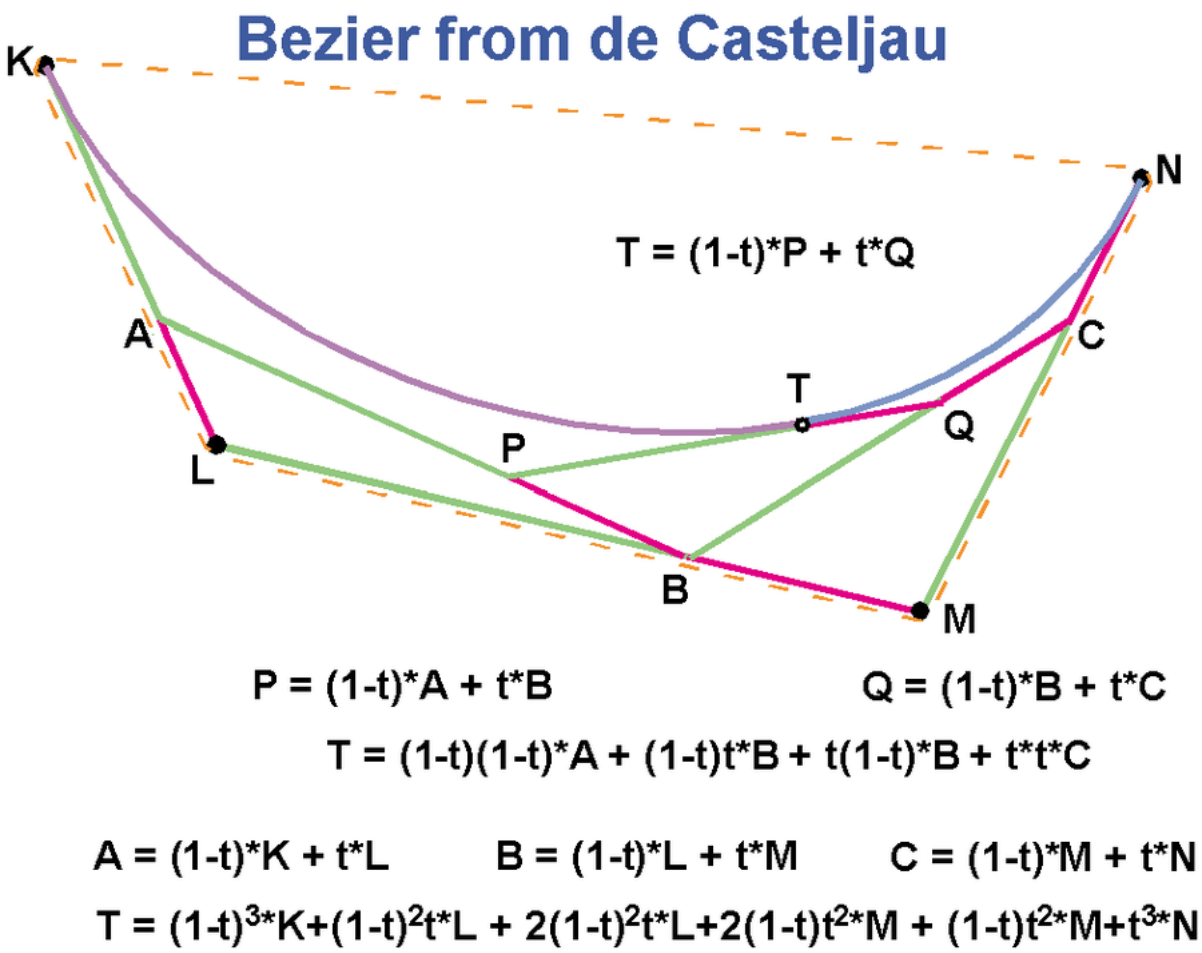
$b_0{}^2(t) = (1-t)^2\ \mathbf{b}_0 + 2t(1-t)\mathbf{b}_1 + t^2\mathbf{b}_2$

This construction consists of repeated linear interpolation and its geometry illustrated in Figure 14. Parabolas are plane curves. However, many applications require true space curves. For those purposes, the previous construction for a parabola can be generalized to generate a polynomial curve of arbitrary degree n: Given n control points $\mathbf{b}_0$, $\mathbf{b}_1$, $\mathbf{b}_2$ be any three points in ^{3}, and let t set

$b_i{}^r(t) = (1-t)\mathbf{b}_i{}^{r-1}(t) + t\mathbf{b}_{i+1}{}^{r-1}(t)$ for r = 1, . . . , n and i = 0, . . . , n - r

And $\mathbf{b}_i{}^0(t) = \mathbf{b}_i$. Then $\mathbf{b}_0{}^n(t)$ is the point with parameter value t on the Bézier curve $\mathbf{b}^n$.

Figura 14 – Construction of a Bézier Curve using the deCasteljau algorithm.



https://people.eecs.berkeley.edu/∼sequin/CS284/LECT09/L3.html

3.2 Convex Hull Property

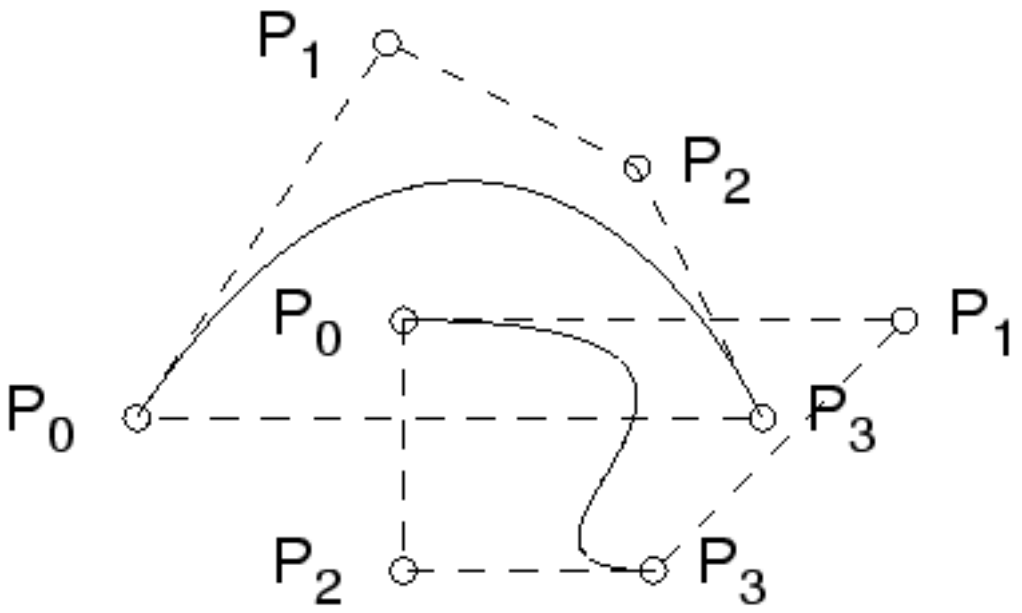
Among all the Bézier curves and surfaces properties, the property known as convex hull property is the most valuable for us. The importance of the convex hull property lies in what is known as interference checking and collision test, needed by physically-based simulations.

Take for instance the example shown on the previous section, where we constructed a quadratic Bézier curve. For t between 0 and 1 all points generated by $\mathbf{b}_0{}^2(t)$ are contained in the triangle formed by $\mathbf{b}_0$, $\mathbf{b}_1$ and $\mathbf{b}_2$, see Figure 14.

Instead of actually computing a possible intersection for every single point combination – or try to solve the linear equation to check for solutions – we can perform a much cheaper test: circumscribe the smallest possible box around the control points of each curve. Since each box contains the control points, and by the convex hull property they also contain the whole curve, we can test the boxes for collision (a trivial test) and assure if the two curves do not intersect each other. The high cost tests then are only

used if the hulls intersect each other, since even if this is true we can't say for sure the curves intersect each other as shown in Figure 15. The possibility for a quick decision of no interference is extremely important, since in practice there are thousands of objects in a scene to perform collision tests.

Figura 15 – Convex hulls intersect but curves don't



Source: The Author

3.3 Splines

Bézier curves provide a powerful tool in curve design, but they have some limitations if the curve has a complex shape and needs lots of control points. As previously stated, in practice, we only want to use four control points maximum. To overcome that limitation we make use of a curve composition technique known as Spline curves. For the purposes of this work, we will focus on cubic and quadratic Spline curves[11].

A Spline curve s is the continuous map of a collection of intervals $u_0 < \ldots < u_L$ into ^{3}, where each interval [u_i, u_{i+1}] is mapped onto a polynomial curve segment, a Bézier curve. Each real number u_i is called a breakpoint or knot. For each interval on the knot sequence we can define a local parameter t that varies from 0 to 1 while u varies from u_i to u_{i+1}:

$t = u - u_i / u_{i+1} - u_i = u - u_i / \Delta i$

Given the notion of global parameter u and local parameters t we can talk about the whole curve s in terms of u or about the individual segments of s as Bézier curves in terms of t. We adopt the definition s_i for the ith segment of s, and we write $s(u) = s_i(t)$.

Suppose we are given two Bézier curves and, with polygons $\mathbf{b}_0, \ldots, \mathbf{b}_n$ and $\mathbf{b}_n, \ldots, \mathbf{b}_{2n}$ respectively. We can say these two Bézier curves defined over $u_0 \ u \ u_1$ and $u_1 \ u \ u_2$, are r times continuously differentiable at $u = u_1$ if and only if

$\mathbf{b}_{n+1} = \mathbf{b}_{n-i}{}^{i}$ (t); i = 0, . . . , r

3.4 Bézier Patches

At the previous sections we defined Bézier curves and explored some of their properties. However, most of the geometric models used in cutting edge computer games can't be represented by curves. To create shapes like in Figure 16 can be created using Bézier patches, which are nothing more than an extension of Bézier curves.

Figura 16 – Freshwater uses a Bézier patches system with procedural displacement waves.

Source: [40]

We can think Bézier patches as a set of curves created using as control points the points of another Bézier curve. In other words, what if the curve equation was dependant on two variables instead of one and looked like the one below shown for quadratic Bézier curves.

From examining the equation for the surface, it is pretty clear that fixing one of the parameters to a constant value; we can generate a set of control points that can be used to generate the points on the surface varying the other parameter.

This leads to some useful properties known about Bézier patches. The most important of all is that since we are forming the surface from Bézier curves, the convex hull

property, used for coarse collision detection, also applies to surfaces. The intermediate points that generate the final surface points are inside the convex hull of the original control points hence the final points are inside the hull.

Another important thing to notice is that we can connect surfaces in much the same way we did on curves. If the patches share the same control points along the edges, they will join at the Bézier curve defined by those control points. Continuity on edges are possible if the control points are adjusted to ensure some kind of smooth connection [11].

4 DUNAS FRAMEWORK

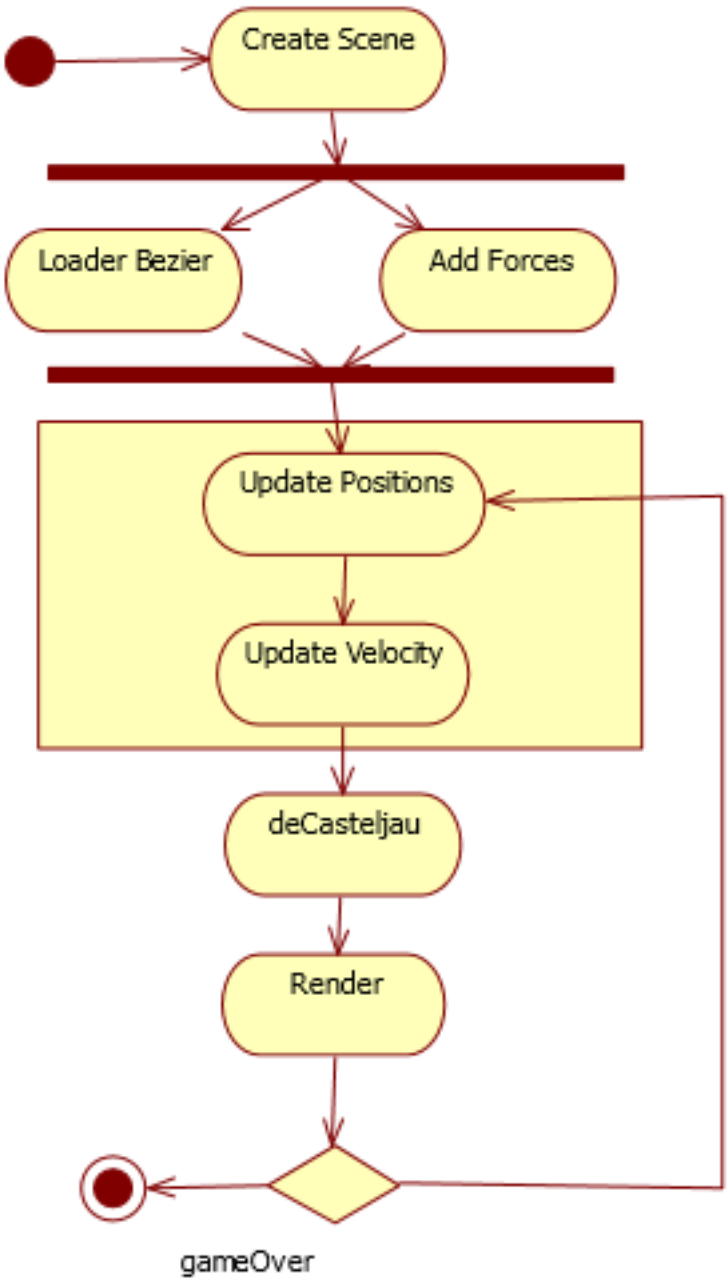
This chapter introduces the Dunas Framework that simulates deformable Bézier models based on physics principles. The first subsection describes the basic pipeline of the framework. The second one describes the physics component for mass-spring systems. The Bézier component, created for the Revolution Engine [4], is described on the forth subsection of this chapter. Following, a brief explanation of the Revolution engine and how the integration was made. A small section explains how the Dunas framework should be integrated into the user's application. In the last section we explain how the framework should be used to create a rope and a piece of cloth in the form of a tutorial on how to use Dunas.

4.1 Framework Overview

The Dunas framework was created on top of the Revolution Engine [4]. The Revolution Engine was chosen based on the familiarity of the author with it, since I created it, and also because this would be a real test of the engine's architecture and design. During the development of the Dunas framework some of the weaknesses of the Revolution engine were identified and the engine gained some new features, like the persistence system.

Regarding the Dunas framework, there is one unique processing pipeline that handles the physics rules to the control points of a Bézier model, either curve or surface. The whole architecture is based on this pipeline and each of the sections in this chapters explains how each one of each works.

Figura 17 – Framework Basic Pipeline



Source: Author

At the first stage of the pipeline on Figure 18, the model is a set of Bézier control points loaded from a file. A XML file definition was created to feed the engine. The control points goes into the physics component – responsible for updating the particle's position and velocity at each time step - that processes the model using the shape shifting algorithms. The modified points are input to the Bézier component that applies the deCasteljau algorithm and create the final model to be rendered by the rendering visitor of the Revolution Engine, the last step of the pipeline.

4.2 Particle Physics Component

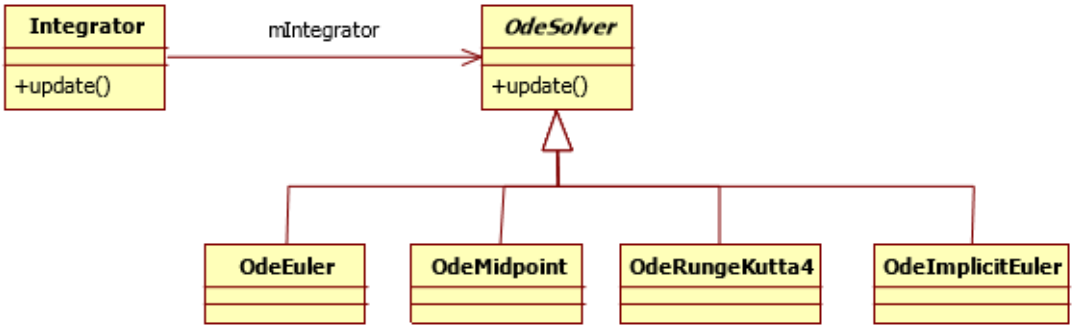
Both of the physics components, mass-spring and finite element method, uses some basic concepts from physically based modeling such as: Particles, Integrators, Impulses and Forces. A base framework, called particle physics component from now on, is built around these concepts and then evolves to more sophisticated components for mass-spring and finite element methods. A particle has a position but no orientation.

A particle, in this component, has a set of default properties named: position, mass, velocity and acceleration. At each time step the physics component updates the particle's position based on the current velocity and updates the acceleration and velocity according to the forces acting on it. To calculate these new values of the properties we use an Integrator.

The integrator consists of two parts: one to update the position of the particle, and the other to update its velocity. The position will depend on the velocity and acceleration, while the velocity will depend only on the acceleration. Integration requires a time interval over which to update the position and the velocity – the time between two renders. Revolution engine uses a central timer system that calculates the duration of each frame.

The Dunas framework implements four different integrators – Euler, Runga Kutta and Implicit Euler - that can be chosen by the user according to the targeted application. It is important, as a framework, to give the user options. Especially in matters of integrators since each one of them works better than others according to the application. The diagram in Figure 19 shows the hierarchy used to handle the different methods.

Figura 18 – Integrator methods Strategy pattern



Source: Author

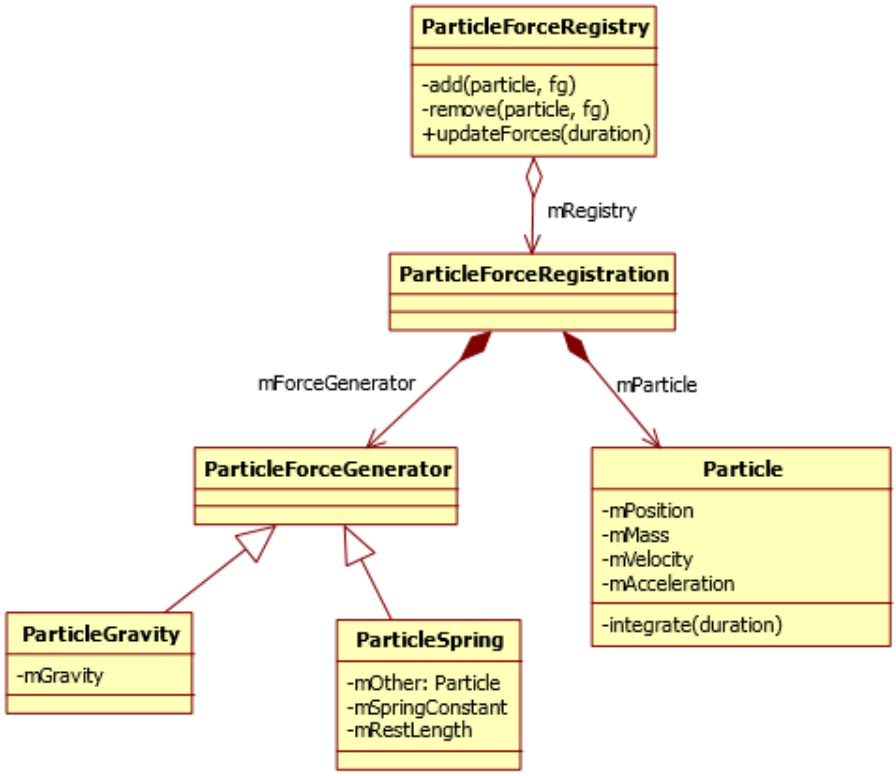
Euler's integration method is the prototype for a numerical solver for ordinary differential equations. The function F is a given. Knowing the input time t, a step size h,

and an input state X (t), the method produces an output time t + h and a corresponding state X(t+h). The general concept is encapsulated by an abstract base class, OdeSolver.

The integrators obey the Strategy design pattern [22]. The OdeSolver works as the base strategy class that defines the same interface for the supported algorithms. Each of the algorithms is implemented by a concrete class that extends the base OdeSolver class. The Integrator class acts as the Context class. The desired algorithm has its concrete solver is linked to the context. The Verlet method is used to update the particle's position at each time step.

The other concept that comes into play is force. To handle multiple forces acting on a particle at the same time we use the notion of force generators and a force accumulator that acts according to the D'Alembert's Principle[1] . The force accumulator acts, as the name implies, accumulating forces by the D'Alembert's principle and then is used in the integration step.

Figura 19 – Particle Physics Component Architecture



Source: Author

A force generator is responsible to deal with a wide range of different forces with different mechanics for their calculation. Some might be constant, others might apply

[1] D'Alembert's Principle, can be simplified to our needs, states that the resultant force acting on an object can be calculate by simply adding all forces acting on it.

some function to the current properties of the object, some might require user input, and others might be time based.

Ideally we would like to be able to abstract away the details of how a force is calculated and allow the physics engine to simply work with forces in general. This would allow us to apply any number of forces to an object, without the object knowing the details of how the force is calculated or how it changes over time.

Using the force generators there can by any force generators as there are types of forces, but each particle doesn't necessarily needs to know how a force generator works. The object simply uses a consistent interface to find the force associated with each generator: these forces can them be accumulated and applied in the integration step. The whole architecture is in the UML diagram Figure 20.

4.3 Mass-spring component

As stated in the second chapter, the spring mathematics is ruled by Hooke's law. Hook discovered that the force exerted by a spring depends only on the distance the spring is extended or compressed from its rest position. A spring extended twice as far will exert twice the force. The formula is therefore

F = -k(|d| - l_0) d

The spring constant k is a value that gives the stiffness of the spring. The same force is felt at both ends of the spring. In other words, if two particles are connected by a spring, then they will each be attracted together by the same force, given by the preceding equation.

The d vector is from the end of the spring attached to the object we're generating a force for, to the other end of the spring. It is given by

d = x_A - x_B

Where is the position of the end of the spring attached to the object under consideration, and is the position of the other end of the spring. The equation is defined in terms of one end of the spring only (the end attached to the object we are currently considering), we can use it unmodified for the other end of the spring, when we come to process the object attached there. Alternatively because the two ends of the springs always pull toward each other with the same magnitude of force, we know that if the force on one end if f, then the force on the other will be –f.

The force generator used for mass-spring system takes this information into account and uses an optimized approach calculating the force once for both ends, using a cache to save the force calculated to one end to same time recalculating it for the other.

It's not only a coiled metal spring that can be simulated using Hook's law. It applies to a huge range of natural phenomena. Anything that has an elastic property will usually have some limit of elasticity in which Hook's law applies. The applications are limitless.

4.4 Bézier Component

This component is responsible to process the deformed control points and generate the rendering model using the deCasteljau algorithm. The Bézier patch is implemented by the Geometry node in the Revolution's Scene Graph, since it is intimately related to this work this section describes how it works despite the fact that it is actually part of the Revolution Engine and not the Dunas framework.

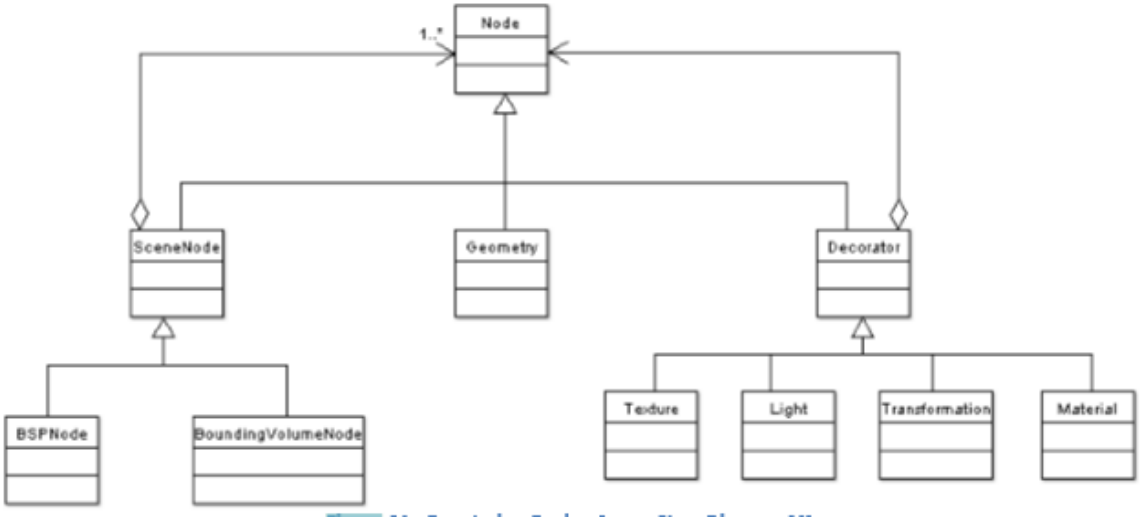
The revolution engine scene graph store all geometry information in a Geometry node – except the vertices that are stored in a unified vertex array for performance and memory improvements. The geometry node is the lowest level structure, scene graph diagram in Figure 21, in the scene graph and it can be one of five different types [41]:

1) A large polygon (polygon with n vertices);
2) A Bézier patch (npu x npv vertices defining the control mesh);
3) Triangle 'soup' (n vertices defining a mesh of triangles;
4) Triangle strip (tri-strip)
5) Triangle fan (tri-fan)

Obviously Dunas uses the Bézier patch type and create some mechanisms, described in the next section, to update the control points based on a physics simulation, described in the previous chapter and sections. Revolution stores an organized vertex array buffer storing the control points so the Geometry node only needs two integers to reference it (the vertex index followed by number of vertices) and each geometry node uses a fixed length structure [41].

To create Geometry nodes of Bézier patch type, two particular integers are specified: the number of control vertices in each direction u and v (npu, npv). The vertices of each control point are stored in the vertex array. The product of the npu and npv gives the number of control points, represented as vertices, in the array.

Figura 20 – Revolution Engine Scene Class Diagram



Source: [4]

To draw a patch, we use triangle strips. Bézier patches have the advantage that they can be used as an easy LOD facility [11].

In order to improve performance, and make sure the component has the best implementation available, Revolution Engine already uses OpenGL's evaluators [4][9]. Our only real concern while rendering Bézier models is to determine the granularity of the subdivision. However, this is done empirically and creating an appropriate subdivision strategy can be quite complicated – too complicated to be in the scope of this work.

4.5 Revolution Integration

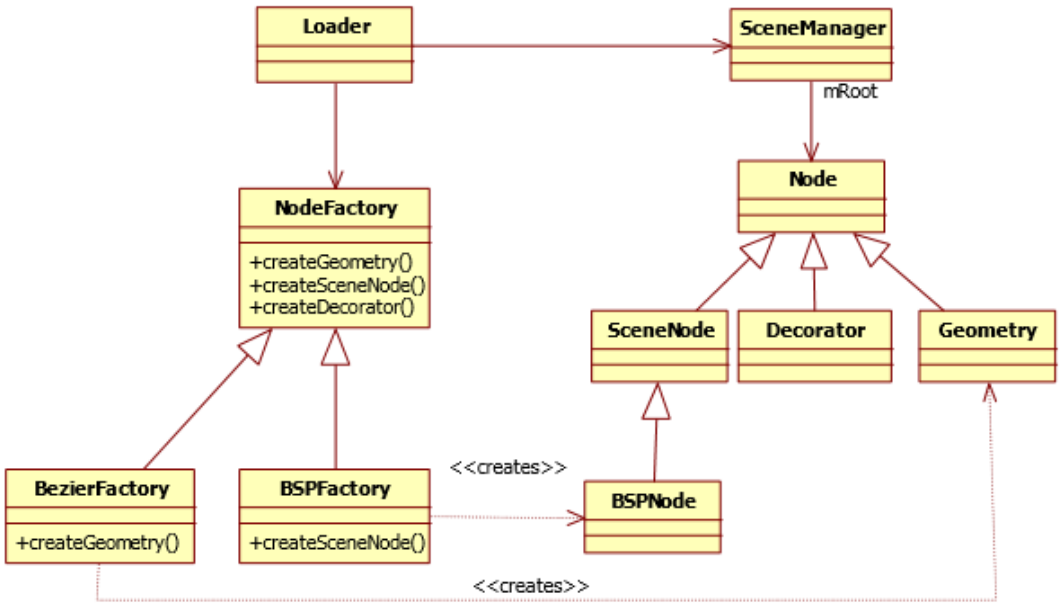
We now have two physics component that can get an array of vertices and update their positions based on physics rules. We also have a Bézier component, part of the Revolution engine, that can use the deCasteljau algorithm to create Bézier curves and surfaces based on a set of control points. However, we still need an add-on for the Revolution engine that actually updates the control point at each frame and how it fits in the engine architecture smoothly. The approach used here can be used with any other engine to use the physical components created.

For the revolution engine this step is pretty straight forward. At every rendering step the force generators are updated which in turn updates the vertices attached to it. The Geometry nodes already reevaluate the mesh points based on the control points every frame update.

The usual way to create a scene graph is to load it from a scene definition file. The first work on the Revolution Engine doesn't mention anything about scene definition files. To extend the engine, and make it more useful, a scene definition file

and persistence system was created.

Figura 21 – Simplified Architecture of the persistence system.



Source: Author

The persistence system is built around the Abstract Factory design pattern [34]. While loading the scene XML file, the main persistence class makes use of an Abstract Factory class that based on the tag name uses the appropriate method and returns a Node. A simplified class diagram is present in Figure 22. Some of the classes are missing but the core idea is presented. The BSPFactory creates only BSPNode, an OctreeFactory also exists, for instance. The Figure 23 presents an analog scheme for the force generators.

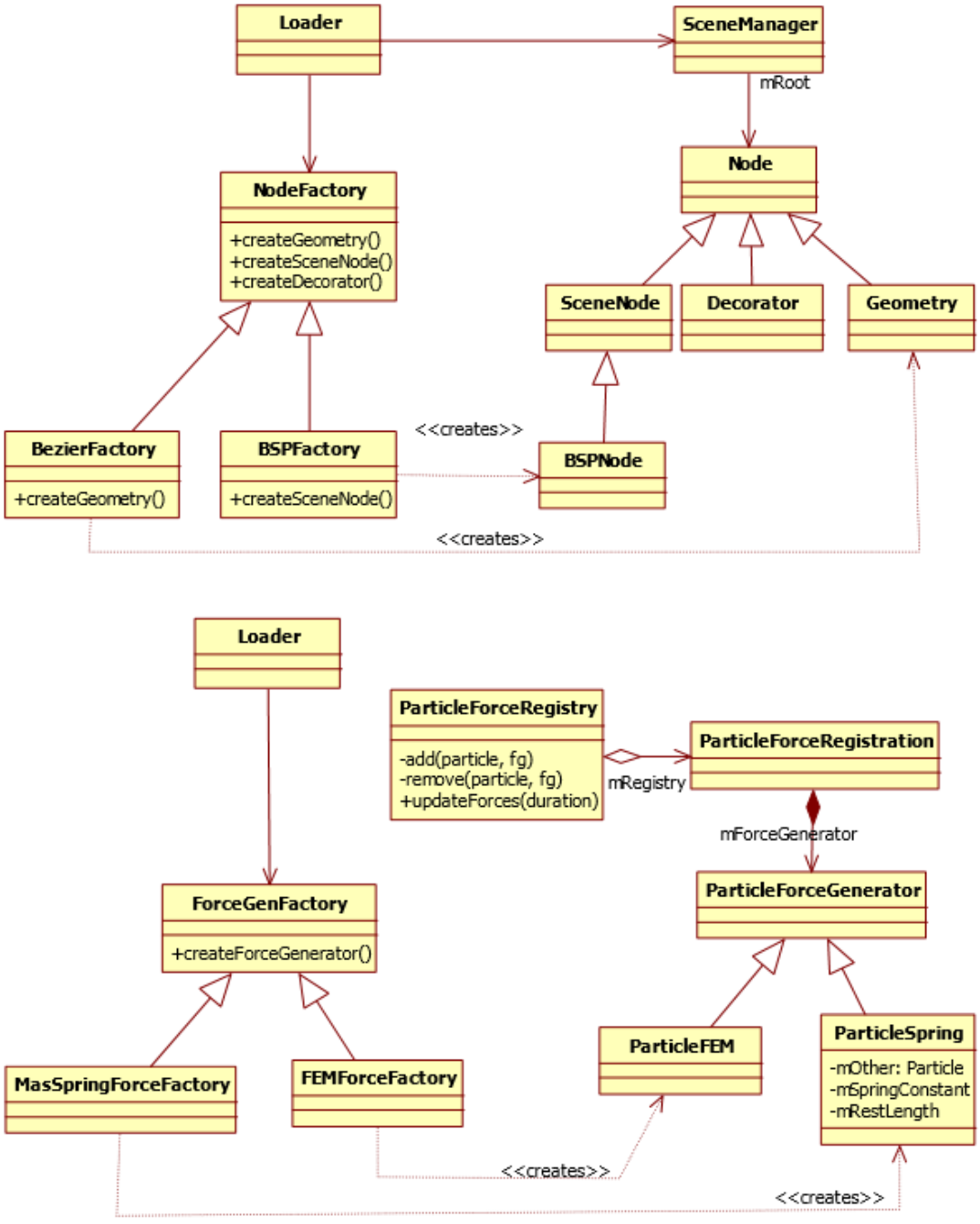
The scene XML is pretty simple. It only needs to hold information about a Bézier curve or surface. The Bézier model is defined by the number of control points in each direction and the control points. The following code is a sample of a possible scene.

Código 4.1 – Scene XML

```
<node type="geometry" name="BezierCloth">
    <geometry type="bezier" npv="4" npu="4">
        <vertex x="-6.0" y="8.0" z="0.0"/>
        <vertex x="-2.0" y="8.0" z="0.0"/>
        <vertex x="2.0" y="8.0" z="0.0"/>
        <vertex x="6.0" y="8.0" z="0.0"/>
        <vertex x="-6.0" y="4.0" z="0.0"/>
        <vertex x="-2.0" y="4.0" z="0.0"/>
        <vertex x="2.0" y="4.0" z="0.0"/>
        <vertex x="6.0" y="4.0" z="0.0"/>
        <vertex x="-6.0" y="0.0" z="0.0"/>
```

```
            <vertex x="-2.0" y="0.0" z="0.0"/>
            <vertex x="2.0" y="0.0" z="0.0"/>
            <vertex x="6.0" y="0.0" z="0.0"/>
            <vertex x="-6.0" y="-4.0" z="0.0"/>
            <vertex x="-2.0" y="-4.0" z="0.0"/>
            <vertex x="2.0" y="-4.0" z="0.0"/>
            <vertex x="6.0" y="-4.0" z="0.0"/>
        </geometry>
    </node>
</scene>
```

Figura 22 – Physics Creation Persistence System



Source: Author

This definition work perfectly to define a Bézier Model. However, the physics properties can't be inferred by this definition only. The XML also needs information regarding how these points' turns into particles and how they are attached to each other. In order to do this we include an id property on the geometry tag referencing another tag that holds the physics properties. This turns the geometry tag into something like:

<geometry type="bezier" npv="3" npu="3" physics="01">

The reason to put the information on two different tags is to be able to use two different factories and each one reads only the part of the XML needed. The referenced physics properties must hold particle's mass and method specific information. For mass-spring system the information required is the connections between vertices and the spring constant. The following code is a sample of a mass-spring XML file:

Código 4.2 – A particle system representation in XML for the framework

```
<mass-spring mass="1.5" gravity="-4.87" damping="0.05"
   constant="3.0" restLength="4.0" geometry="BezierCloth">
<!-- structural -->
<connection node1="0" node2="1"/>     <connection node1="0"
   node2="3"/>     <connection node1="1" node2="2"/>     <
   connection node1="1" node2="4"/>     <connection node1="2"
    node2="3"/>     <connection node1="2" node2="5"/>     <
   connection node1="3" node2="6"/>          <connection
   node1="4" node2="8"/>     <connection node1="4" node2="5"/
   >     <connection node1="5" node2="6"/>     <connection
   node1="5" node2="9"/>     <connection node1="6" node2="7"/
   >     <connection node1="6" node2="10"/>     <connection
   node1="7" node2="11"/>

    <connection node1="8" node2="9"/>     <connection node1="
       8" node2="12"/>     <connection node1="9" node2="10"/>
            <connection node1="9" node2="13"/>     <
       connection node1="10" node2="11"/>     <connection
       node1="10" node2="14"/>     <connection node1="11"
       node2="15"/>

    <connection node1="12" node2="13"/>     <connection node1
       ="13" node2="14"/>     <connection node1="14" node2="
       15"/>

    <!-- bending -->     <connection node1="0" node2="2"/>
           <connection node1="0" node2="8"/>     <connection
       node1="1" node2="3"/>     <connection node1="1" node2=
       "9"/>     <connection node1="2" node2="10"/>

    <connection node1="4" node2="6"/>     <connection node1="
       4" node2="12"/>     <connection node1="5" node2="7"/>
           <connection node1="5" node2="13"/>     <connection
        node1="6" node2="14"/>
```

```
    <connection node1="8" node2="10"/>     <connection node1=
       "9" node2="11"/>

    <!-- shear -->     <connection node1="0" node2="5"/>
       <connection node1="1" node2="4"/>     <connection
       node1="1" node2="6"/>     <connection node1="2" node2=
       "5"/>     <connection node1="2" node2="7"/>     <
       connection node1="3" node2="6"/>

    <connection node1="4" node2="9"/>     <connection node1="
       5" node2="8"/>     <connection node1="5" node2="10"/>
           <connection node1="6" node2="9"/>     <connection
       node1="6" node2="11"/>     <connection node1="7" node2
       ="10"/>

    <connection node1="8" node2="13"/>     <connection node1=
       "9" node2="12"/>     <connection node1="9" node2="14"/
       >     <connection node1="10" node2="13"/>     <
       connection node1="10" node2="15"/>     <connection
       node1="11" node2="14"/>

  </mass-spring>
</dunas>
```

The particle-system tag states that it is a mass-spring system, the id of the system reference by the Bézier model definition tag and the spring constant. A list of connection nodes defines the topology of the spring mesh connecting neighbor nodes in this particular case.

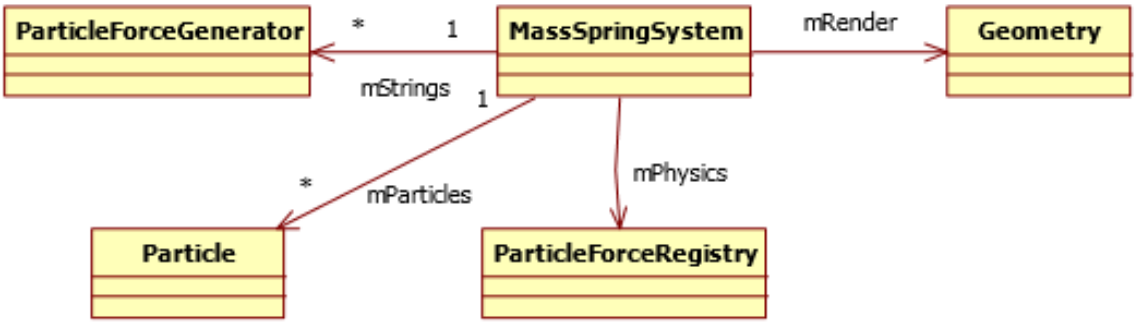
4.6 The Application Programming Interface

The whole architecture presented is used to simulate mass-spring system. How the user actually use it is done via one single class, the interface of Dunas to other systems, called MassSpringSystem.

The MassSpringSystem class does the whole job of interfacing with the Revolution engine, as described in the previous section, transparently to the user. The user can either, use the class alone or extend it to create more functionalities, as described in the next section.

The following class diagram, Figure 24, describes how the MassSpringSystem class interacts with the rest of the Dunas framework.

Figura 23 – How the MassSpringSystem class is related to Dunas and Revolution (via Geometry).



Source: Author

At the start, the init() method of the class has to be called and inside it the createParticles() method too, so the MassSpringSystem already knows the correct geometry node is assigned and the creation of a list of particles, that will be part of the particle system can be created.

The next step is creating the connection between the particle which can be done using the persistence system, loading a file, or by code – both ways are explained in details on the next section. After initialization, the update() method of the MassSpringSystem class needs to be called so it can properly simulate the mass spring system involved. This should be done by the user in the main game class that extends the REngine class – also explained in the next section. The following code shows how this is done in the MassSpringSystem class update() method.

Código 4.3 – MassSpringSystem class update() method

```
void MassSpringSystem::update(real fTimeSinceLastFrame)
{
    // update forces
    mPhysics->udpateFoces(fTimeSinceLastFrame);

    // update particles
    std::vector<pengine::Particle*>::iterator i = mParticles.
       begin();
    int num = 0;
    for(; i != mParticles.end(); i++)
    {
        (*i)->integrate(fTimeSinceLastFrame);
        revolution::Vector3 pos = (*i)->getPosition();
        mRender->setControlPoint2(num, pos.x, pos.y, pos.z);
        num++;
    }
}
```

4.7 Using the framework

This last section presents the code of the application used to create the samples we analyze in details in the next chapter. The explanation is made on the form of a step-by-step tutorial so the reader is able to recreate the tests alone.

The first step is downloading the library and setup the environment in your favorite tool for C++ development. In this tutorial we use the Visual C++ 2008 Express Edition [42] which is free and can be downloaded from Microsoft's website [43]. You need to create a project in the Visual C++, if you don't know how to do it there are plenty of tutorials in the internet that explain how to do it [44].

The Revolution engine and the Dunas framework are both compiled as static libraries (.lib files). So, after you have created a new Win32 Application (NOT a console application) in Visual C++, you will need to link the libraries – for the engines and the libraries they use. In Visual C++ go to Project, Settings, and then click on the LINK tab. Under "Object/Library Modules" at the beginning of the line (before kernel32.lib) add Revolution.lib PEngine.lib openGL32.lib glut32.lib ILUT.lib DevIL.lib ILU.lib. Once you've done this click on OK. You're now ready to write an OpenGL Windows program.

There is basically two ways to create a physics simulation using Dunas. One is using coding only and defining the curves and surfaces and attaching it to the physics engine. This is done in the first sample of this tutorial. The second way is also very simple and uses the persistence system created in the work to load a Bézier model and the mass-spring setup defined in XML files. The principle is the same but using the persistence model gives us a lot more flexibility to test the application since we don't need to recompile it each time we want to change any property of the system being it a control point rest position or the stiffness constant of the system.

To make a rope simulation, using code only, we start by creating a new class called Rope that extends the MassSpringSystem class from the Dunas Framework. The MassSpringSystem class is responsible for the integration between the physics engine, Dunas, and the rendering engine, Revolution. The MassSpringSystem class has two methods that can be overwritten by the user to customize the system we are building. The first one is the init() method, where we are going to put the code for the initialization of the geometry, control points, and the mass-spring system setup, connection between the particles. The following code is the simple declaration of a class that extends MassSpringSytem:

Código 4.4 – Extending the MassSpringSystem class

```
#include "MassSpringSystem.h"

namespace revolution {  class Geometry; }
```

```
class Rope : public pengine::MassSpringSystem
{
public:
    Rope(revolution::Geometry* render);
    virtual ~Rope(void);

    // init method, load the physics properties and get ready
        to render
    // this methods should be replaced if no definition file
       was indicated
    void init();

};
```

The implementation of these methods is very simple, we just need to set a list of control points and then attach one to another based on any criteria we think it fits. For this sample we create a simple rope with eight points, horizontally aligned, each one connected to two others, or just one for the end points.

Código 4.5 – Creating the mass-spring system in C++

```
#include "pfgen.h"
#include "Particle.h"
#include "Geometry.h"

Rope::Rope(revolution::Geometry* render) : pengine::
   MassSpringSystem(render)
{ }
// init method, load the physics properties and get ready to
   render
// this methods should be replaced if no definition file was
   indicated
void Rope::init()
{
    mParticleMass = .5f;  mDamping = 0.15f;  mGravity = -4.87
       f;

 // create a simple rope with a set of points vertically
    aligned
    for (int i = 0; i < 8; i++)  {   mRender->
       setControlPoint2(i, -16.0f + i*2, 12.3f, 0.0f);  }

 // this methods needs to be called so the MassSpringSystem
    class  // can create the particles based on the
    mControlPoints just set
    createParticles();

    revolution::real restLength = 2.0f;
    revolution::real springConstant = 35.f;

    // create connectins, springs
```

```
    pengine::ParticleAnchoredSpring* springBA =    new
       pengine::ParticleAnchoredSpring(   new revolution::
       Vector3(-16.f, 12.3f, 0.f), springConstant, restLength
       );

    mPhysics->add(mParticles[1], springBA);
    pengine::ParticleSpring* springBC = new pengine::
       ParticleSpring(   mParticles[2], springConstant,
       restLength);
    mPhysics->add(mParticles[1], springBC);

    pengine::ParticleSpring* springCB = new pengine::
       ParticleSpring(   mParticles[1], springConstant,
       restLength);
    mPhysics->add(mParticles[2], springCB);
    pengine::ParticleSpring* springCD = new pengine::
       ParticleSpring(   mParticles[3], springConstant,
       restLength);
   mPhysics->add(mParticles[2], springCD);

pengine::ParticleSpring* springDC = new pengine::
   ParticleSpring(   mParticles[2], springConstant,
   restLength);
mPhysics->add(mParticles[3], springDC);

pengine::ParticleSpring* springDE = new pengine::
   ParticleSpring(   mParticles[4], springConstant,
   restLength);
mPhysics->add(mParticles[3], springDE);

pengine::ParticleSpring* springED = new pengine::
   ParticleSpring(   mParticles[3], springConstant,
   restLength);
mPhysics->add(mParticles[4], springED);

pengine::ParticleSpring* springEF = new pengine::
   ParticleSpring(   mParticles[5], springConstant,
   restLength);
mPhysics->add(mParticles[4], springEF);

pengine::ParticleSpring* springFE = new pengine::
   ParticleSpring(   mParticles[4], springConstant,
   restLength);
mPhysics->add(mParticles[5], springFE);

pengine::ParticleSpring* springFG = new pengine::
   ParticleSpring(   mParticles[6], springConstant,
   restLength);
mPhysics->add(mParticles[5], springFG);

pengine::ParticleSpring* springGF = new pengine::
   ParticleSpring(   mParticles[5], springConstant,
   restLength);
mPhysics->add(mParticles[6], springGF);
```

```
pengine::ParticleSpring* springGH = new pengine::
   ParticleSpring(   mParticles[7], springConstant,
   restLength);
mPhysics->add(mParticles[6], springGH);

pengine::ParticleSpring* springHG = new pengine::
   ParticleSpring(   mParticles[6], springConstant,
   restLength);
mPhysics->add(mParticles[7], springHG);

// store springs to correct simulation  mSprings.push_back(
   springBA);  mSprings.push_back(springBC);  mSprings.push_
   back(springCB);  mSprings.push_back(springCD);  mSprings.
   push_back(springDC);  mSprings.push_back(springDE);
   mSprings.push_back(springED);  mSprings.push_back(springEF
   );  mSprings.push_back(springFE);  mSprings.push_back(
   springFG);  mSprings.push_back(springGF);  mSprings.push_
   back(springGH);  mSprings.push_back(springHG);
```

The next part is creating a simple Revolution application and attaches this code to it so we can see how the system acts. In order to do that, we need to talk a bit more about the Revolution and how we create an application to use it. Each Revolution application has to extend the REngine class and override the methods: update() and init(). Above, the code with the Game class declaration that does this job:

Código 4.6 – Declaring the game class

```
#include "REngine.h"

// ahead declaration namespace pengine
{
    class MassSpringSystem;
}

class Game :  public revolution::REngine {
private:
/**   * Rope animation   */
    pengine::MassSpringSystem* mRope;

public:  Game(void);  ~Game(void);

 /**    * Start the engine and does all persistence loading
    operations   */
    void init();

 /**    * update   */
   void update(revolution::real fTimeSinceLastFrame);

};
```

This class overrides the REngine class and has an instance of a mass spring

system from the physics engine (mRope) that does the simulation, as we will describe earlier.

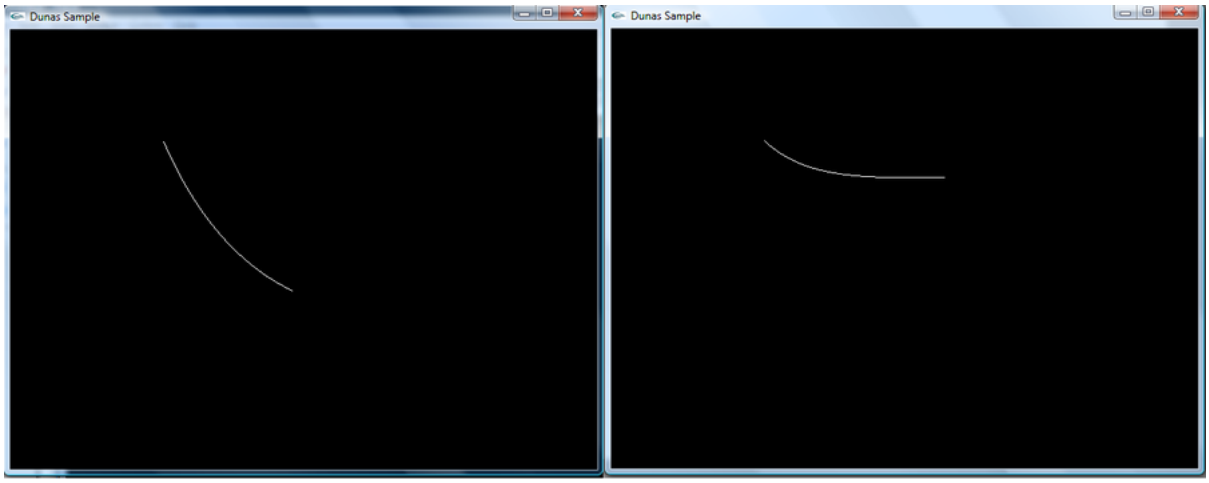
The init() method is responsible for the whole initialization and it's where we tell Dunas to load the scene definition and where it should be placed on the Scene Graph. The update() methods is the main loop of the engine and is called each frame before rendering. We should tell the Dunas framework to update the simulation every frame so Revolution can render it correctly, we do this by calling the update() method of the MassSpringSystem instance we are using. The code, for both methods, is very simple as the following code shows:

Código 4.7 – Implementing the game class

```
Game::~Game(void) {  if (mRope) delete mRope; }

/**   * Start the engine and does all persistence loading
   operations  */
void Game::init() {
// create the geometry node to hold the bézier curve and
   attach it to
// the scene graph so it can be rendered
    revolution::Geometry* geo = new revolution::Geometry(
       revolution::Geometry::BEZIER_CURVE);
    getSceneManager().getRoot()->addChild(geo);

 // create the mass-spring system and initialize it
    mRope = new Rope(geo);  mRope->init();

}

/**  * update */
void Game::update(revolution::real fTimeSinceLastFrame)
{
    if (mRope) mRope->update(fTimeSinceLastFrame);
}
```

Now, we just need to compile and run the code and see the OpenGL window opened and rendering the simulation, Figure 25. Next, we show how we do the same thing, however, using the persistence system.

Figura 24 – Screenshots of the rope simulation running



Source: Author

The basic difference in this way is that the init() method have to load the geometry node, with the Bézier curve, and the mass-spring setup of the physics simulation stored on XML files. This is done using the persistence function on the SceneManager and MassSpringSystem classes. In this sample we are going to load a cloth piece, using a Bézier surface, instead of a rope, using a Bézier curve. We are going to use inheritance again, this time with a class named Cloth that also extends the MassSpringSystem, as shown in the following code:

Código 4.8 – Declaration of the cloth class. The implementation would be similar to the rope one but with a surface.

```
namespace revolution {  class Geometry; }


class Cloth : public pengine::MassSpringSystem
{
public:
    Cloth(revolution::Geometry* render);
    virtual  ~Cloth(void);

 // init method, load the physics properties and get ready to
     render
// this methods should be replaced if no definition file was
   indicated
void init();
};
```

After loading the scene, the geometry node containing the Bézier model is retrieved from the scene in order to be attached to a mass-spring system. Note that the geometry node can be created by code, without the need to define a file with the curve control points as we did in the previous sample. However, the file system permits a far

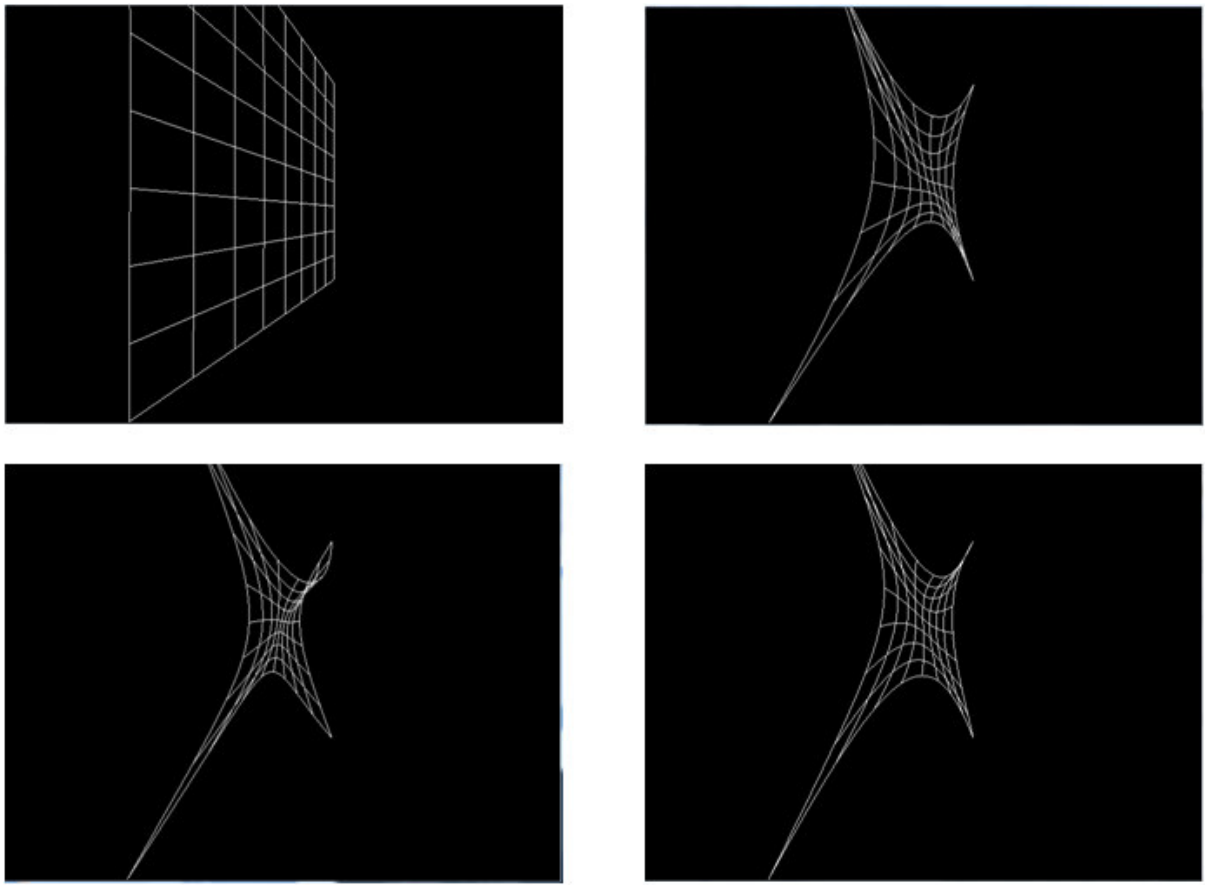
more flexible architecture and the possibility of non-coders to change the control points without recompiling. After that, we need to attach the geometry node to the mass-spring system with the proper simulation and load the connections, as shown in the code for the init() method of the Game class. The Game class is basically the same, as declared before, but with minor changes as shown in the following code rop

Código 4.9 – Implementation of the Game class for the cloth simulation

```
#include "Geometry.h"
#include "SceneManager.h"
#include "Cloth.h"

Game::Game(void) : mCloth(0) { }

Game::~Game(void) {  if (mCloth) delete mCloth; }

/**   * Start the engine and does all persistence loading
   operations  */
void Game::init() {

// load the scene from the file
    getSceneManager().loadFromFile("Cloth.xml");
// retrieve the geometry node that has the desired system to
   simulate
    revolution::Geometry* geo = (revolution::Geometry*)
       getSceneManager().getNode("BezierCloth");

 // create the mass-spring system
    mCloth = new Cloth(geo);
// load the connections
  mCloth->loadFromFile("ClothPhysicsSystem.xml");  // init it
       mCloth->init(); }

/**  * update */
void Game::update(revolution::real fTimeSinceLastFrame)
{
    if (mCloth) mCloth->update(fTimeSinceLastFrame);
}
```

Now, we just need to compile and run the code and see the OpenGL window opened and rendering the simulation, Figure 26 – the files used were the XML samples given earlier. Note that this time we just need to change things from the xml files and run the program again to see the changes.

Figura 25 – Screenshots of the cloth simulation running



Source: Author

5 RESULTS AND FUTURE WORK

This chapter presents some final considerations about this work and the main topics discussed by it, including the results of the study cases made using the framework and some work that can be done in the future the extend it.

5.1 Test Framework

As previously described in the first chapter, each sample will be evaluated using the following variables:

- Physically-based technique used
- Implementation
  - for Bézier curves if applicable
  - for Bézier surfaces if applicable
  - Simple UML static and dynamic models
- Performance of the technique in terms of frames per second and pre-computation required by the method
- Time to implement the technique and difficulty to integrate a renderer
- Images with visual results o Using wire-frame rendering o Using real-time illumination techniques supported by OpenGL 2.0

  All the tests were made using

- C++ Programming Language, Visual C++ 2008 Express Edition and OpenGL and Glut [10] (used by the Revolution Engine)
- Flash CS3 and Action Script 3.0
- Computer ASUS Notebook G1 Series
  - Intel® Core™2 Duo CPU T7700 @ 2.40GHz
  - 2 GBytes RAM Memory
  - Windows Vista Home Premium

Each of the samples is based on an already established application of the physics model or Bézier model that could be improved by animation. For the mass-spring physical method three samples were made for ropes, clothes and water. On the other hand, Bézier surfaces are widely used for smooth terrains and cloth models.

The following sections describe a sample each. The description of the sample discusses the points previously stated to compare the regular implementation with the implementation used in Dunas. The sections also talks about applications on problems of the real world that could use the sample – mostly games.

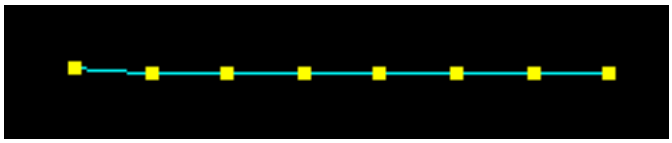
5.2 Mass-spring Rope

The first experiment is a 2D rope simulation. This simulation was motivated by a real problem on a game from Digital Chocolate [35] named Crazy Penguin Catapult[36]. The game is originally designed for mobile devices, by Digital Chocolate, and was ported to Flash to be playable on the web.

The Flash version of the game was created by Jynx Playware[45] where I had the chance to be part of the team that developed the game. The game consists of a war between penguins, which is controlled by the player, and bears. The bears kidnapped some of the penguins and now the penguin's army must rescue them all. The first part of each level has a catapult that throws the penguins to the fight against the bears. Each penguin swings attached to a rope and with the click of the mouse, it is catapulted.

A regular rope is easily created using mass-spring systems by just attaching a series of particles in a chain (the theory is based on the curve masses explained on the state of the art chapter). Like the following image:

Figura 26 – Rope modeled as a mass-spring system for the game



Source: Author

To create the desired behavior of the rope at least thirty particles were needed using a regular mass spring system . For a personal computer game, this number of particles is really low and wouldn't be a problem to create the simulation according to the game designer's requirements. However, the Flash runtime would only render eight to fifteen frames per second on high-end machine with incredible setups - much higher then the required by the client. This frame per second rate is unacceptable in modern games. This problem is even worse in Flash games since most of them are frame based, which means that players can take advantage of slower machines or even use intensive applications to purposely influence on the game's performance. This makes performance on Flash games crucial to maintain competition and proper gameplay.

Using the Dunas approach, we were able to create a very similar behavior using only eight particles. Using eight particles in the regular approach would give us a very

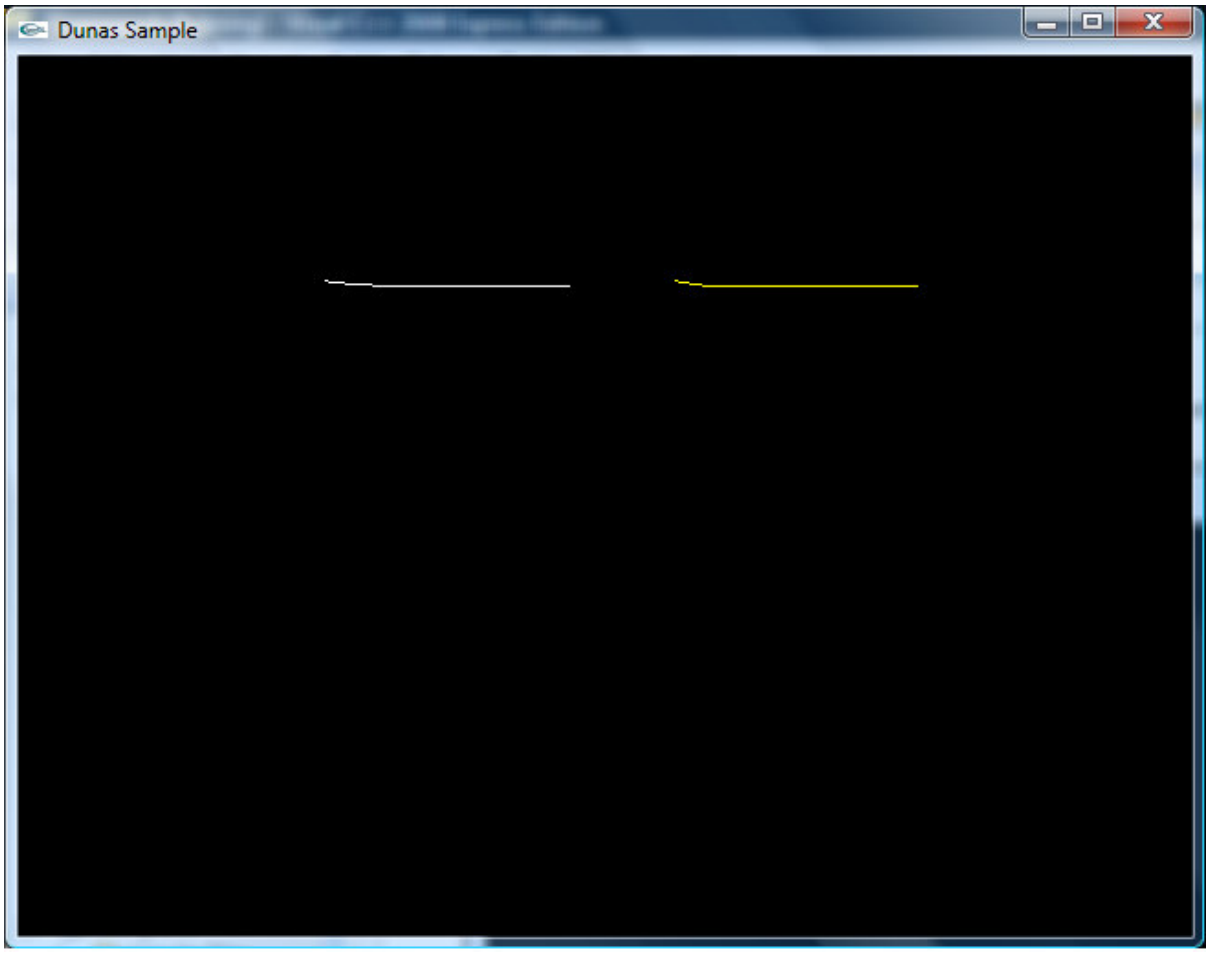
simple rope and the resulting image unacceptable for today's standards. However, the smoothing process of the deCasteljau algorithm creates very good images as shown in Figure 28.

Figura 27 – Rope in the game Crazy Penguin Catapult

Source: [36]

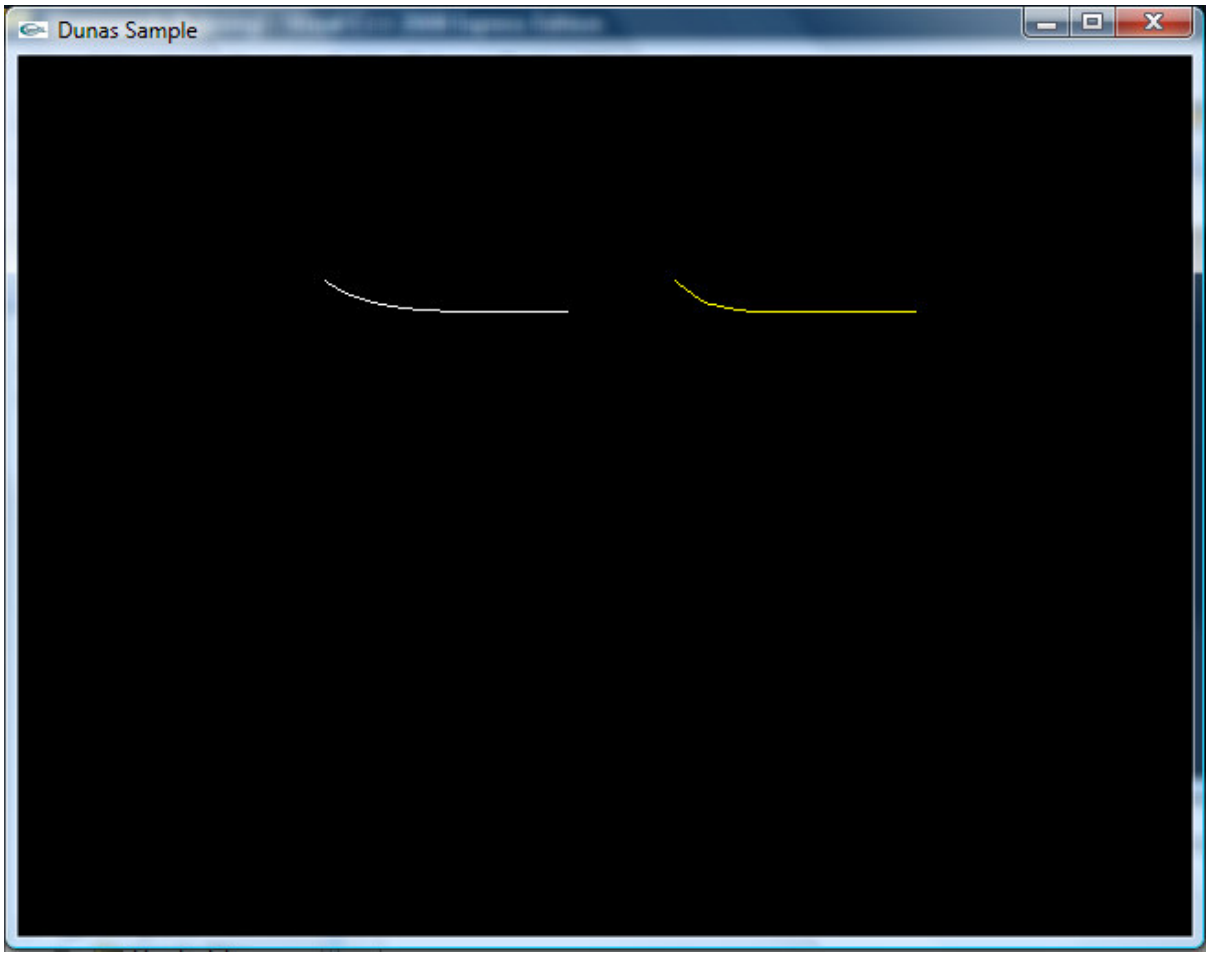
In summary, Dunas not only gives us a realistic dynamic simulation of the rope using mass spring system but, at the same time, provides very good visual results even using an inexpensive number of particles in the simulation. The following images, Figure 29 to 32, shows the difference between two ropes simulated using only 8 particles, with a regular approach (in yellow, on the right) and with Dunas approach (in white, on the left).

Figura 28 – Two ropes simulated with and without smoothing



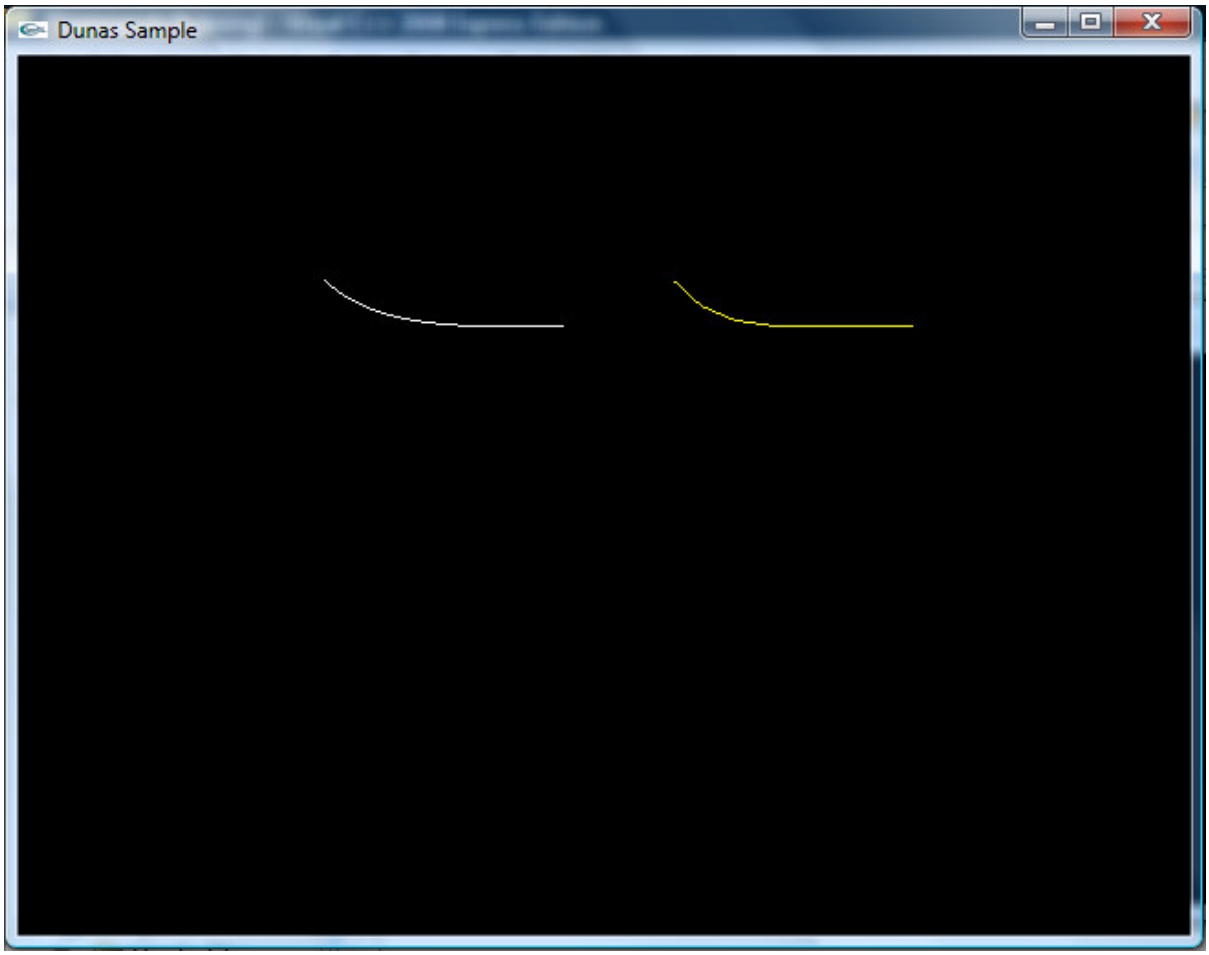
Source: Author

Figura 29 – Smoothing the rope process on the left rope (white)



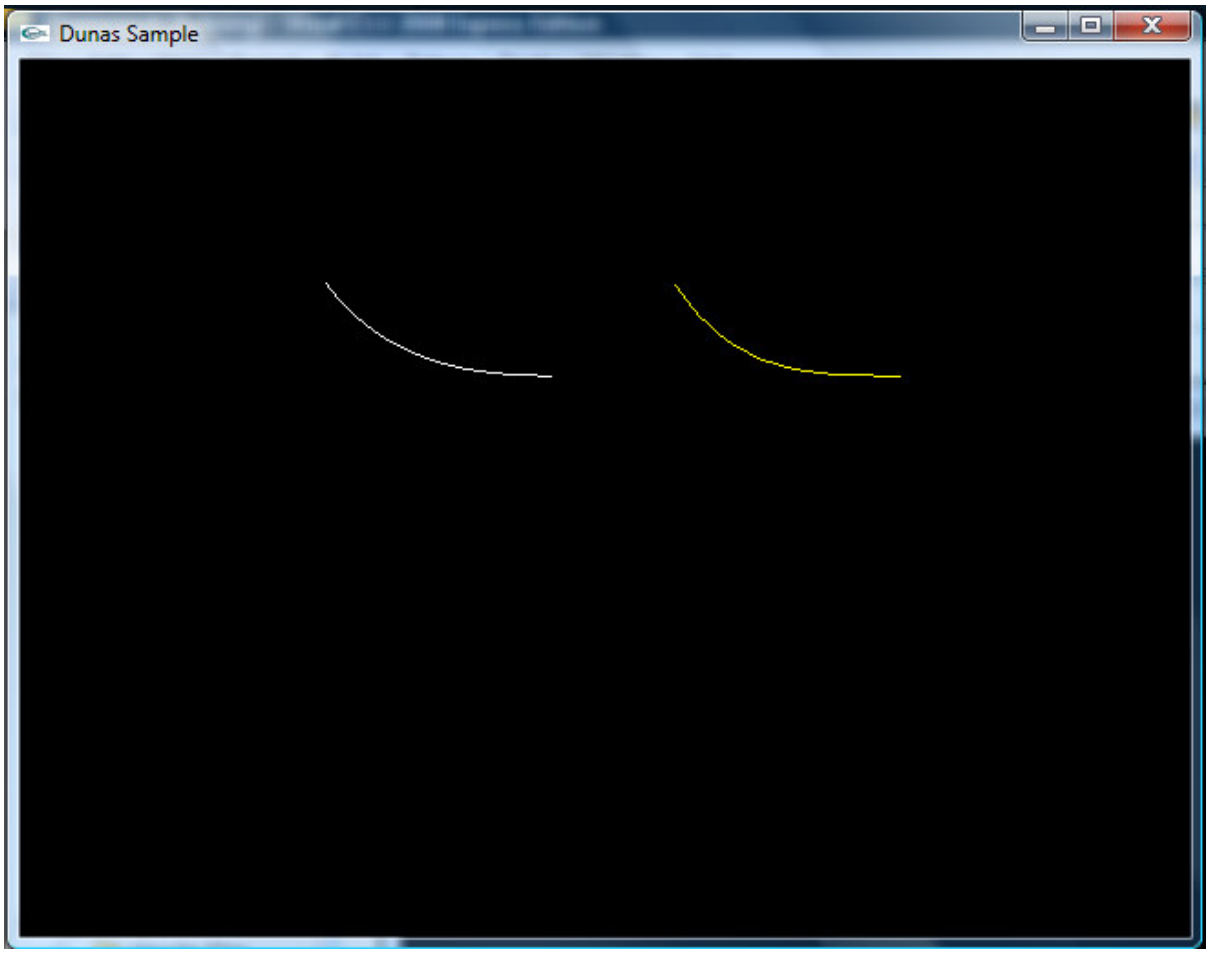
Source: Author

Figura 30 – Third screenshot of the smooth comparison



Source: Author

Figura 31 – Forth screenshot of the smooth comparison



Source: Author

A second Flash game developed by Manifesto Game Studio Ltda [47] was named Merlin's Adventure 3 [46]. In this game, Merlin is attached by a rope to Santa's sleigh that is been stolen by a gang of avocados. The rope in this game must use very few particles as explained before. The results are outstanding since we end up with a nicely rendered rope with realistic dynamics simulation as shown in Figure 33.

Figura 32 – Merlin's Adventures Game

Source: [46]

This game is a bit different from the above, the physics simulation is used as a gameplay factor and the correct behavior doesn't necessarily imitate a real world

rope. However, using the Dunas approach, the game designer was able to create a fun game and still get a smooth rope with a nice graphical result. The configuration used is shown in the Figure 33. This configuration is pretty non-intuitive and was achieved with a lot of extensive tests. This sample is a good example that, even in cases that we don't want realistic simulation, the Dunas approach doesn't affect the great ability about mass-spring system.

Figura 33 – Mass-spring setup for the Merlin's Game. All particles are connected to the first one. They are also connected to the nearest neighbors, as the usual.

Source: Author

5.3 Mass-Spring Cloth

The first experiment gave us incredible results for 2D objects in low processing power platforms. On the other hand, we wanted samples that could use the same approach and solve problems for games on personal computers. One of the most challenging aspects of today's games is the simulation of cloth. Some work has been done in the area and we follow the same approach presented in [39] that uses mass-spring system. However, we also use the Bézier techniques to smooth the surfaces and drop the number of particles needed for the simulation.

The whole implementation of this sample was presented in the last section of the previous chapter. Here we show the results and compare them, in terms of performance and graphics, to the original approach. The sample uses a rectangular mesh of control points to simulate physics and can be used as a model of sails in boats. As you can

see in Figures 34, 35 and 36, the results can be pretty good using only 64 (an 8x8 grid) particles. To achieve similar results without the smooth mechanism of the deCasteljau algorithms we need at least 400 particles (a 20x20 grid). The following table shows a comparison of the setups.

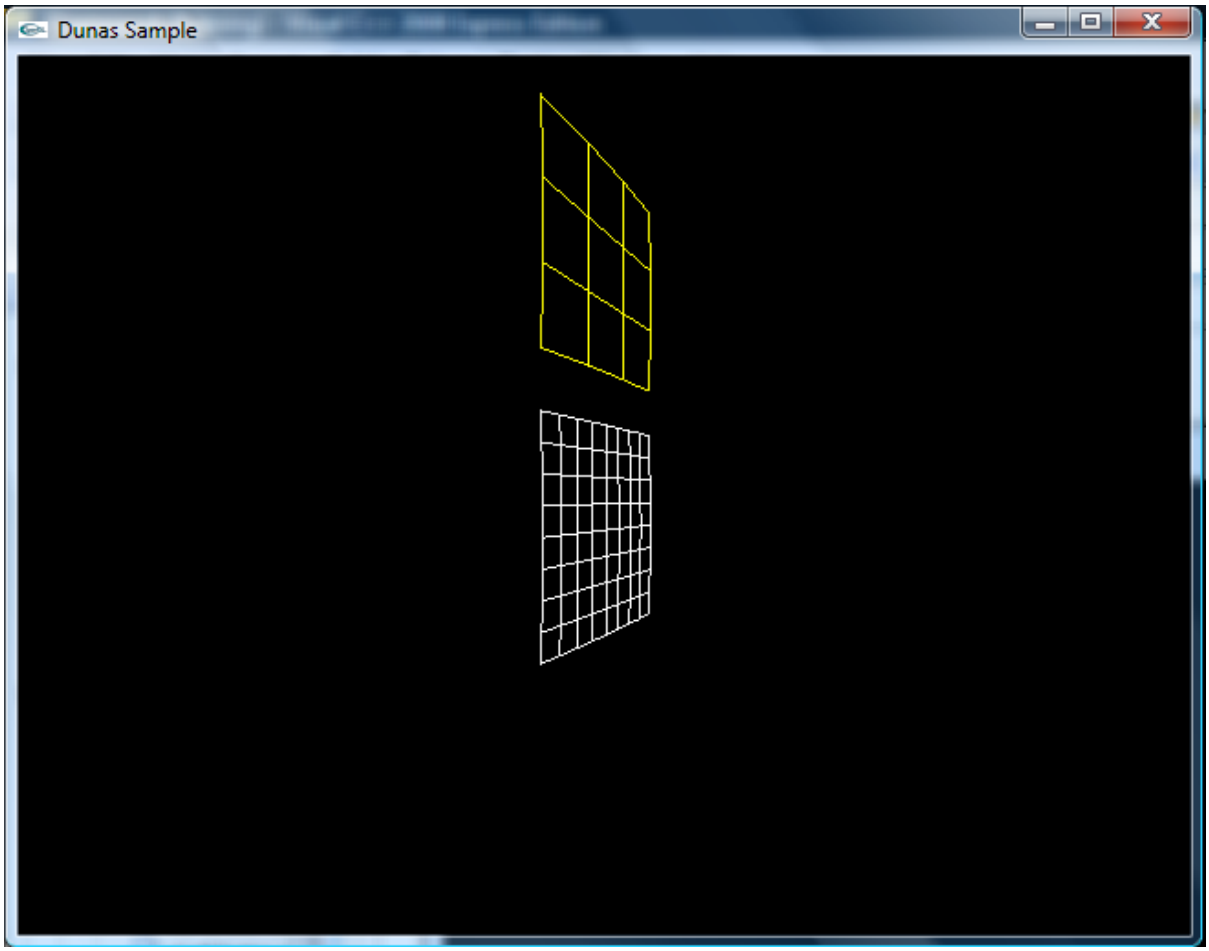
Tabela 1 – Comparison of regular techniques for cloth simulation and the dunas approach.

Technique	Particles	Connections
Regular	400	$\sim$3000
Dunas	64	$\sim$512

Source: Author

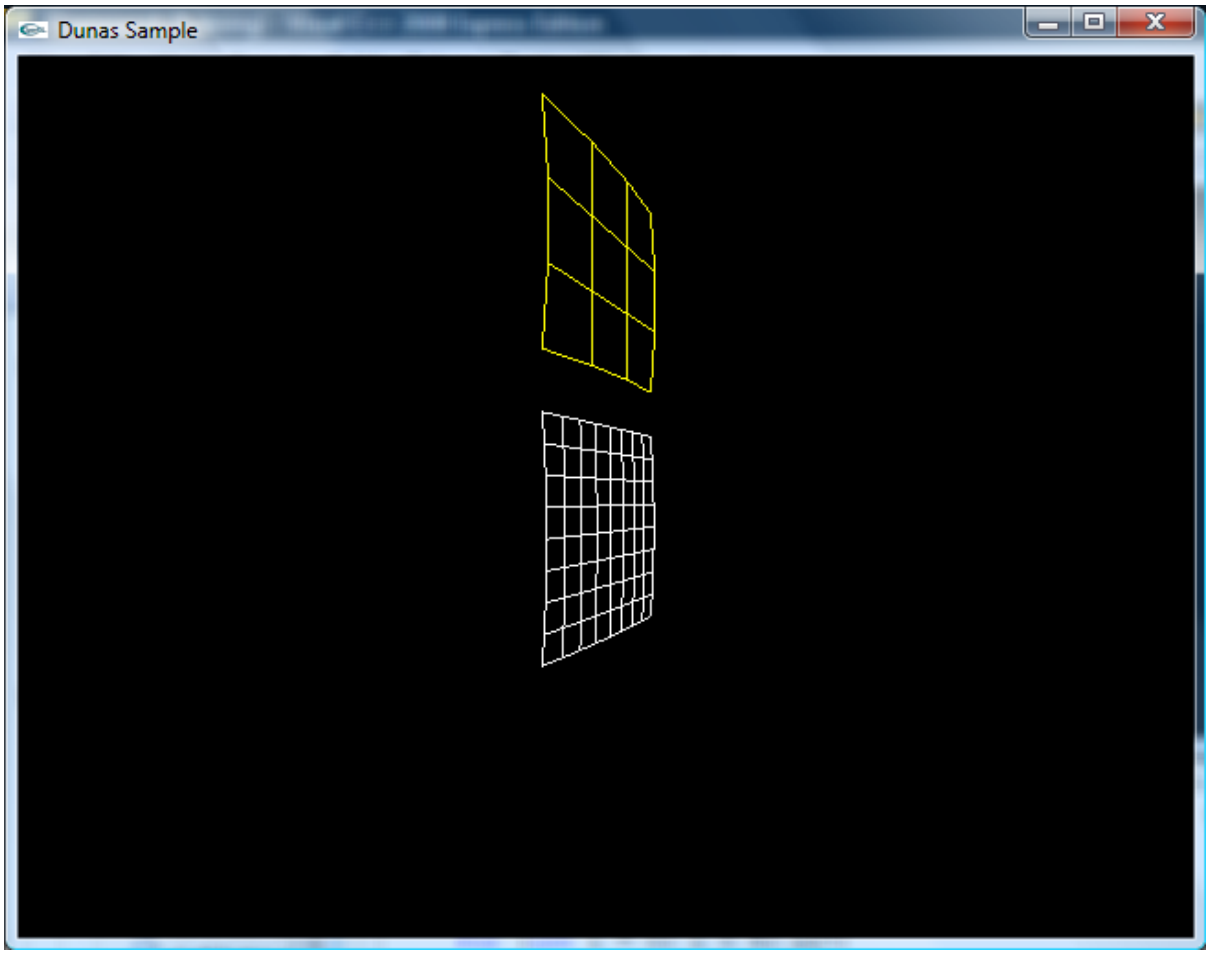
By the table becomes clear the improvement in performance we achieve using the Dunas approach. The Dunas framework is up to four times faster then the regular approach. This means that in a game scene with a ship battles we can have four times more ships in the sea using the cloth simulation for their ropes. This is a huge improvement if you think that instead of 25 boats, we can use 100. The Figure 38 shows a game, named 7Seas developed for the Computer Games course. The game achieves high frame rates, above 30 fps, with scene with more then 10 boats.

Figura 34 – Screenshot of the sail simulation running



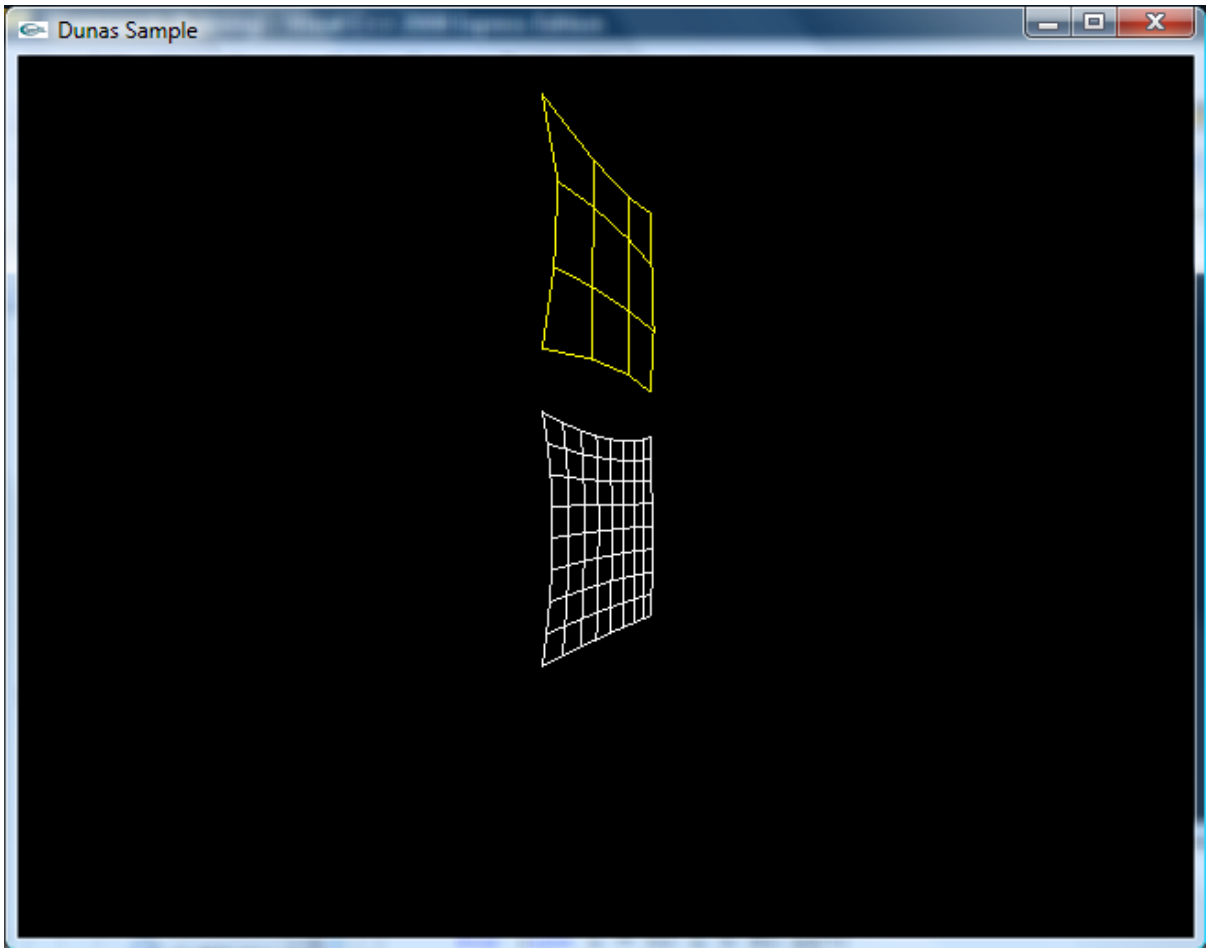
Source: Author

Figura 35 – Second screenshot of the sail simulation runnning



Source: Author

Figura 36 – Third Screenshot of the sail simulation running



Source: Author

Figura 37 – Sails running on the boat of a test game

Source: Author

5.4 Future Work

During the development of this work there were lots of interesting ideas to improve the usefulness of the Dunas framework. These ideas are presented here as suggestion for future works on the area.

Our model provides a simple yet effective method for realistic modeling of ropes and clothes in real time. The simulations look realistic and were used with success in a commercial project released to the public with tremendous success.

There is still a lot of room for improvement. The first step would be creating a good framework for collision detection with other objects and self-collision of rope and cloth. This would improve the realism for the dynamic simulation and the convex hull property of Bézier models is extremely helpful to this task.

Another open problem is the collision of objects with the cloth itself and not just the particles involved in the simulation. This is still an open problem in cloth simulation. The collision systems usually only take into account the particles involved in the simulation and small objects can easily pass through the cloth without being noticed.

This problem is especially important to our work since the number of particles involved in the simulation is even low.

Taking a more mathematical approach to the problem and actually adapt the physics algorithms to work on the control points and the result being the same as if it were acting on the surface itself. There would result in changing the equations of force and motion based on Bernstein equations from Bézier theory. This kind of work is being done for batch application and other computer aided areas using Bézier B-Splines, but not for real-time interactive applications with significant results yet.

Some issues are specific to the method we use to smooth the model. The Bézier method is not suitable for surfaces that need details, this is the case for the simulation of water, for example. To simulate water we need a lot of control points in the Bézier Surface so the details wouldn‘t be lost by the smoothing process. However, increasing the number of particles goes against the idea of using Dunas. Some other methods for surface smoothing could be tested and be used for other kinds of models, other than cloth and ropes.

Every year the hardware industry creates new pieces of dedicated hardware for personal computers. Since 1999 Graphics Processing Units have been widely used in computer games, especially after the introduction of programmable pipelines for these cards. One of the major players on the industry, nVIDIA, just released Physically Processing Units and the idea is to improve the support of physics on real-time applications. In this context, it is necessary to adapt all algorithms that can benefit from this hardware of specific purpose. So, a good extension to the Dunas framework would be to use GPUs and PPUs to process some of the algorithms.

The framework created during the course of this work was implemented using the C++ programming language and the Revolution Engine [4]. However, the techniques used here can be useful in a handful of platforms that these technologies don’t work such as Flash Games, J2ME Games, iPhone Games, etc. The port of the framework to other languages would make possible the physics mechanics it can provide, as discussed in the previous section.

Dunas is just a little part of a whole game engine. The Revolution Engine started as a graduation project and still has lots of improvements and coding to be done. Using the undergraduate students to implements some of the algorithms for both parts of the engine would be a good way to evaluate the student but to create knowledge inside the University [6].

New improvements added to Dunas are intended to be reported in the Revolution project website (www.cin.ufpe.br/∼mtcfa/Revolution), which also provides a link to download the binaries of the engine and sample codes. The 7Seas Games can also be

downloaded from the same webpage.

(1, 2, 3, 4, 5, 6)

REFERENCES

[1] KASS, M.; WITKIN, A.; TERZOPOULOS, D. Snakes: Active contour models. International Journal of Computer Vision, 1, 4, 321-331, 1987.

[2] KAVAN, L.; ZÁRA, J. Real Time Skin Deformation with Bones Blending. In: WINTER SCHOOL OF COMPUTER GRAPHICS, 2003, Plzen, WSCG SHORT PAPERS proceedings. Union Agency Science Press, 2003.

[3] BARAFF, D.; WITKIN, A. Large Steps in Cloth Simulation. In: INTERNATIONAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, 1998, United State, Proceedings of the 25th annual conference on Computer Graphics and interactive techniques, New York: ACM, 1998.43-54

[4] ALBUQUERQUE, M. Revolution Engine: 3D Game Engine Architecture (in Portuguese), BS conclusion paper, Federal University of Pernambuco, 2005;

[5] ROCHA, E. Forge 16V: An Isometric Game Development Framework (in Portuguese), MSc dissertation, Federal University of Pernambuco, 2003;

[6] MADEIRA, C. FORGE V8: A Computer Games and Multimedia Applications Development Framework (in Portuguese), MSc dissertation, Federal University of Pernambuco, 2003;

[7] Microsoft.com, The Xbox Console, http://www.microsoft.com/xbox; Visited in: December, 1st 2008

[8] Playstation.com, PlayStation Console, http://www.us.playstation.com; Visited in: December, 1st 2008

[9] OpenGL.org, About the OpenGL Architecture Review Board, http://www.opengl.org/about/arb/overview.html; Visited in: December, 1st 2008.

[10] OpenGL.org, GLUT - The OpenGL Utility Toolkit, http://www.opengl.org/resources/libraries/glut.html; Visited in: December, 1st 2008.

[11] FARIN, G. Curves and Surfaces for Computer Aided Geometric Design. Academic Press, 1990.

[12] GIBSON, S.; MIRTICH. A survey of deformable modeling in computer graphics. Technical Report TR97-19, MERL Technical Report, 1997.

[13] Popcap.com, Popcap Games, http://www.popcap.com/; Visited in: December, 1st 2008.

[14] Bigfish.com, Bigfish Games, http://www.bigfishgames.com/; Visited in: December, 1st 2008.

[15] RealArcade.com, Real Arcade, http://brazil.real.com/games/lp/?src=realarcade&tps=br_; Visited in: December, 1st 2008.

[16] Adobe.com, Adobe Flash Player, http://www.adobe.com/products/flashplayer/; Visited in: December, 1st 2008.

[17] OSHITA, M.; MAKINOUCHI, A.; Real-time Cloth Simulation with Sparse Particles and Curved Faces, Computer Animation 2001, 8 pages, November 2001.

[18] RUDOMÍN, I.; CASTILO, J. Real-Time Clothing: Geometry and Physics. In: WINTER SCHOOL OF COMPUTER GRAPHICS, 2002, Plzen, WSCG SHORT PAPERS proceedings. Union Agency Science Press, 2002.

[19] RUDOMÍN, I., MELÓN, M.A.: Multi-Layer Garment Using Hybrid Models, Visual 2000 Proceedings, pp.118-128, 2000.

[20] Java.sun.com, The Java ME Platform, http://java.sun.com/javame/index.jsp/; Visited in: December, 1st 2008.

[21] Uml.org, Object Management Group – UML, http://www.uml.org; Visited in: December, 1st 2008.

[22] GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995;

[23] MOORE, P.; MOLLOY, D; A Survey of Computer Based Deformable Models. In: Machine Vision and Image Processing Conference, 2007. Kildare, IMVIP Proceedings, 5566, 2007.
[24] HERTZMANN, A.. Machine Learning for Computer Graphics: A Manifesto and Tutorial. Proc. Pacific Graphics 2003. Invited Paper. Banff, Alberta. October, 2003. pp. 2236.
[25] QIN, H.; TERZOPOLOUS, D.; D-NURBS: A Physics-Based Framework for Geometric Design. In: IEEE Transactions on Visualization and Computer Graphics, VOL. 2, NO. 1, March 1996.
[26] SEDEBERG, T. W.; PARRY, S. R.; Free-form deformation of solid geometric models. In: Proceedings of SIGGRAPH 1986, pages 151–160, 1986.
[27] HUNTER, P.; FEM/BEM notes 2005. Available in http://www.qoop.com/schoolbooks/viewpdf.php?pdfurl=http://SchoolLibrary.com/Members/ Math_eBook_Collection/Finite_Element_Method__Boundary_Element_Method.pdf&title=FEM/BEM+NOTES Visited in: December, 1st 2008.
[28] NEALEN, A.; MÜLLER, M.; KEISER, R.; BOXERMAN, E.; CARLSON, M.; Physically Based Deformable Models in Computer Graphics. Journal compilation 2008 The Eurographics Association and Blackwell Publishing Ltd. Volume 25 Issue 4, Pages 809 - 836, 2008.
[29] ROTH, S.; GROSS, M.; TURELLO, S.; CARLS, F.; A Bernstein-Bézier Approach to Soft Tissue Simulation. In: EUROGRAPHICS, 1998, Proceedings of Eurographics 98, Blackwell Publishers, Volume 17, Number 3, 1998.
[30] SCHELKLE, E; REMENSPERGER, R.; Integrated Occupant-Car Crash Simulation With the Finite Element Method: the Porsche Hybrid Iii-Dummy and Airbag Model. In: International Congress & Exposition, 1991. Detroit, MI.
[31] TERZOPOULOS, D.; QIN, H.; Dynamic NURBS with geometric constraints for interactive sculpting. ACM Transactions on Graphics, Volume 13, Issue 2, April 1994. 103136.
[32] QIN, H.; FEM-Based Dynamic Subdivision Splines. In: Pacific Conference on Computer Graphics and Applications. Proceedings of the 8th Pacific Conference on Computer Graphics and Applications. 2000, IEEE Computer Society Publisher. 184
[33] HUI, C.; HANQIU, S.; XIAOGANG, J.; Interactive Haptic Deformation of Dynamic Soft Objects. In: CM International Conference on Virtual Reality Continuum and Its Applications VRCIA. Proceedings of the 2006 ACM international conference on Virtual reality continuum and its applications. 2006. 51-57.
[34] VLACHOS, A.; PETERS, J.; BOYD, C.; MITCHEL, J.; Curved PN Triangles, Proc. of the 2001 ACM Symposium on Interactive 3D Graphics, 2001. (http://alex.vlachos.com/graphics/)
[35] DigitalChocolate.com, Digital Chocolate, http://www.digitalchocolate.com/; Visited in: December, 1st 2008.
[36] DigitalChocolate.com, Crazy Penguin Catapult, http://www.digitalchocolate.com/games/pc/crazy-penguin-catapult.html; Visited in: December, 1st 2008.
[37] PROVOT, X.; Deformation Constraints in a Mass-Spring Model to Describe Rigid Cloth Behavior, In Proc. Graphics Interface '95, pp 147-154, 1995.
[38] HOUSE, D.; BREEN, D.; Cloth Modeling and Animation. s.l.: A.K. Peters, 2000.
[39] Gamasutra.com. Devil in the Blue-Faceted Dress: Real-Time Cloth Animation. Visited in: December, 1st 2008.

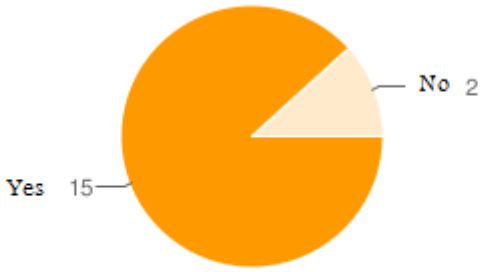
[40] Fresh3D.com. A unique architecture for unique results. http://www.fresh3d.com/fresh_engine/overview.php. Visited in: December, 1st 2008.
[41] WATT, A.; POLICARPO, F.; 3D Games: Animation and Advanced Real-time Rendering. Harlow, Addison-Wesley, 2003.
[42] Microsoft.com. Visual C++ 2008 Express Edition, http://www.microsoft.com/express/vc/. Visited in December, 1st 2008.
[43] Microsoft.com. Visual C++ 2008 Express Edition Download, http://www.microsoft.com/express/download/. Visited in December, 1st 2008.
[44] MrMoen.com, OpenGL with Visual C++ 2008 Express Edition. http://www.mrmoen.com/2008/03/30/opengl-with-visual-c-express-edition/. Visited in November, 24th 2008.
[45] Jynx.com.br. Jynx Playware. http://www.jynx.com.br. Visited in December, 1st 2008.
[46] Manifestogames.com.br. Merlin Adventures. http://manifestogames.com.br/?page=game&name=merlin3, Visited in December, 1st 2008.
[47] Manifestogames.com.br. Manifesto Game Studio. http://www.manifestogames.com.br/, Visited in December, 1st 2008

APÊNDICES

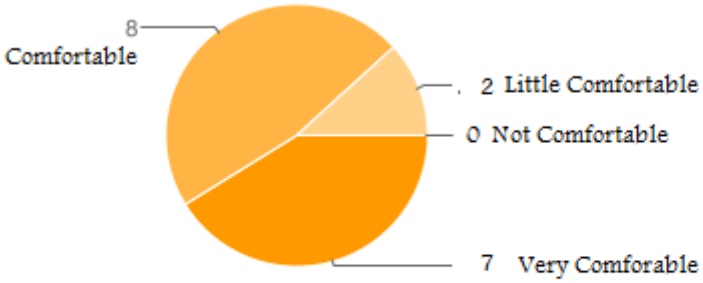
Appendix A – NURBS AND BÉZIER POOL

This appendix shows the pool with professional designers in Recife regarding the utilization of Bézier curves and surfaces and NURBS. This pool was made with seventeen designers that work professionally in the area of game development in Recife. The original pool, in Portuguese, can be found at http://spreadsheets.google.com/viewform?key=p8ohC-Au_bD-x6geoThYEfA.

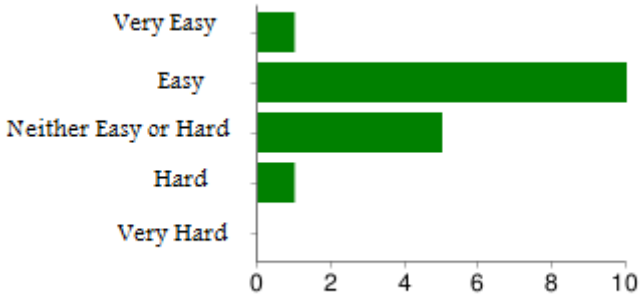
1. Do you know the term Bézier Models?



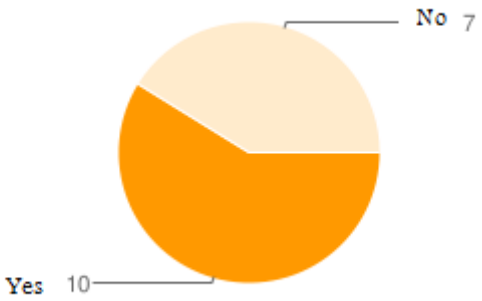
2. How comfortable do you feel about using Bézier Models?



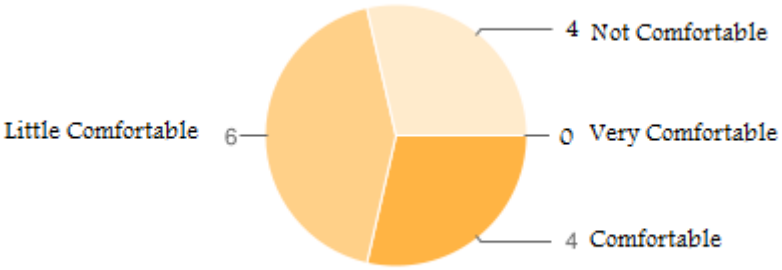
3. How would you classify Bézier Models?



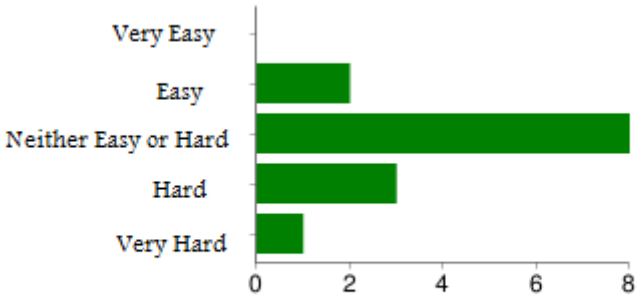
4. Do you know the term NURBS?



5. How comfortable do you feel about using NURBS?



6. How would you classify NURBS?



7. Which one would you use more?