UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

LEONARDO GARCIA FISCHER

3DS-BVP: A Path Planner for Arbitrary Surfaces

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Profa. Dra. Luciana Nedel
Advisor

Porto Alegre, November 2011

CIP – CATALOGING-IN-PUBLICATION

Fischer, Leonardo Garcia

3DS-BVP: A Path Planner
for Arbitrary Surfaces / Leonardo Garcia Fischer. – Porto Alegre:
PPGC da UFRGS, 2011.

117 p.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul.
Programa de Pós-Graduação em Computação, Porto Alegre, BR–
RS, 2011. Advisor: Luciana Nedel.

1. 3D path planning. 2. Motion planning. 3. Potential fields.
4. Laplace's equation. I. Nedel, Luciana. II. Título.

“If you wanna make the world a better place
Take a look at yourself and then make a change”
Man In The Mirror
— MICHAEL JACKSON

AGRADECIMENTOS

Antes de tudo, preciso agradecer à minha família pelo suporte dado. Sem o suporte deles, não teria chegado até este ponto. Obrigado ao Vilson e Regina, meus pais, e ao Victor e Beatriz, meus irmãos. Em particular, preciso agradecer ao Victor por ter suportado dividir o quarto comigo enquanto ele precisava dormir cedo para acordar às 5h da madrugada, enquanto eu dormia depois da 1h para concluir algum trabalho.

Um obrigado muito especial à Francele, minha namorada, por ter dividido comigo grandes momentos e me dado forças quando eu mais precisava. Nestes 6 anos de namoro, ela foi mais do que uma namorada, foi a amiga e companheira que me confortava e alegrava.

A todos os amigos e companheiros que fiz nos laboratórios do grupo de Computação Gráfica da UFRGS. O grupo é grande, então vou citar apenas os nomes (em ordem alfabética) que fizeram uma contribuição significativa neste trabalho: Leandro Augusto Frata Fernandes (Tocha), Marilena Maule (Lenna) e Vitor Fernando Pamplona. Se eu fosse citar os nomes do grupo de quem contribuiu na minha vida e desenvolvimento pessoal, esta lista com certeza seria muito maior.

Mas dois nomes do grupo de Computação Gráfica precisam ser agradecidos com especial destaque: (i) Luciana Nedel, minha orientadora, que me ajudou muito durante esta jornada; (ii) Renato Silveira, cujas ideias foram os principais motivadores que levaram ao desenvolvimento deste trabalho.

E, finalmente, ao Instituto de Informática da UFRGS e o CNPq pela bolsa de mestrado, auxílios nas viagens para apresentações de artigos e a visita ao Indian Institute of Technology Delhi – IIT-Delhi. Todos estes recursos foram essenciais para que eu chegasse até aqui.

A todos, muito obrigado,

Leonardo Garcia Fischer

CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS

BVP	Boundary Value Problem
CAD	Computer-aided Design
CDT	Constrained Delaunay Triangulation
DCEL	Doubly-Connected Edge List
DOF	Degree-of-freedom
GPU	Graphical Processing Unit
PRM	Probabilistic RoadMap
RRT	Rapidly-Exploring Random Tree

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Efficient path planning methods are being explored along the years to allow the movement of autonomous robots or virtual agents. Basically these algorithms search the environment for a path with low probability of collision with obstacles that conduces the agent from an initial to a goal position. Although the first path planning algorithms to compute routes in graphs were presented more than 50 years ago, there is still a lot of effort into improving the current approaches.

The current path planning algorithms usually assume that the environment can be easily projected on a plane. There are also other algorithms that can easily deal with higher dimensional spaces. But a class of environments that cannot be easily treated by current algorithms is the one composed by arbitrary surfaces. These surfaces, with holes and bends for instance, cannot be directly projected on a plane. Because the path must be on the surface, it has only 2 degrees of freedom in any point of the surface, which is not trivial to map for a higher dimensional path planning algorithm.

This work presents a new technique for path planning on 3D surfaces called 3DS-BVP. This new path planner is based on a previous path planning algorithm for 2D environments. The former algorithm, called BVP-Path-Planner, uses Boundary Value Problems (BVP) and harmonic functions to generate potential fields. By following the gradient descent of these potential fields, it is possible to produce smooth paths free from local minima from any point of the environment to a given goal position.

Our algorithm generates a potential field directly on the 3D surface using a numerical method inspired on the one used by the BVP-Path-Planner. The 3DS-BVP works over complex surfaces of arbitrary genus or curvature, represented by a triangle mesh, without the need of 2D parametrizations.

Our results demonstrate that our technique can generate paths with similar quality as those generated by the BVP-Path-Planner in planar environments. The same algorithm is also able to generate paths in arbitrary surfaces at interactive frame rates.

Keywords: 3D path planning, motion planning, potential fields, Laplace's equation.

3DS-BVP: Um Planejador de Caminhos para Superfícies Arbitrárias

RESUMO

Métodos eficientes para planejamento de caminhos têm sido explorados ao longo dos anos para permitir movimento de robôs autônomos ou agentes virtuais. Basicamente, estes algoritmos buscam pelo ambiente por um caminho com pouca probabilidade de colisão com obstáculos, e que conduza o agente de uma posição inicial para uma posição objetivo. Apesar de os primeiros algoritmos para planejamento de caminhos para calcular rotas em grafos foram apresentados a mais de 50 anos atrás, ainda existe muito trabalho sendo realizado para melhorar as técnicas existentes hoje em dia.

Os algoritmos de planejamento de caminhos atuais normalmente assumem que o ambiente pode ser facilmente projetado em um plano. Também existem diversos algoritmos que podem trabalhar facilmente com mais dimensões. Porém, uma classe de ambientes que não podem ser facilmente tratadas pelos algoritmos atuais é composta por superfícies arbitrárias. Estas superfícies, com buracos e torções, por exemplo, não podem ser facilmente projetadas em um plano. O fato de o caminho ser restrito à superfície faz com que, em um dado ponto o algoritmo precise calcular um caminho 2D em uma superfície 3D, o que não é trivial de mapear em um algoritmo de planejamento de caminhos para várias dimensões.

Este trabalho apresenta uma nova técnica de planejamento de caminhos em superfícies 3D, chamada 3DS-BVP. Este novo planejador de caminhos é baseado em um algoritmo de planejamento de caminhos para ambientes 2D. O algoritmo anterior, chamado BVP-Path-Planner, utiliza problemas de valor de contorno (*Boundary Value Problems*, BVP) e funções harmônicas para gerar campos potenciais. Ao seguir o gradiente descente destes campos potenciais, é possível produzir caminhos suaves livres de mínimos locais, partindo de qualquer posição do ambiente para um dado objetivo.

Nosso algoritmo gera um campo potencial diretamente na superfície 3D utilizando um método numérico que foi inspirado por aquele utilizado no BVP-Path-Planner. O 3DS-BVP trabalha sobre superfícies complexas com buracos ou curvaturas, representadas por malhas de triângulos, sem a necessidade de parametrizar a superfície em uma representação 2D.

Nossos resultados demonstram que a técnica pode gerar caminhos com qualidade similar àqueles gerados pelo BVP-Path-Planner em ambientes planos. O mesmo algoritmo é capaz de gerar caminhos em superfícies arbitrárias a taxas de atualização interativas.

Palavras-chave: Planejamento de caminhos em 3D, planejamento de movimentos, campos potenciais, Equação de Laplace.

1 INTRODUCTION

A problem faced in our everyday life's is how to go from a given known location to a new one that we never had before. If these places are very far one from one another (in a city scale, for example) we may ask for directions or consult a map of the region. If we know that they are close one from the other (in the same building, for example), we simply walk around to learn the environment until we find were the new location is.

No matter the scale of the problem or what strategy we use to solve it, we say that we need to plan how to navigate from one initial place to another. This problem is also faced very often within computational systems. For example, this happens when a robot needs to move inside an environment or the artificial intelligence inside a video game needs to control an enemy during a match.

Navigation is a complex interactive task. According to Bowman (BOWMAN et al., 2004), it is divided into two subtasks: *wayfinding* and *travel*. Wayfinding precedes the travel by being the cognitive component of the navigation. It includes the high-level thinking, planning and decision making by the individual who is making the navigation. Wayfinding also requires knowledge about the localization of obstacles and what are the limits of the navigable regions, which constitute the spatial understanding of the environment. By thinking and taking decisions over the understanding of the environment, the individual determines a path from its current location to the goal location.

Using the path determined during the wayfinding task, the individual starts the travel task, which is the motor component of navigation. It includes the low-level actions that one takes to control its position and gaze in order to follow the defined path. It also includes the handling of changes in the environment, such as avoid collision with dynamic obstacles.

Although navigation is an everyday task that human beings solve quite naturally, mapping it to a computational problem is not a trivial task. Path planning algorithms are being explored for years. Several solutions were effectively applied into robotics and virtual environments. Most of these solutions focuses on providing a high performance path finding, by including a preprocessing phase on the algorithm (CALOMENI; CELES, 2006). There are also solutions that focuses on solving the problem with better paths, providing ways to minimize the probability of collision with obstacles (KALLMANN, 2010) or improving the credibility of the generated paths within a simulation (KANG; KIM; KIM, 2010).

Navigation on three-dimensional surfaces is a relevant problem for many application areas, such as: scientific visualization, where a user needs to inspect different objects, such as organs in a medical application or engines in a CAD system; robotics, with the automatic definition of paths and motions for autonomous robots; and entertainment, more specifically on the video games domain, where the exploration of complex 3D worlds are

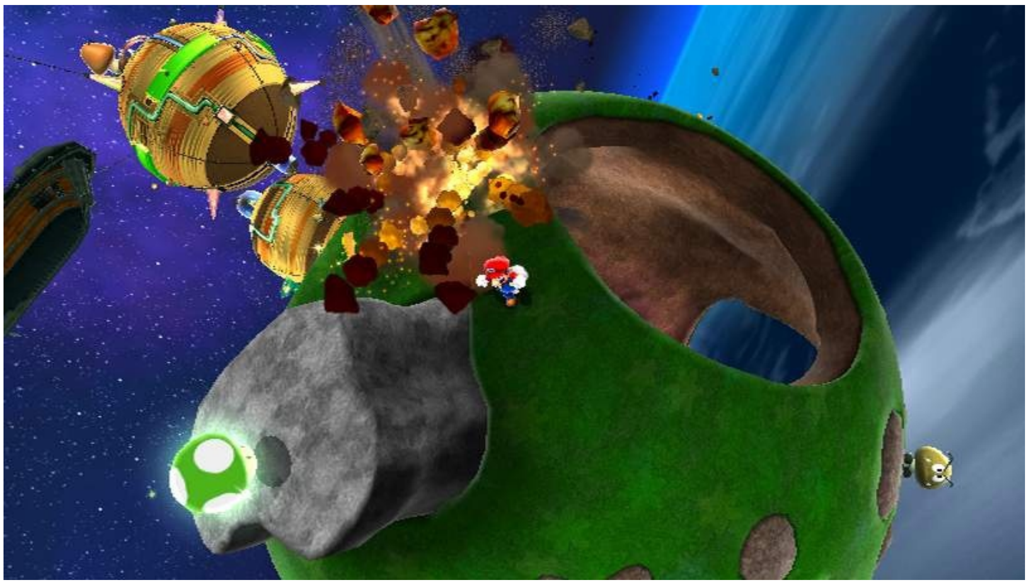


Figure 1.1: A snapshot of the video-game Super Mario Galaxy® (NINTENDO, 2007). This picture shows a level composed of a very small planet, with several holes in it. The character, as well as the enemies, can walk freely over the surface of the planet.

much more challenging for the player than planar ones.

Most of the current path planning algorithms make efforts to solve the problem on a 2D representation of the environment. These solutions usually project the obstacles on the Euclidean plane, and assume that the areas uncovered by projected obstacles represent the navigable space. Then the navigable space is discretized (usually into a graph, grid, or triangulation). Finally, a dedicated algorithm searches in this discretization for a sequence of free positions that connects the initial and the goal positions.

Some of the current path planning algorithms are robust enough to handle systems with more than two degrees of freedom – DOFs – by supporting representations of the environment with higher dimensionality. For example the work of Carsten et al. (CARSTEN; FERGUSON; STENTZ, 2006) uses 3D regular grids to discretize the environment, and a graph search algorithm (STENTZ, 1995) is used to find a set of contiguous cells to draw the 3D path. Other algorithms support several DOFs by using some probabilistic approach to sample the free space. This is usually done to compute paths for robotic arms with several joints (BELGHITH et al., 2006).

However, path planning methods restricted to arbitrary surfaces are not well explored in the literature. This problem consists in finding a path connecting two points on a surface, where the path does not interpenetrate neither *jumps* from one point of the surface to another. One example of this problem would be the control of virtual cameras in a CAD or modeling application (Figure 1.2), in order to help the evaluation and visualization of the 3D models. Another example would be in virtual environments, where virtual agents would navigate over those kind of surfaces. One example of these kind of virtual environment is in a video-game called Prey® (3D Realms, 2006), where the player and the enemies can walk freely over several walls, ceilings and footbridges, as well as on the floor. Figure 1.1 illustrates another of these video-games where the player and his enemies walk over arbitrary surfaces.

Methods that focus specifically on 2D path planning cannot be trivially modified to

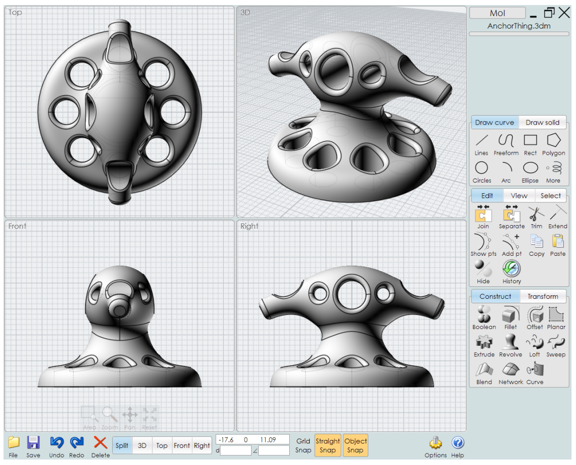


Figure 1.2: A snapshot of a CAD software called *Moment of Inspiration 3D* (MOI3D, 2011). Such software could benefit from new tools that, for example, control the camera over the surfaces that the designer is working on.

handle arbitrary surface cases. For these, one possible approach would be the use of sophisticated projection techniques from the 3D surface to the Euclidean plane, adapting the 2D path planner to work on this projection. But this is not a trivial task, due to the nature of the projection. Depending on the complexity of the surface, it would be necessary to "break" the surface in several pieces. This may introduce gaps on the projection of these pieces on the 2D plane that must be correctly handled by the algorithm.

Algorithms that handle 3D environments depend on their nature to be adapted to 3D surfaces, as in a given point of the surface the algorithm should have only 2 degrees of freedom. Graph-based approaches are fast enough for real-time applications, but the generated paths are not as smooth as the ones produced by other approaches. Techniques that use grid representations of the volumetric environment would require a very high number of cells to represent the surface, thus jeopardizing the algorithm performance and memory consumption. In all these cases, the required work for porting is not negligible, and it is not clear how these algorithms will behave in this kind of environments.

In this work we developed a solution for the first part of the navigation task, the wayfinding. We present a path planning algorithm that handles the arbitrary 3D surface case, so called *3DS-BVP*, an acronym for *3D Surface Path Planner using Boundary Value Problems system*. By using a triangle mesh discretization of the surface, the proposed algorithm develops a boundary value problem (BVP) system that produces a potential field on this mesh. Using the gradient of the potential field, a path on this surface can be easily computed from any point to a given goal. Thus, the main contributions of this work are:

- An approximation to the Laplace's Equation using a triangular discretization;
- A numerical method that generates potential fields with similar characteristics to those generated by the Laplace's Equation in arbitrary surfaces;
- A strategy to apply the numerical method to compute paths on 3D triangular surface meshes with holes and bends.

We developed this method based on the method proposed by Silveira et al. (SILVEIRA et al., 2009), which is based on harmonic functions. The path planner by Silveira et al. uses the solution of a modification of the Laplace's equation in a regular planar grid to generate the potential field, which generates the paths through its gradient descent. Our algorithm was based on it in the sense that we also developed a numeric method that mimics the results produced by the Laplace's equation, but using a triangle mesh discretization.

For planar cases our algorithm was able to reproduce the paths generated by the algorithm developed by Silveira et al., despite its dependency on the quality of the generated paths. Concerning surfaces of arbitrary genus or curvature, our algorithm is also able to generate smooth paths. Performance tests show that our algorithm is able to run at interactive frame rates.

The remainder of this Thesis is organized as follows. Chapter 2 presents the related work on path planning for interactive applications and robotics. Chapter 3 presents the main concepts behind the path planner presented by Silveira et al. Chapter 4 explains the 3DS-BVP formulation and algorithm. Chapter 5 shows the results achieved in this work and some discussion about it, while Chapter 6 presents the degenerated cases and other limitations found in our algorithm. Finally, in Chapter 7 conclusions and future works are discussed.

2 RELATED WORK

In this work we will assume that *path planning* refers to an algorithm that finds a collision-free path (or check that no one exists) between a *start* and a *goal* positions in an environment, based on its geometry and obstacles (HSU; LATOMBE; MOTWANI, 1997). The generated path will be followed by an *agent*, which will represent the virtual entity (an avatar or a virtual camera) or an autonomous robot in the real world. The set of possible positions that an agent can assume is usually called the *configuration space*. Although there are algorithms that deal with exploratory path planning (BROCK; GRUPEN, 2003) and dynamic environments (BELGHITH et al., 2006) (TREUILLE; COOPER; POPOVIć, 2006), in this work we assume that the environment is static and already known when the path planning algorithm is required.

Path planning algorithms are being actively explored for at least the last 20 years. Many different approaches have been proposed to solve this problem, and each one has its particular advantages and disadvantages.

Our technique uses potential fields computed over a triangle mesh to produce paths over arbitrary surfaces. Based on this perspective, in this chapter we will review the most relevant work on path planning algorithms, classifying them according to the different approaches used to solve the problem.

2.1 Graph based path planning

Due to its performance and low memory requirements, graph-based approaches are the most common kind of path planning algorithms found in the game industry. Popular game engines, such as the Unreal Engine® and CryEngine®, made use of it. The characteristics of graph based path planning made these algorithms useful in other areas than robotics and agent simulations, like the routing of data packages in network systems (TANENBAUM, 2002).

In these methods, the environment is discretized into a graph. In this graph, the nodes are associated with sites in the environment, and an edge represents a free path connecting these sites. Figure 2.1 illustrates these concepts. To produce a path, the agent needs to specify the start and goal sites of the environment, and then make a search for a sequence of nodes of the graph that connect both sites. The Dijkstra algorithm (DIJKSTRA, 1959; CORMEN et al., 2001) is used to find a path between two nodes of this graph. Heuristic based approaches, like the A* algorithm (HART; NILSSON; RAPHAEL, 1968), are also used in order to improve the performance of the algorithm. While classical work on path planning usually call these graphs as *roadmaps*, game engines usually call it as *navigation meshes* (VALVE, 2006). In this work we will refer to them as roadmaps.

There are several graph-based approaches for path planning, and most of the differ-

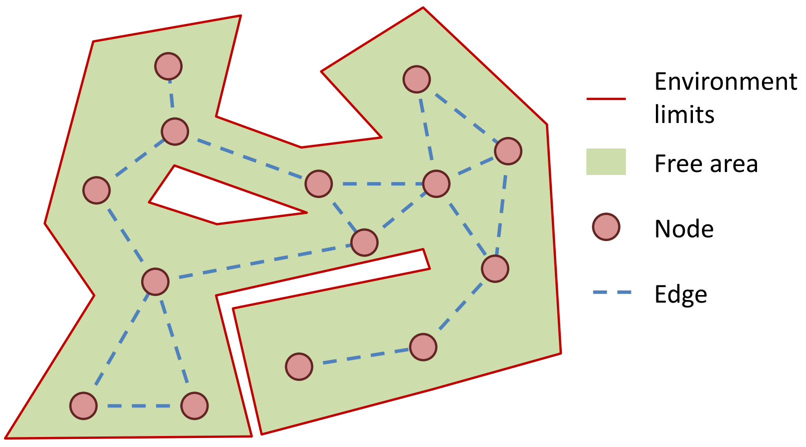


Figure 2.1: Example of an environment and a roadmap representing it. The nodes represents relevant free positions of the environment, while the edges represent valid paths between these positions.

ences between them are the methods used to sample the free areas of the environment into a graph. All of them try to reach an optimal relation between the number of nodes in the graph and the coverage of the environment. The graph representation implicitly requires that, to increase the coverage of the environment, the number of nodes also needs to increase. This has a direct impact on the performance of the search algorithm.

One of the simpler approaches to sample the free space of the environment is to use a grid approach (DELOURA, 2000). The environment is discretized into a regular grid, where each cell of the grid is classified as *free* or *occupied*. Each *free* cell is mapped to a node of the graph, and the connectivity between the nodes comes from the adjacency of the cells in the grid. Engines for Real Time Strategy (RTS) games, like the Spring RTS (SPRING, 2011) engine make use of this approach. This is a very simple approach, which can lead systems that handle large environments to have the performance affected due to the great number of nodes that this method can produce. Approaches such as quadtree and convex polygons seek to reduce the number of cells and improve the representation of the free space by adjusting the size and format of cells according to the location that they represent of the environment.

Lengyel et al. (LENGYEL et al., 1990) developed a grid-based approach for path planning. It uses the standard graphics hardware and algorithms to rasterize the configuration space in a set of bitmaps. The obstacles are rasterized in the bitmaps using the *Minkowski sum* (BERG et al., 2008). These bitmaps are then used to compute a grid, which is used to find a collision-free path on the environment. Figure 2.2 illustrates this method.

Other approach used in some games for path planning is the construction of the roadmap manually by the game-developer (NIEUWENHUISEN; KAMPHUIS; OVERMARS, 2007). This approach has the advantage that the developer can explicitly control the coverage of relevant areas of the game while keeping the number of nodes of the roadmap the smallest possible. On the other hand, this approach is very time-consuming during game development, and explicitly requires that the environment is known before the construction of the roadmap. It frequently leads to repetitive behavior, which can be very unnatural. This approach is not applicable to systems that dynamically generate the environments.

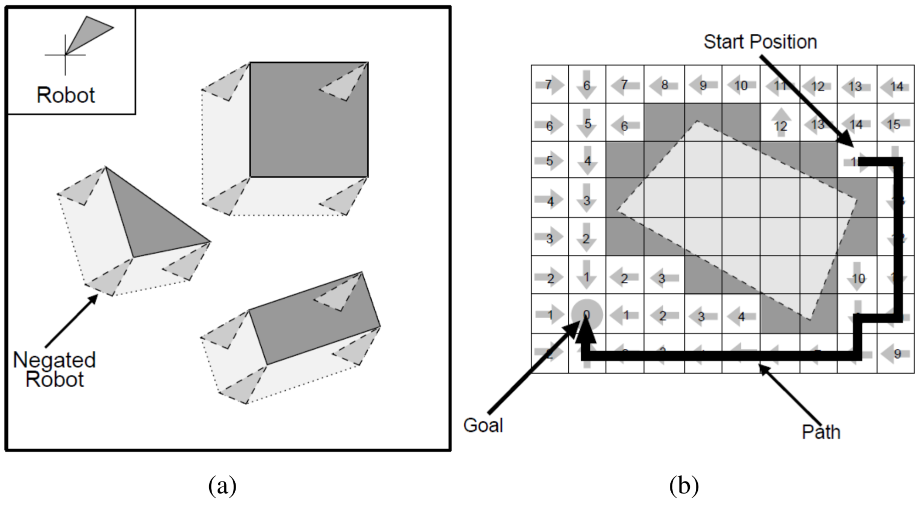


Figure 2.2: Method proposed by Lengyel et al. (LENGYEL et al., 1990) using *Minkowski sums* and regular grids. (a) The Minkowski sum applied to the obstacles, which increases the obstacle dimensions to reduce the collision probability with the autonomous robot. (b) Obstacle expanded by the Minkowski sum rasterized on a grid and a path found on the remaining free space. Figures from (LENGYEL et al., 1990).

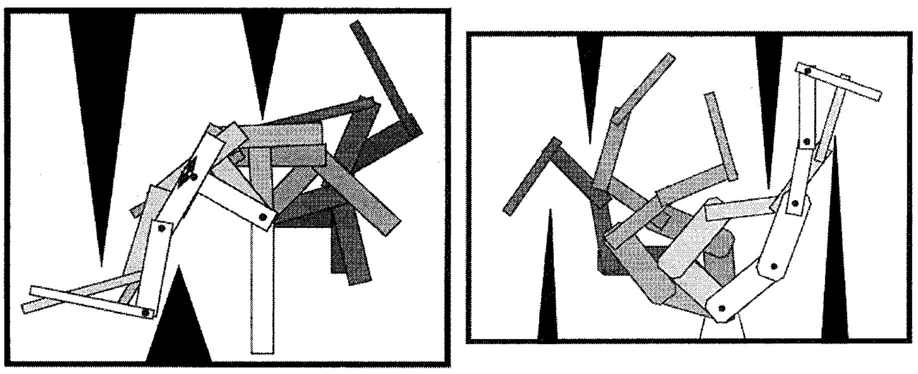
Probabilistic approaches also have been very popular due to several successful applications in high-dimensional configuration spaces (i.e. robots with many degrees of freedom – DOFs) (LINDEMANN; LAVALLE, 2005). The graphs within these methods are usually called Probabilistic-Road-Maps (PRM), and are generated during a sampling step (the *learning phase*) of the algorithm. The sampling step used by these methods is what differentiates one method from the other.

Kavraki et al. (KAVRAKI et al., 1996) introduced the PRM approach by randomly selecting positions of the possible configuration space and then testing if it was a valid configuration or not. Every time a valid configuration is found, it is added to the roadmap. Also, links to the nearest nodes are added when they represent a valid path between the related nodes. Several tests where made with articulated robots, as illustrated in the Figure 2.3.

Inspired by the PRM approach, LaValle (LAVALLE, 1998) developed the Rapidly-Exploring-Random-Tree – RRT – approach. LaValle developed a tree structure that, during the learning phase of the algorithm, *grows* from an initial point through the environment. This is done by randomly choosing valid configurations of the environment, and then expanding the tree in that direction.

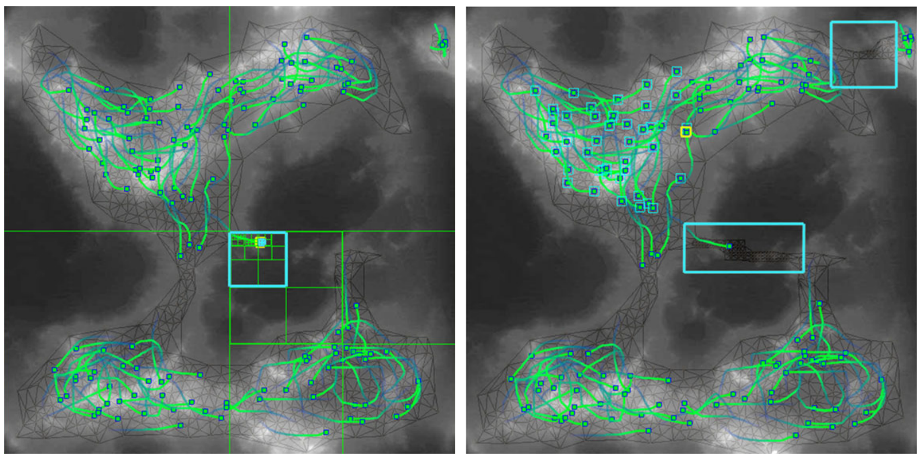
Recently, Kang et al. (KANG; KIM; KIM, 2010) presented an adaptative method that improves the roadmap based on user input. Kang et al. focused on interactive virtual environments with *non-playable-characters* (NPCs) and *playable-characters* (PCs), like in video-games. The work uses the paths generated by the PCs to identify regions of interest and improve the roadmap in these areas. This method is illustrated in the Figure 2.4.

While graph-based approaches for path planning appear to be well developed, especially for robots with many DOFs, they still lack on the quality of the generated paths for some cases. After the learning phase and construction of the roadmap, a search for a path results in a list of nodes of the graph that must be followed to reach the goal po-



(a) (b)

Figure 2.3: Path planner proposed by Kavraki et al. applied to robotic arms with many DOFs. In (a) and (b) the robots have a fixed point in the environment. Articulated arms need to move from one position to another, while avoiding collision with the obstacles. Each figure illustrates a sequence of valid poses that the robot assumes. (a) Robot with 4 joints. (b) Robot with 5 joints. Figure from (KAVRAKI et al., 1996).



(a) (b)

Figure 2.4: Adaptative approach for path planning based on user input. (a) Original roadmap generated by traditional methods. (b) Updated roadmap, with improved areas based on playable characters highlighted. Figures from (KANG; KIM; KIM, 2010).

sition. The most common approach to move the agent between two nodes is to use a linear path. This results in paths with little smoothness [1], which is acceptable for robots with many DOFs. Some applications use some kind of parametric interpolation (like Splines (PIEGL; TILLER, 1996)), but these methods do not guarantee that the resulting path will be collision-free.

Moreover, graph-based approaches are not explored for surface-restricted path planning. These methods were highly developed for use with many DOFs robots, as they efficiently solve search queries on multi-dimensional search spaces. Virtual environments that require path planning solutions (like games and interactive virtual environments) usually are easily projectable onto a plane or have complete freedom to navigate in a 3D environment. The current graph based approaches do not explore problems that may arise (such as the path quality or interpenetration of the path in the surface) when they are applied to compute paths on arbitrary surfaces.

2.2 Triangulation based path planning

Algorithms based on triangulations in general assume a discretization of the environment into a triangle mesh, executing searches in this mesh to produce paths. Although a triangle mesh could be used to generate a roadmap (and then use techniques like the ones presented in Section 2.1), the techniques presented in this section focus on extracting special features from the triangle mesh to compute the resulting path.

A path planner based on geodesic distances over triangular 3D meshes was recently proposed by Torchelsen et al. (TORCHELSEN et al., 2010). Based on a previous work (TORCHELSEN et al., 2009), Torchelsen et al. divide the input mesh (that represents the environment) into *Quasi-developable* meshes, which can be parametrized into a planar representation with minimal distortion. Based on this parametrization of the surface, Torchelsen et al. also parametrizes the goal position of navigation into the plane, and uses this point as the *origin* to compute a distance field for the surface. The idea is that the Euclidean distance from any point on the parametrization to the origin point is equal to the geodesic distance on the surface to the goal position. Torchelsen et al. uses the gradient descent of this distance field to compute the shortest path on the surface. Figure 2.5 illustrates the distance field of a triangular mesh.

The work of Torchelsen et al. focuses on multi-agent systems, and uses an occupancy grid approach to handle the collision avoidance between agents. The main advantage of this method is the high performance achieved by using an efficient CPU/GPU architecture. On the other side, the paths generated are close to the shortest ones (except by some minimal distortion, which can be controlled by the system). By using this approach, the generated paths can get very close to obstacles or generate very sharply curves, which reduce significantly the path suavity. Our potential field approach produces smooth paths that whenever is possible avoid getting very close to obstacles.

Kallmann (KALLMANN, 2005) also developed a path planner based on a triangle mesh representation of the environment. In his method, Kallmann projects the environment into a plane. The projected obstacles are used as constraints to generate a triangulation of the free areas of the environment. Kallmann used the Constrained Delaunay Trianguation – CDT (CHEW, 1987) to compute such triangulations. Based on this trian-

[1]The path is considered smooth if it doesn't have sharp corners. If the path is described by a parametrical function, it should have at least C^1 continuity. In robotics, a smooth path is important to alleviate odometric errors. In animated scenes, a smooth path usually increases the scene credibility.

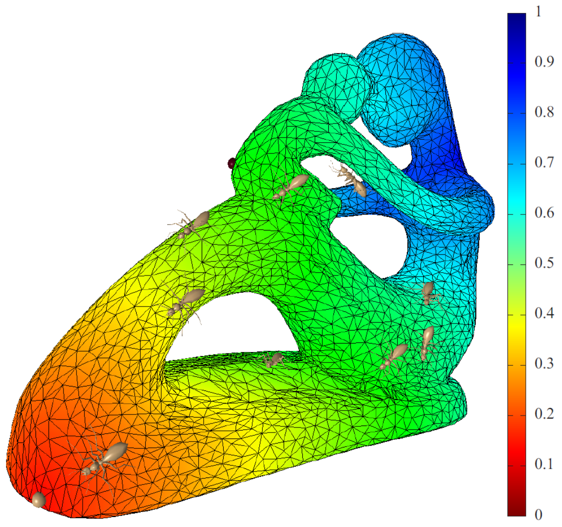


Figure 2.5: Geodesic distance field of a model. The scale represents the geodesic distance from any point of the mesh to a point in the red region. Distances are normalized to the interval $[0, 1]$. Figure from (TORCHELSEN et al., 2010).

gulation, Kallmann generates a roadmap, which can be queried for paths. Yan et al. (YAN et al., 2008) developed a similar method that improves the construction of the roadmap, thus increasing the quality of the solution for some queries. The main advantage of associating the roadmap with a triangulation is that some advanced queries can be easily solved, such as guarantee a minimum clearance on the path (KALLMANN, 2010). Figure 2.6 illustrates this kind of query.

Despite the fact that the methods developed by Kallmann or Yan et al. uses triangle meshes as the main data structure, they are developed exclusively for planar environments. Our method does not make any difference between planar and 3D surfaces.

2.3 Path planning based on potential fields

Path planning algorithms based on potential fields have in common the fact that they define a function, which is evaluated for the whole environment. The parameters of these functions are the obstacles and goal of the navigation. The result of the function is a field of values (usually refereed as *potential* values). The directions to reach the goal position can be extracted from this field.

The main difference among these techniques is the function used to compute the potential field and how directions are extracted from the field. The choice for a particular function is motivated mainly by the kind of behavior that a technique produces through the generated paths.

The work of Khatib (KHATIB, 1986) introduced the use of potential fields to implement a real-time collision avoidance system for robots with several joints. Khatib divided his work into a high level system, where the goal position of the robot is defined, and a low level system, where the potential field is used to produce commands that lead the robot to the position defined in the high level system. A set of gravitational forces was used to produce the potential field, where the goal position produces attractive forces and the obstacles produce repulsive forces.

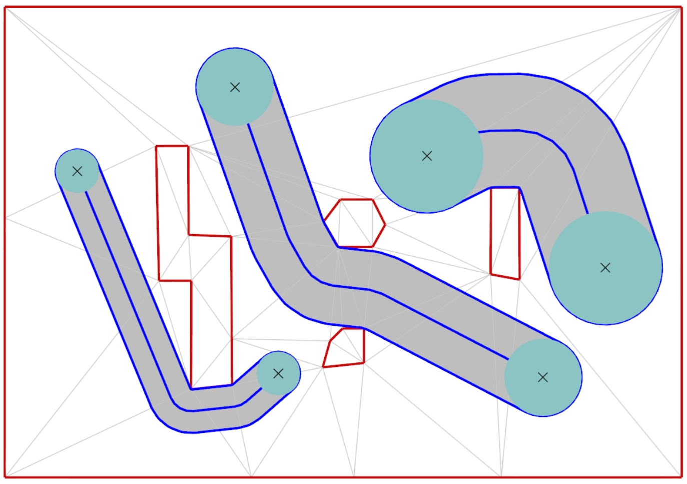


Figure 2.6: Minimum clearance path planner proposed by Kallmann. In this figure, three different paths were computed with different minimum clearance. The red lines define the environment limits, while the blue lines illustrates the resulting paths. Figure from (KALLMANN, 2010).

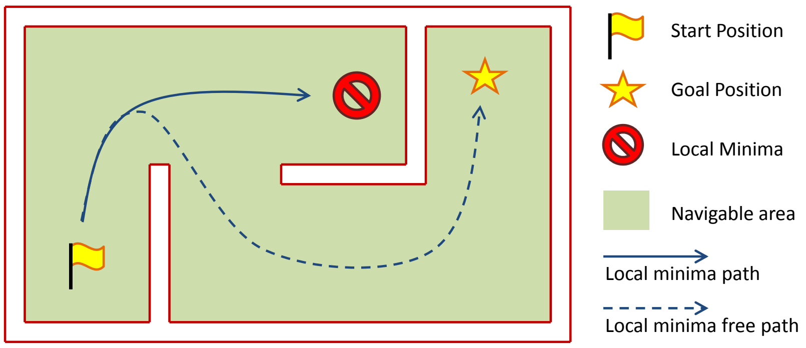


Figure 2.7: Illustration of a case where a local minima must be handled, in order to avoid that the agent get lost in the environment.

The main drawback in Khatib's approach is the occurrence of local minimum in cluttered environments, which can lead to a stable positioning of the robot before it reaches his goal. Basically, this problem happens in local approaches, where the agent does not have a global view of the environment. The algorithm makes the agent move inside the environment and, eventually, leads to a *dead end*. Figure 2.7 illustrates a case that, if not dealt correctly by the algorithm, can lead to a local minima.

To solve the local minima problem, Connolly et al. (CONNOLLY; BURNS; WEISS, 1990) developed a path planner based on harmonic functions. Using the Laplace's Equation to generate the potential field, Connolly demonstrated that his path planner generates smooth and free of collisions paths. As Connolly presented, Laplace's Equation does not produces local minima. In Figure 2.8 Connolly et al. illustrated some of their results in 2D and 3D environments.

Several path planners based on harmonic functions have been proposed after the one proposed by Connolly. Iñigues and Rosell (IñIGUEZ; ROSELLY, 2003; ROSELL; INIGUEZ, 2005) presented the Probabilistic Harmonic-function-based Method - PHM,

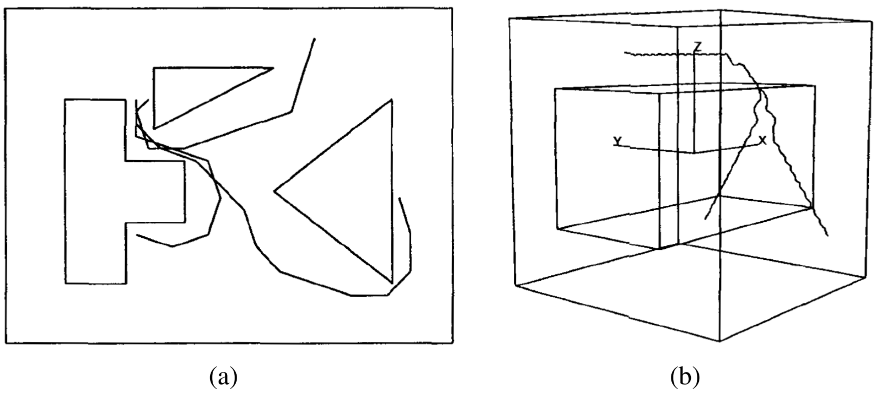


Figure 2.8: Examples of paths generated with Connolly's path planner. (a) a 2D environment with 3 obstacles, 3 start positions and 1 goal. (b) view of a 3D environment with 2 starting points and one goal. Images from Connolly et al. (CONNOLLY; BURNS; WEISS, 1990).

where the path planner is based on a combination of harmonic functions and a random sampling scheme. This was done to overcome the fact that path planning algorithms based on potential fields require full knowledge of the environment before the potential field can be computed, while keeping the advantages of the algorithms based on potential fields.

Trevisan et al. (TREVISAN et al., 2006) presented a framework for path planning and exploration based on harmonic functions. Based on a previous work (PRESTES et al., 2001), Trevisan exploited the use of harmonic functions in order to generate paths inside an environment, favoring an exploratory behavior of the agent. This behavior was achieved through the solution of the Laplace's Equation, using a set of rules to control the boundary conditions of the system.

Although Connolly's work presents an example of application in a full 3D environment, the most part of his work assumes that the environment can be projected on a 2D plane. Other works improved Connolly's work to deal with robots with several joints or in full 3D environments (ZHAO; FAROOQ; BAYOUMI, 1996). Recent work based on harmonic functions like the ones of Iñigues and Rosell (IñIGUEZ; ROSELLY, 2003; ROSELL; INIGUEZ, 2005) and Trevisan et al. (TREVISAN et al., 2006) also assume that the environment can be discretized in that way. None of these works raised the problem of path planning constrained to arbitrary surfaces.

In the same way as the works based on harmonic functions, Hussein and Elnagar (HUSSEIN; ELNAGAR, 2002) developed a path planner based on Maxwell's Equations. These equations, which are commonly used for magnetic field problems, also do not present local minima. Although Hussein and Elnagar raised the problem of path planning in 3D environments, this work also does not presents a solution focusing only on surfaces.

To deal with several agents at the same time (which is usually called in the literature as *crowd simulation*), Treuille et al. (TREUILLE; COOPER; POPOVIć, 2006) developed a path planner based on dynamic potential fields, which integrates global navigation and local collision avoidance between the agents in the same set of equations. The Treuille et al. work is based on a particle simulation system, where the velocity of each particle and the distance between them is controlled within his equations. These equations are computed within a regular grid, representing the environment. One of the advantages of the method is the performance, which is able to simulate up to 10,000 agents in a standard

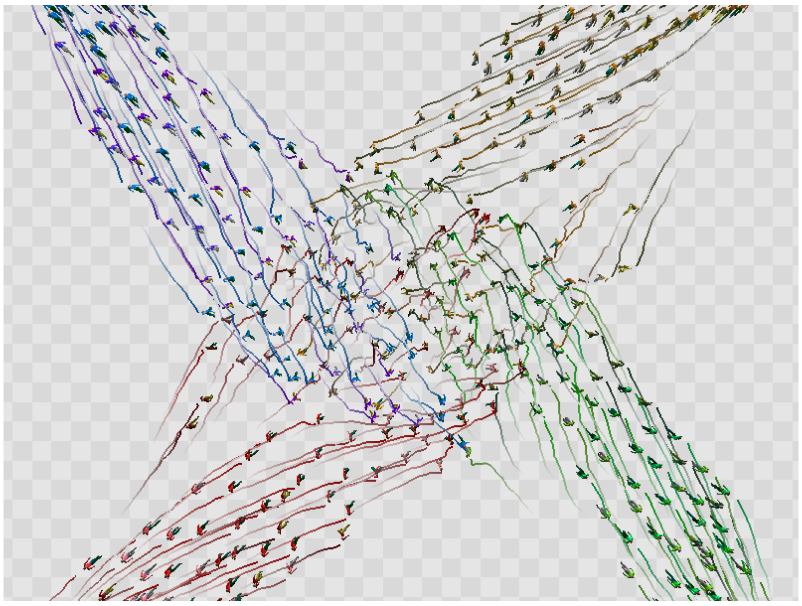


Figure 2.9: Groups crossing using Treuille's model for crowd simulation. Note the traces left by the agents while walking. Figure from Treuille et al. (TREUILLE; COOPER; POPOVIć, 2006).

personal computer at the time of his work. Figure 2.9 presents a case where several groups of agents cross each other is illustrated.

Also within the crowd simulation field, Park (PARK, 2010) developed a path planner based on particle simulation. His work uses a potential field approach to control the particles associated with agents. These potential fields are generated by harmonic functions. The main advantage of his method is the possibility to control the behavior of agents. By using some *control particles*, the potential field is affected in a way that it can control how the whole crowd moves.

There are also approaches that focus on crowd simulation for 3D environments. The work of Reynolds (REYNOLDS, 1987) is focused on the simulation of flock of birds and school of fish, also by simulating a particle system. In his work, the agents can move freely within the environment while keeping the flocking behavior and avoiding obstacles. Recently, Hartman and Benes (HARTMAN; BENES, 2006) developed a technique introducing the concept of leadership in these kind of simulation.

As in the other path planning algorithms, systems that simulate crowds of agents only developed solutions for 2D or 3D navigation. Methods like the ones presented by Park or Treuille et al. assume that the environment can be projected on a 2D plane. Other methods, such as the one of Hartman and Benes make efforts to improve the flocking behavior, and so do not present any solution for path planning constrained to 3D arbitrary surfaces.

3 BVP PATH PLANNER

In this chapter, we will present the BVP Path Planner. It is a technique used to generate smooth paths to control both real robots and virtual agents. It was proposed by Trevisan et al. (TREVISAN et al., 2006), and several other extensions were developed after that. Among these extensions, Silveira et al. (SILVEIRA et al., 2009) developed it for simulating human behaviors within virtual environments. The 3DS-BVP proposed in Chapter 4 was based on the work developed by Dapper, which is presented in this chapter.

The BVP Path Planner is a 2D Path Planner based on boundary value problems to solve a variation of the Laplace's equation. It is able to find paths by mapping the environment to a 2D plane and solving the equation on that surface. The Laplace's equation guarantees that the algorithm is free of local minima. The gradient descent of the generated potential field is used to guide the agent through the environment.

Intuitively, it will assign values to some points of the environment (normally, the obstacles and the goal position of the navigation). Then, a function will use the previous assigned values to compute new values to other points (normally, the free area of the environment). At the end of this process, the values on these points can be used to guide an agent from one point to another of the environment, by always look for points with lower values than the one on its current position. This is repeated until the target position is reached.

In the reasoning above, the potential value is the one assigned or evaluated on each point of the environment. So, the potential field is the set of points on the environment and its associated potential values. The boundary conditions are the set of values defined to some points of the environment, like obstacles, limits of the environment and goals. Finally, the gradient descent is the direction that an agent needs to take on a given position in order to go to a position on the environment where the potential value is lower than the one on that position. Following the gradient descent, the agent will reach its goal, if a path exists.

3.1 Potential fields for path planning

The BVP path planner is a 2D path planner that generates paths using the potential information computed from the numeric solution of the partial differential equation

$$\nabla^2 p(\mathbf{r}) \; = \; \epsilon \mathbf{v} \cdot \nabla p(\mathbf{r}) \; , \tag{3.1}$$

with Dirichlet boundary conditions. In this equation, $\mathbf{v} \in \Re^2$ and $|\mathbf{v}| = \mathbf{1}$ corresponds to a vector that inserts a perturbation in the potential field; $\epsilon \in \Re$ corresponds to the intensity of the perturbation produced by $\mathbf{v}$; and $p(\mathbf{r})$ is the potential at position $\mathbf{r} \in \Re^2$.

The gradient descent on these potentials represents navigational routes from any point of the environment to the goal position. Trevisan et al. (TREVISAN et al., 2006) show that this equation does not produce local minima and generates smooth paths.

To numerically solve a BVP, we can consider that the solution space is discretized in a regular grid. The grid has $m \times n$ cells, each one identified by its coordinates (i, j). Each cell (i, j) is associated to a squared region of the environment and stores a potential value $p(i, j)$. Each cell is distant from its neighbors by 1 unit. Each region is classified as:

- *Occupied*, if the region contains any obstacle that may block the agent navigation;
- *Goal*, if the region contains the position that the agent wants to reach at the end of the navigation. We assume that there is only one *goal* region on the environment, once an agent can go to only one position at a time;
- *Free*, if the agent can freely navigate over it.

After each region is classified, the associated cells of the grid receive an initial potential value. We must define the *high potential* and the *low potential* values, which are usually defined as 1 and 0, respectively. Then, each cell receives a potential value as follows:

- Cells associated with *occupied* regions receive the *high potential.* Also, they are tagged as *occupied*;
- The cell associated with the *goal* region receives the *low potential.* Also, the cell is tagged as *goal*;

The Dirichlet boundary conditions will be defined by the cells with the *occupied* and *goal* cells. The cells with the *occupied* tag (and, thus, the higher potential value) will prevent the agent from running into obstacles, while the *goal* cells (with the lower potential values) will generate an attraction basin that pulls the agent. This attraction to the goal and repulsion from the obstacles comes from the interpretation of the gradient descent of the potential field. Applying the *occupied* tag and the *high potential* value to the cells at the border of the grid will also make these cells repulse the resulting paths. This will avoid that the agents get out of the area covered by the grid.

After the boundary conditions being defined, the Gauss-Seidel method is employed to compute the potentials of the *free* cells. In this method, the potential of the *free* cells are improved at each iteration, getting closer to the correct solution of the Equation 3.1. From Equation 3.1, the Gauss-Seidel method updates the potential of a cell c through

$$p_c = \frac{p_b + p_t + p_r + p_l}{4} + \frac{\epsilon((p_r - p_l)v_x + (p_b - p_t)v_y)}{8} \tag{3.2}$$

where $p_c = p(i,j)^t$, $p_b = p(i, j-1)^t$, $p_t = p(i, j+1)^{t-1}$, $p_l = p(i-1, j)^t$, $p_r = p(i+1, j)^{t-1}$ and $v = (v_x, v_y)$. t is a reference to a value in the current iteration, while $t - 1$ refers to a value in the previous iteration. Figure 3.1 illustrates these cells in a grid.

The GS method allows the use of partial results as an approximation of the potential field (PRESTES et al., 2001). Since the exact solution is not necessary, we can stop the iterative process when the accumulated error is below a tolerance threshold e_{max}. Depending on whether the accumulated error is below or not e_{max}, it is said that the

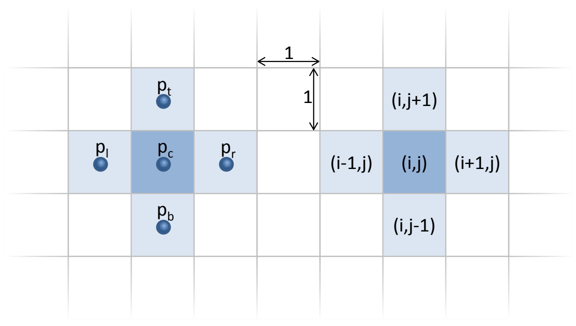


Figure 3.1: Cells of a grid, when Equation 3.2 is evaluated for the cell p_c.

potential field has converged or not, respectively. The accumulated error e in a given iteration t is computed by

$$e(t) = \sum_{i=1}^{m} \sum_{j=1}^{n} |p(i,j)^t - p(i,j)^{t-1}|, \tag{3.3}$$

where $p(i,j)^t$ is the potential of the cell (i,j) at the iteration t, $p(i,j)^{t-1}$ is the potential of the same cell in the previous iteration and m and n are the grid dimensions. When $e(t) \leq e_{max}$, the GS method stops its execution and the agent can start to walk on the environment.

After the computation of the potential field, the agent moves following the direction of the gradient descent at its current position r_a. This is computed through the cell (i,j) associated with the position r_a, with the Equation

$$-\nabla p_c = \left(\frac{p_l - p_r}{2}, \frac{p_b - p_t}{2} \right), \tag{3.4}$$

where p_c, p_l, p_r, p_b and p_t uses the same notation used in Equation 3.2.

3.2 Global path planner

The global path planner is composed of several *global maps*. Each possible goal position required by at least one agent requires one *global map* associated in the global path planner. A *global map* is composed by a goal position and a potential field. The grid of cells used by the potential field is obtained from the discretization of the entire environment. The potential field is built and computed as described in Section 3.1.

For any position in the environment, if is possible to define a path that leads to a given goal position, the *global map* is able to find that path. This can be guaranteed by the properties of Equation 3.1, if the discretization represents the connectivity of the environment and if the potential field has converged after the GS method. This way, any set of agents that needs to reach the same goal position can share the same *global map*, even if they are in different locations of the environment. This is useful to avoid the computation of several *global maps* for the same goal.

3.3 Local path planner

Every agent in the environment is associated with one *local map*, which is used for local navigation. As the *global map*, the *local map* is composed of one potential field and

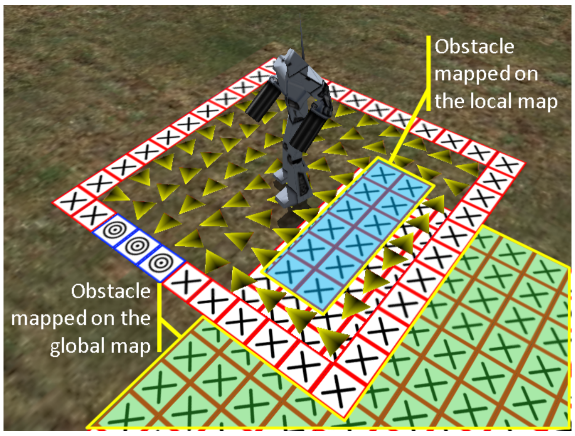

Obstacle mapped on the local map
Obstacle mapped on the global map

Figure 3.2: *Global map* and *local map*. Part of the *local map* of the agent is over an obstacle (not shown in the figure). From the *global map*, only the obstacle cells are shown, to avoid visual pollution.

one goal position. Differently from the *global map*, it cannot be shared between different agents.

The *local map* is associated with a small area of the entire environment, centered at the current position of the agent. Every time the agent moves through the environment the *local map* moves in the same way. Thus, the *local map* is *overlaid* on the *global map*. This way, at a given time t, each cell of the *local map* is associated with a region r on the environment, which is also associated with a cell on the *global map*. So, through the region r, each cell of the *local map* is associated with a cell on the *global map*. Figure 3.2 illustrates this.

Usually, the distance between the cells in the *local map* is smaller than the distance in the *global map*. As the resolution of the grid impacts on the quality of the generated paths the *global map* uses bigger cells to assist the agents in the whole environment, while the smaller cells of the *local map* improve the quality of the motion of the agents.

The *local map* is divided into three zones: the border zone (*b-zone*), the free zone (*f-zone*) and the update zone (*u-zone*). The *b-zone* comprises the cells on the border of the *local map*, the *f-zone* comprises the cells adjacent to the *b-zone* that are not on the *b-zone*, and the *u-zone* comprises the cells adjacent to the *f-zone*, which are not on the *b-zone* or *f-zone*. Figure 3.3 illustrates these zones in the potential field grid.

While the agent moves through the environment, each cell of the *local map* is updated according to the zone that it belongs to and the associated cell on the *global map*. The cells of the *local map* are classified in the following way:

- The cells on the *b-zone* are classified either as *occupied* or *goal*. If the *goal* position is on the *f-zone* or the *u-zone*, all of these cells are classified as *occupied*. Otherwise, the direction of the gradient descent given by Equation 3.4 on the *global map* is projected on the border of the *local map* to choose the cells that will be tagged as *goal*, and the remaining are classified as *occupied*;

- The cells on the *f-zone* are classified either as *free* or *goal*. The cells in the *f-zone* associated with a *goal* cell on the *global map* are classified as *goal*. All other cell are classified as *free*, no matter if the associated cell on the *global map* is a *free* or *occupied* cell;

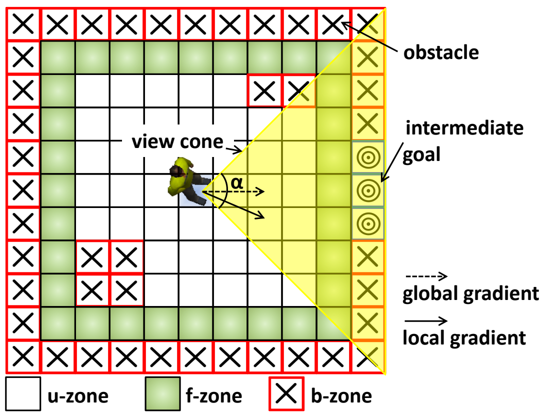


Figure 3.3: The zones of a *local map*. The *b-zone* refers to the cells on the border of the local map, and its cells are classified as *occupied* or *goal*. In the *f-zone* the cells can be classified as *free* or *goal*. Finally, the cells at the *b-zone* can be classified as *occupied*, *free* or *goal*. As the local map has only local knowledge about the environment, the gradient from the *global map* is used to set the goal on the border, thus avoiding that the agent get trapped in local minima.

- The cells on the *u-zone* are classified in the same way as the associated cells on the *global map* are. This way, the cells on the *local map* associated with *free* cells on the *global map* are classified as *free*, the cells associated with *occupied* cells on the *global map* are classified as *occupied*, and the cells associated with *goal* cells on the *global map* are classified as *goal*.

The cells of the *local map* are classified this way to avoid local minima. Because the *local map* covers only a small fraction of the environment, the agent can get stuck in a local minima if no global information is used. The usage of the gradient from the *global map* inserts the global information into the *local map*, avoiding the local minima problem.

The main advantage in the use of the *local map* approach is that it is smaller than the *global map*, reducing the computation needed to update its potential field. Because the *global map* discretizes the entire environment, it tends to have a large number of cells. Potential fields with a big number of cells are slower to update, making it a not so good option for use in dynamic environments. The *local map* approach reduces the problem to a small region around the virtual agent, with a small number of cells. Assuming that a dynamic obstacle will influence the motion of an agent only when one is near the other, the *local map* approach produces good results without spending much processing time. This way, an agent can use the gradient (Equation 3.4) of the *local map* instead of the one from the *global map* to move within the environment while avoiding dynamic obstacles.

4 PATH PLANNING IN 3D TRIANGLE MESHES

In this chapter, we will introduce our contributions by explaining the 3DS-BVP Path Planner. This is a new technique for path planning in triangle meshes with holes and bends. This technique is inspired on the BVP Path Planner, presented in Chapter 3. This way, it computes a potential field on the mesh, and uses its gradient descent to draw the paths on the surface.

The environment considered in this technique is a 3D surface, discretized in a mesh of triangles. Although any set of polygons could be used, the choice for a triangle mesh has a set of advantages. Arbitrary surfaces can be defined using just triangles, and computing the area and normal of 3D triangles are very cheap tasks in terms of processing. Also, triangle meshes are widely used in computer graphics industry to model objects and virtual environments. Many modeling software like Maya® (AUTODESK, 2011a) and 3D Studio Max® (AUTODESK, 2011b) natively handle triangle meshes. Most commercial game engines (like CryEngine® (CRYTEK, 2011), Unreal Engine® (EPICGAMES, 2011) and Unity3D® (UNITYTECHNOLOGIES, 2011)) also use triangle meshes as their main data structure for the virtual environments. All this means that our technique can be easily applied to existing 3D models and surfaces.

Concerning the potential field, the same aspects defined for the BVP Path Planner are used. The cells of the potential field are associated with the triangle vertices, and each cell is associated to a potential value and a tag. Also, there is the relaxation step, that evaluates the potential for some cells until a maximum accumulated error threshold e_{max} is achieved. At the end, the gradient is computed at the triangle level, based on the potential from the cells associated with its vertices. The main difference of the methods used in the 3DS-BVP from the methods employed in the BVP Path Planner is how the potential value and gradient is computed, as the distance between the cells in a triangle mesh is not constant in a regular grid.

This chapter is organized as follows. Initially, we present the method to compute the potential field in Section 4.1, which can be divided in terms of the way that the environment is discretized (Subsection 4.1.1) and the numerical method itself (Subsection 4.1.2). Then, we present the equations used to compute the gradient of the potential field in Section 4.2, which is used by the agent to guide itself on the surface. Next, we present some considerations about the potential fields on 3D surfaces in Section 4.3. Finally, we present the approach used to implement the BVP Path Planner in Section 4.4, presenting algorithms and data structures.

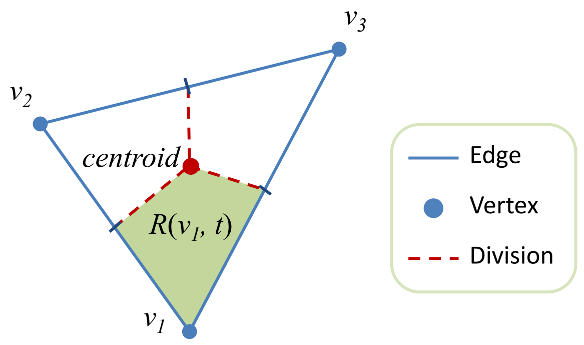


Figure 4.1: A triangle divided in regions to build a cell. In highlight, the area returned by the function $Region(v_1, t)$.

4.1 Potential fields in open triangle meshes

The path planner is *inspired* by the solution of the Equation 3.1 with the inner parameters $\epsilon = 0$ and $v = (0, 0)$. In a two dimensional environment it is easy to understand how the parameters ϵ and v are used and how these parameters affect the generated paths. But in a path restricted to a three dimensional surface it is not clear how these parameters will affect these paths. Also, the use of $\epsilon = 0$ allows the simplification of Equation 3.1 to the Laplace's equation, which is

$$\nabla^2 \, p(r) \; = \; 0 \, . \tag{4.1}$$

The word *inspired* emphasizes that our equation does not directly derives from the Equation 4.1. It is based on a heuristic function that mimics the potential fields generated by the Laplace's Equation.

In order to present the heuristic function used to approximate the potential fields generated by Equation 4.1, we first present the way that a triangle mesh representation of an environment is discretized into cells. Then, we present how the potential of each cell influences each other, resulting in a numerical method that computes the potential field.

4.1.1 Environment discretization

We assume that the environment is represented by a triangle mesh. The set of vertices of the triangle mesh induces the set of cells of the potential field. Each cell is a small fraction of the surface, composed by a fraction of each triangle sharing the same vertex. As each triangle has three vertices, each triangle is divided in three regions, being each region part of a different cell. The triangle is divided in regions by connecting each median of the triangle edges to the triangle centroid. Each triangle region is associated to the vertex in its limits and, for a given vertex v of the triangle t, the function $Region(v, t)$ returns that associated region. This is illustrated in the Figure 4.1.

In order to define a cell, we need to know the regions that are part of it. The set of triangles that share the same vertex v is returned by the function $Triangles(v)$. Then, we say that the cell associated to a given vertex v is defined by $Cell(v) = \{Region(v, t) | t \in Triangles(v)\}$. The function $Vertex(c)$ returns the vertex associated to a given cell c. Figure 4.2 illustrates these concepts by showing an example, where the cell associated to the vertex v_1 is highlighted, as well as the triangle regions around that vertex.

Finally, the concept of *neighbor cell* is derived from the connections between the vertices in the triangle mesh. Two cells are *neighbors* if the associated vertices are shared by

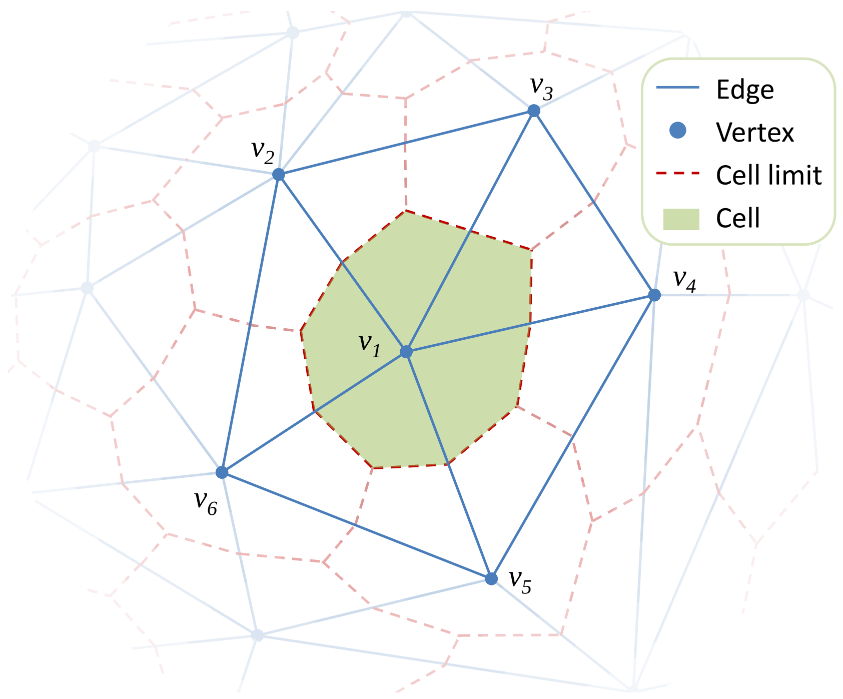


Figure 4.2: A triangle mesh and its corresponding cell division. The five highlighted triangles ($\{v_1, v_2, v_3\}$, $\{v_1, v_3, v_4\}$, $\{v_1, v_4, v_5\}$, $\{v_1, v_5, v_6\}$ and $\{v_1, v_2, v_6\}$) share the same vertex v_1 and, therefore, they are in the set $Triangles(v_1)$. Applying the function $Region(v_1, t)$ for each triangle t in $Triangles(v_1)$ results in the cell associated to the vertex v_1, which is highlighted with the dotted red line around v_1.

the same triangle. This way, two cells c_1 and c_2 are neighbors if $\{Triangles(Vertex(c_1) \cap Triangles(Vertex(c_2)\} \neq \{\emptyset\}$. This way, the set of neighbors of a cell c is defined by $Neighbors(c) = \{Cell(v) | \forall t \in Triangles(c), \forall v \in Vertices(t), v \neq Vertex(c)\}$.

Note that, although Figure 4.2 looks like a Voronoi diagram (BERG et al., 2008), it is a different concept. In this work, Voronoi diagrams are not used to avoid certain cases where a Voronoi cell may cover a triangle that does not share the vertex associated to the cell. For example, consider the case depicted in Figure 4.3. The Section 4.2 explains how a moving agent uses the gradient of the cells associated to the vertices of the triangle where the agent is on. Consider a 3 dimensional environment with an agent moving on the triangle $t = \{v_2, v_3, v_4\}$. If the vertex v_1 is not on the plane defined by t, the gradient between the cells will not be parallel to that plane. As it would be desirable that the gradient follows the surface, we choose to not use Voronoi diagrams.

4.1.2 Computing the potential field

In order to compute the potential field on the set of cells, the boundary conditions must be set according to the environment discretization. These boundary conditions are set according to the positions of obstacles and the goal in the environment. We assume that obstacles and goal positions are constrained to the surface. As in the BVP path planner, each cell receives a tag and an initial potential value, as follows:

- Cells associated with occupied areas of the surface are tagged as *occupied* and receive the *high potential* value. Cells associated with limits on the triangle mesh are tagged as well;

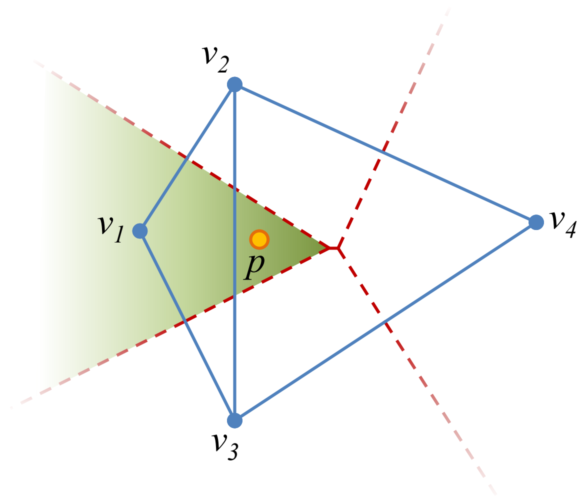


Figure 4.3: The Voronoi diagram of the triangle vertices. Note that the Voronoi cell associated to the vertex v_1 overlaps the triangle $\{v_2, v_3, v_4\}$. In this case, an agent over the point p can be associated to the vertex v_1.

- The cell associated with the goal position receives the *goal* tag and the *low potential* value;
- Cells associated with *free* navigable areas of the environment are tagged as *free* and receive a mean value between the *low* and *high potential* values, as in the BVP path planner.

In the classification above, we assume that a limit on the triangle mesh is an edge that belongs to only one triangle. In this text, these edges will be called as *limiting edges*. This way, if a triangle mesh has limits, it will be called *open mesh*. If it has no *limiting edge* (like a sphere is, for example), it will be called *closed mesh*. The cells associated with limits are those associated with any vertex in one of the ends of a *limiting edge*.

In order to update the potential value of the *free* cells, a set of functions need to be defined. Assuming that the function $Dist(v_1, v_2)$ returns the Geodesic distance between the vertices v_1 and v_2 (which is equal to the Euclidean distance if the vertices are adjacent in the triangle mesh), equations

$$D_{min}(c_i) = min(\{Dist(Vertex(c_i), Vertex(c_j)), \forall c_j \in Neighbors(c_i)\}) \tag{4.2}$$

and

$$D_{max}(c_i) = max(\{Dist(Vertex(c_i), Vertex(c_j)), \forall c_j \in Neighbors(c_i)\}) \tag{4.3}$$

return the minimum and maximum values, respectively, in a set constituted by the distances between the vertex of a cell c_i and the vertices in its neighborhood.

Based on Equations 4.2 and 4.3, the influence of a cell c_j over the cell c_i, relative to the neighborhood of c_i is

$$I(c_i, c_j) \quad = \quad D_{min}(c_i) + D_{max}(c_i) - Dist\left(Vertex(c_i), Vertex(c_j)\right) \tag{4.4}$$

Equation 4.4 is a heuristical measure of how much influence the potential of one cell c_j has over its neighbor c_i. As Equations 4.2 and 4.3 are computed from the same input

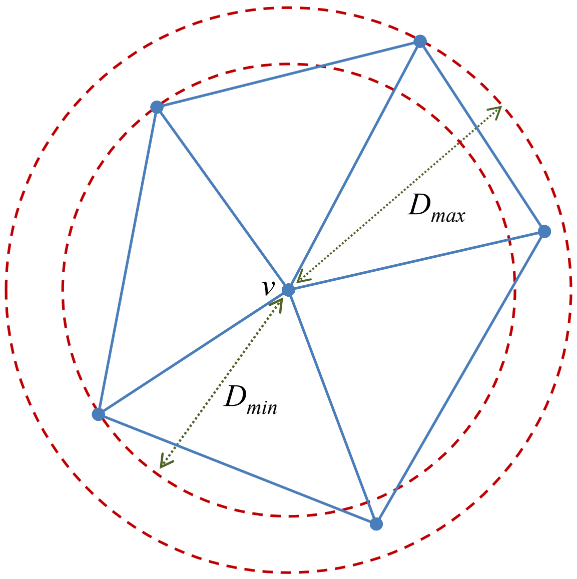


Figure 4.4: The values D_{min} and D_{max} of the cell $Cell(v)$. D_{min} can be though as the maximum radius of a circle centered in v that keeps the neighbor vertices of v outside of the circle. D_{max} can be though as the minimum radius of a circle centered in v that keeps the neighbor vertices of v inside of it.

set and they return, respectively, the minimum and maximum values of it, it is guaranteed that $D_{min}(c_i) \leq D_{max}(c_i)$. The closer the cell c_j is to c_i, the closer the value of $I(c_i, c_j)$ is to D_{max}. The farther a cell c_j is from the cell c_i, the closer the value of $I(c_i, c_j)$ is to D_{min}. Then, the result of $I(c_i, c_j)$ can be interpreted as how close a cell c_j is to c_i, in a scale between $D_{min}(c_i)$ and $D_{max}(c_i)$. Figure 4.4 illustrates the values D_{min} and D_{max} of $Cell(v)$.

Based on Equation 4.4, the function

$$I_{total}(c_i) = \sum_{j=1}^{\#Neighbors(c_i)} I(c_i, c_j) \tag{4.5}$$

computes the sum of the influences that a cell receives from its neighbors. This function is then used in the equation

$$p(c_i) = \sum_{j=1}^{\#Neighbors(c_i)} p(c_j)\frac{I(c_i, c_j)}{I_{total}(c_i)}, \tag{4.6}$$

which is used to update the potential $p(c_i)$ of a *free* cell c_i. It is a heuristic method to estimate the value of *free* cells in the potential mesh.

The potential values of the *free* cells are updated using Equation 4.6 until the convergence sets in, as in the BVP path planner. A threshold error e_{max} is used to verify if the potential field has converged. Based on Equation 3.3, the accumulated error at a given iteration t on the set of cells C is

$$e(t) = \sum_{i=1}^{\#C} |p^t(c_i) - p^{t-1}(c_i)| \tag{4.7}$$

As said in the Section 3.1, the exact solution of the potential field is not required as an approximate solution is able to provide directions to the defined goal position. Although this is true, a path can only be found if at least some minimal error is achieved, which is represented by the threshold we compare to the result of Equation 4.7. Ideally the error threshold should be equal to 0, but this could lead to a high number of iterations until the convergence sets. In our experiments, we choose the threshold by experimentation.

4.2 Computing the gradient of the potential field

After the computation of the potential field, an agent should be able to follow its gradient descent in order to reach the goal position. Equation 4.6 is able to mimic the results produced by the Laplace's Equation. We can then calculate the gradient of the potential field produced with Equation 4.6, as we use to do with the Laplace's Equation.

One possible solution from the calculus to compute the gradient of the Laplace's Equation in an unstructured triangular discretization involves the use of the integral form of that equation. This can be obtained by integrating the equation over some volume Ω, and then applying the Gauss Divergence Theorem. The integral form of the Laplace's Equation and its equivalent after applying the Gauss Divergence Theorem is

$$\int\limits_{\Omega} \nabla^2 p(r) d\Omega = \oint\limits_{\partial\Omega} \nabla p(r) \cdot n dA = 0. \tag{4.8}$$

The relation between the gradient of the function $p(r)$ over a volume and the integral over the surface of the volume of the function times the normal area vector is

$$\int\limits_{\Omega} \nabla p(r) d\Omega = \oint\limits_{\partial\Omega} p(r) n dA. \tag{4.9}$$

The relation above, when computed in sufficiently small volumes, can be used to compute an approximation of the gradient. This result in

$$\nabla p(r) \approx \frac{1}{\Omega} \oint\limits_{\partial\Omega} p(r) n dA. \tag{4.10}$$

Assume that we want to compute the approximation of the gradient in the triangle $t = \{v_0, v_1, v_2\}$, illustrated in Figure 4.5. For this, we need to compute the edge normals n_0, n_1 and n_2, which are vectors associated to the edges of t. These vectors are in the same plane defined by the triangle t, in the orthogonal direction of the associated edge, and have the same length of the edge. We also need the area of the triangle t, given by $Area(t)$. Finally, we assume that the function $p(r)$ is already computed in each vertex, and the value of $p(r)$ in the vertex v can be retrieved by $p(v)$.

The approximation of the Equation 4.10 over the triangle t yelds the equation

$$\begin{aligned}\nabla p(t) \approx \frac{1}{Area(t)} \Bigg[&\left(\frac{1}{2}(p(v_1) + p(v_2))n_1\right) + \left(\frac{1}{2}(p(v_2) + p(v_0))n_2\right) \\ &- \left(\frac{1}{2}(p(v_0) + p(v_1))n_0\right)\Bigg].\end{aligned} \tag{4.11}$$

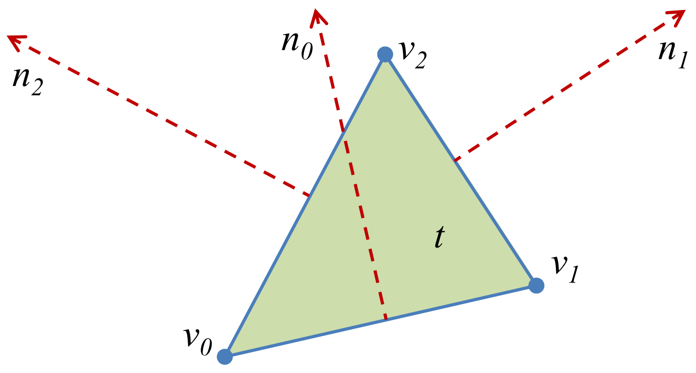


Figure 4.5: A triangle t, its vertices and *edge normals*. These normals are required to compute the gradient of the triangle $t = \{v_0, v_1, v_2\}$. Note that $Dist(v_0, v_1) = ||\hat{n}_0||$, $Dist(v_1, v_2) = ||\hat{n}_1||$ and $Dist(v_2, v_0) = ||\hat{n}_2||$.

The Equation 4.11 can be simplified to

$$\nabla p(t) \approx \frac{1}{2Area(t)} \left[p(v_0)(n_2 - n_0) + p(v_1)(n_1 - n_0) + p(v_2)(n_1 + n_2)\right]. \tag{4.12}$$

As the vectors n_0, n_1 and n_2 are in the same plane and have the same lengths of the edges of the triangle (being orthogonal to each edge), the Appendix A - Proof of Equation 4.13 allows the following equality:

$$n_1 + n_2 - n_0 = 0. \tag{4.13}$$

This leads a simplification in the Equation 4.12 to

$$\nabla p(t) \approx \frac{1}{2Area(t)} \left[-p(v_0)n_1 - p(v_1)n_2 + p(v_2)n_0\right], \tag{4.14}$$

which is then used to approximate the gradient of the potential field at the centroid of the triangle t, which is then followed by the agent to reach the goal position.

4.3 Potential fields in 3D surfaces

In a completely planar environment discretization, the path planner presented until Section 4.2 behaves in a similar way to the BVP Path Planner (Chapter 3). The goal position must be checked with a *goal* tag in the respective cell, while the obstacles are tagged as *occupied* in their cells. Any limits in the environment are also tagged as *occupied* and all the remaining cells are tagged as *free*. All cells receive some potential value, according to what was specified in Sub-Section 4.1.2. Then, the relaxation step evaluates adequate values for *free* cells. And finally, the agent can use the gradient of the potential field to draw a path to follow.

In such kind of environment, the path planner can take advantage of a simpler math, since all the vertex positions and normals can be manipulated in 2D. Naturally, a 3D discretization will require a third coordinate for the position of vertices and normals.

For 3D open meshes (as defined in Sub-Section 4.1.2) the path planner works with minimal modification, and the addition of a coordinate axis does not introduce significant changes in any of the methods presented in the previous section. During the relaxation,

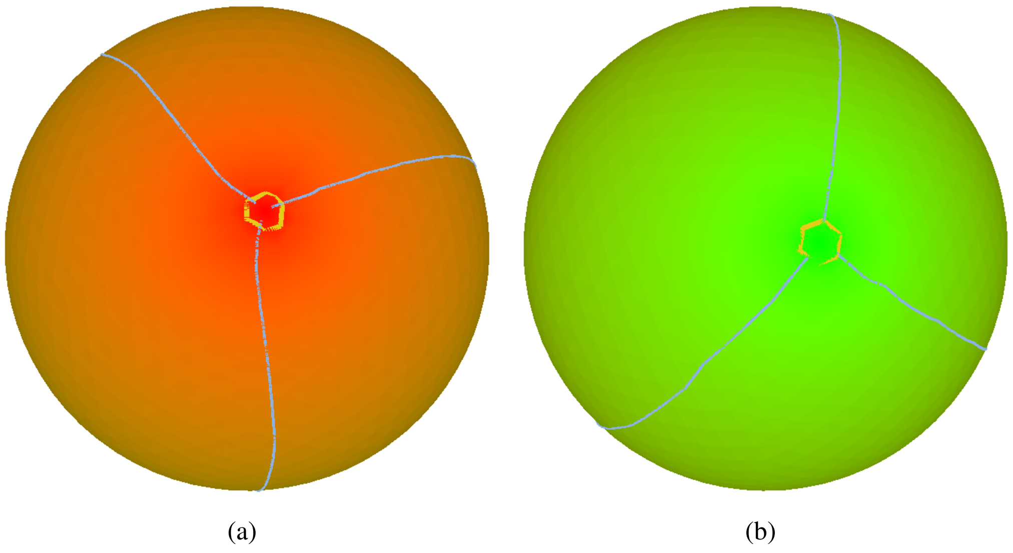


Figure 4.6: Views of the north and south pole of a sphere, and three paths on it. (a) North pole, with an artificial obstacle. (b) South pole, with the goal position. According to the initial position inside the artificial obstacle in (a) the agent takes a very different path to reach the goal position in (b).

only the function $Dist(v_1, v_2)$ changes and must compute the Geodesic distance in 3D (note that for two adjacent vertices in the mesh, this is equal to the Euclidean distance). In addition, the normals used to compute the gradient also must be stored in 3D, being parallel to the plane defined by the triangle which they belong.

The most relevant modification in the algorithm to deal with 3D meshes is the handling of boundary conditions. Our algorithm requires that at least two different boundary conditions must be set: one *goal* cell and one *occupied* cell. We assume that the *goal* is always defined, so for some 3D meshes we should take a special care with the *occupied* cells.

If the 3D mesh is open, the cells associated with the limits of the mesh will force the existence of *occupied* cells. Closed meshes with obstacles on its surface will also force the existence of *occupied* cells. This will generate a gradient that prevents the agent from leaving the surface limits or colliding with obstacles, and guides to the goal position.

However, if the mesh is closed and have no obstacle on its surface, then our algorithm will generate only *goal* cells. If the boundary conditions comprises only one distinct potential value, the solution of the potential field will comprises cells with this same value. This raises the problem of null gradient in the potential field (the result of Equation 4.14 will be $(0, 0)$). As the agent needs a valid gradient as a guide to the goal position, the null gradient gives no clue about which direction it should follow, and the agent will be stuck in the current position.

To avoid the occurrence of null gradients, the cell containing the initial position of the agent is tagged as *occupied* and receives the *high potential* value. Although the cell does not contain any obstacle, this will force the potential field to have a gradient capable of guiding the agent from its initial position to the goal. Also, as the agent is leaving that position away, the obstacle added will not modify the existence of a path from the initial to the goal position.

Meanwhile, this approach has a drawback. The path taken by the agent may change significantly according to the initial position of the agent inside the modified cell. Because the cell that corresponds to the initial position is modified to receive the *occupied* tag and *high potential* value, the gradient on the cell borders will be significantly different according to the triangle where the position is. The fact that the mesh is closed and do not have other boundary conditions allows the situation where many different paths can be taken to reach the same goal position. This drawback is illustrated on the Figure 4.6.

4.4 Implementation

Based on the considerations of the previous sections, we will present our approach to implement the 3DS-BVP Path Planner. Although the data structure itself is not a contribution of this work, our algorithm description is based on it. So, initially we present the data structure used, and then the algorithms developed.

4.4.1 The data structure

The main operation needed from a data structure to store the potential field is the query for cells and their neighborhood. As in the BVP Path Planner, the environment is discretized in a set of cells. Each cell has an associated potential value and a tag. During the relaxation step of the algorithm, the potential value of a cell will be updated with a value computed from its neighbors. Also, a navigating agent will use the gradient of the potential value of the cells associated with the triangle where it is on. Finally, when the agent moves around the environment, eventually the agent must update the current cells from where the gradient is computed. As the agent does not walk with big jumps, these new cells are just neighbors of the current ones. All these operations will be detailed in the next sections.

The most appropriate solution for these queries is the use of a data structure that represents the topological division of cells in the environment. The topological representation of the environment and its cells allows us to see it as a graph, where a given cell and its neighbors are easily retrieved from the data structure. Viewing the set of cells as a graph allows us to easily associate the cell properties (the potential value, tag and position, for example) to the nodes and edges of the graph.

There are some already well defined data structures for topological divisions. The *Winged Edge* (BAUMGART, 1975), the *Quad-Edge* (GUIBAS; STOLFI, 1983) and the *Doubly-Linked Face List* (AKLEMAN; CHEN, 1999) are examples of data structures that explicitly store the connectivity of a subdivision in an efficient way. All these data structures differ in how they represent and store the objects of the division, being more suitable for one algorithm or another. For this work the *Doubly-Connected Edge List* (*DCEL*) data structure (MULLER, 1978; BERG et al., 2008) was chosen due to its simplicity and capacity to address the neighbors of a vertex, face or edge in constant time.

The DCEL data structure was initially proposed to manipulate 3D polyhedra in algorithms to compute their intersections. The DCEL is composed of three types of records: the vertex, the face and half-edge. These records are used to define the vertices, faces and edges of the surface being manipulated. The way that it is defined allows several queries about one record and how it relates to its neighbors. Also, it is possible to add additional attributes to these records without interference in the main data structure.

Figure 4.7 illustrates several of the DCEL concepts. The vertex record is used to define a point of the polyhedron that creates a corner in its geometry. An edge in the

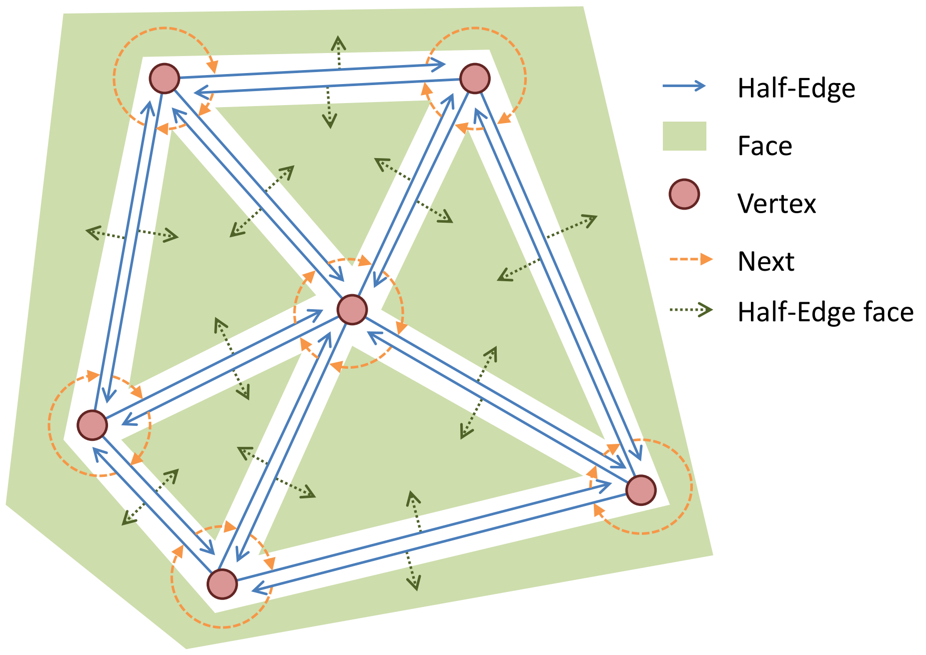


Figure 4.7: DCEL data structure. Edges are represented by pairs of parallel blue arrows, vertices by red circles, and faces by green polygons.

DCEL connects two vertices and is actually defined by a pair of half-edges, being each half-edge associated with one of these vertices. The vertex associated to one half-edge h is the *origin* of it, and is accessible through the function $Origin(h)$. About this pair of half-edges, it is said that one is the *twin* of the other. The *twin* of a half-edge is returned by the function $Twin(h)$. An edge is not explicitly stored in the data structure, but can be recovered given one of the half-edges and its twin. Finally, a face is defined by a sequence of half-edges, usually in a counter-clockwise order. Each half-edge is associated with only one face. The fact that one edge is defined by two twin half-edges allows an edge to be associated to the two faces that it divides. The half-edges are connected among themselves through the *next* and *previous* pointers, which can be retrieved by the functions $Next(h)$ and $Previous(h)$.

Several different half-edges can have its origin at a vertex v. Explicitly, the vertex v is associated with only one half-edge, returned by the function $Incident(v)$. As the set of half-edges that have their origin at v can be sorted in clockwise or counter-clockwise order, the next half-edge starting at v from the half-edge h in clockwise order is returned by $Next(Twin(h))$ and in the counter-clockwise order is $Twin(Previous(h))$.

A face f in the DCEL data structure is associated with only one half-edge, that can be retrieved by the function $Boundary(f)$. The sequence of half-edges that limits a face can be obtained by following the *next* or *previous* pointers of the half-edge returned by $Boundary(f)$. The data structure does not restrict the format of a face, but in this work only triangles are considered.

Each half-edge h can also be associated with one face, through the function $Face(h)$. As this work will consider only triangular faces, in a closed mesh each half-edge is associated with a triangle. For open meshes, some half-edges will not be associated with any specific face, and the function $Face(h)$ will return the constant *null* for these half-edges.

Finally, an auxiliary function was added to the DCEL data structure. For any two vertices v_1 and v_2, the function $Link(v_1, v_2)$ returns the half-edge which has its origin at v_1 and the origin of its *twin* half-edge at v_2. The function $Link(v_1, v_2)$ returns the constant value $null$ if the vertices v_1 and v_2 cannot be connected by a single pair of twin half-edges.

Figure 4.7 is useful as an example of the DCEL structure. In it, the half-edges, vertices and faces records are illustrated. Each pair of parallel arrows represents the twin half-edges. The *next* pointer of the half-edges are also depicted in Figure 4.7. The *previous* pointers are not, as they would be equals to the *next* pointers in the opposite direction of the arrows. Also, the associated faces of the half-edges are indicated. The origin of each half-edge is the vertex that they are connected to.

4.4.2 Algorithm

In this section we present the algorithm for the 3DS-BVP Path Planner. It assumes that the environment is already discretized in a triangle mesh stored in a DCEL. These algorithms are exactly the same for 2D environments and 3D surfaces.

The Algorithm 1 resumes the basic steps needed to execute the 3DS-BVP Path Planner. In it, the lines 2 up to 10 initializes all cells, by tagging it as *free*, except those cells associated with the limits of the mesh. From the lines 11 up to 16, it sets the tag *occupied* in all cells that have an obstacle in it. Finally, the lines 17 up to 20 sets the goal position in the respective cell, using the tag *goal*. In the lines 2 up to 20, the cells also receive a potential value, according to the Sub-Section 4.1.2.

After the initialization of the cells, the potential field can be computed (lines 21 and 22 of Algorithm 1). The relaxation is detailed in Algorithm 2, which receives as parameters the error threshold e_{max}, as well as the DCEL previously initialized. The lines 4 up to 12 keep in loop until the accumulated error in one single iteration is bellow e_{max}. When this happens, the algorithm finishes. Line 5 serves to restart the sum of the accumulated error in one iteration. Lines 6 up to 11 update the potential values of the *free* cells, as well as compute the error of one iteration. The line 8 solves the Equation 4.6. Line 9 computes the error between the current and the previous potential value of the cell c, adding it to the accumulator $e_{current}$. Finally, line 10 updates the potential value of the cell c.

When the Algorithm 2 returns to Algorithm 1 with the potential field computed in the DCEL T, the Algorithm 3 starts to move all agents over the surface of the triangle mesh. Basically, the lines 4 up to 14 stays in loop until all the agents reach the goal position, and at each iteration it picks up one agent a to move. The line 5 retrieves the current triangle t where the agent a is on the surface. Lines 6 up to 8 retrieve the cells associated with each vertex of the triangle t. Lines 9 up to 11 computes the *edge normals* of t. As said in Sub-Section 4.2, these normals are in the same plane of t. The functions $Vertex0(t)$, $Vertex1(t)$ and $Vertex2(t)$ retrieves each vertex of the triangle t. Finally, the line 12 computes the gradient descent vector (Equation 4.14), used to move the position of the agent a, in line 12.

Finally, Algorithm 4 updates the position of an agent a, according to the current gradient descent $grad$. It tests on the line 5 if the agent is already on a goal cell. If not, the lines 4 up to 13 update the current agent position and, eventually, the current face where the agent is on. Line 4 computes the new position, based on the current position of the agent, the normalized direction $grad$ and the agent speed. The test at the line 6 check if the new position is inside or outside the triangle associated to the agent. If inside, line 7 updates the current position of the agent. If not, 9 up to 12 compute a new position on the limits of the current triangle, finds the neighbor triangle that shares the crossed edge, and updates the position and current face of the agent. The auxiliary function $CrossedEdge(t, Position(a), p)$ returns the edge of t crossed by the agent when going from $Position(a)$ to p. The function $Intersection(e, Position(a), p)$ computes the intersection between the edge e and the segment starting at $Position(a)$ and ending at p.

Algorithm 1 Path planning using triangular surfaces

```
1: $T \leftarrow$ the triangle mesh representing the environment, in DCEL format
2: for all $v \in Vertices(T)$ do
3:   if $v$ is a limiting vertex then
4:     $Tag(Cell(v)) \leftarrow OCCUPIED$
5:     $p(Cell(v)) \leftarrow high$ potential value
6:   else
7:     $Tag(Cell(v)) \leftarrow FREE$
8:     $p(Cell(v)) \leftarrow$ intermediate value between $low$ and $high$ potential value
9:   end if
10: end for
11: $O \leftarrow$ the set of obstacles in the environment
12: for all $o \in O$ do
13:   $v \leftarrow Region(Position(o), T)$
14:   $Tag(Cell(v)) \leftarrow OCCUPIED$
15:   $p(Cell(v)) \leftarrow high$ potential value
16: end for
17: $g \leftarrow$ the goal position of the navigation
18: $v \leftarrow Region(g, T)$
19: $Tag(Cell(v)) \leftarrow GOAL$
20: $p(Cell(v)) \leftarrow low$ potential value
21: $e_{max} \leftarrow$ the max allowable accumulated error in the potential field
22: $Relaxation(T, e_{max})$
23: $A \leftarrow$ the set of agents that must reach the goal $g$
24: $MoveAgents(T, A, g)$
```

Algorithm 2 $Relaxation(T, e_{max})$

```
1: {$T$: the triangle mesh representing the environment, in DCEL format}
2: {$e_{max}$: the max allowable accumulated error in the potential field, $e_{max} > 0$}
3: $e_{current} \leftarrow \infty$
4: while $e_{current} > e_{max}$ do
5:    $e_{current} \leftarrow 0$
6:    for all $v \in Vertices(T)$, where $Tag(Cell(v)) = FREE$ do
7:       $c \leftarrow Cell(v)$
8:       $p \leftarrow \sum_{j=1}^{\#Neighbors(c)} \left[ p(c_j) \frac{I(c,c_j)}{I_{total}(c)} \right]$ {Equation 4.6}
9:       $e_{current} \leftarrow e_{current} + |p - p(c)|$
10:      $p(c) \leftarrow p$
11:   end for
12: end while
```

Algorithm 3 $MoveAgents(T, A, g)$

```
1: {$T$: the triangle mesh, after the Algorithm 2}
2: {$A$: the set of agents that wants to walk over the surface defined by $T$}
3: {$g$: the goal position of the navigation}
4: while $\exists a \in A$ where $Tag(Cell(Position(a))) \neq GOAL$ do
5:    $t \leftarrow Face(T, Position(a))$
6:    $c_0 \leftarrow Cell(Vertex0(t))$
7:    $c_1 \leftarrow Cell(Vertex1(t))$
8:    $c_2 \leftarrow Cell(Vertex2(t))$
9:    $n_0 \leftarrow EdgeNormal(t, Vertex0(t), Vertex1(t))$
10:   $n_1 \leftarrow EdgeNormal(t, Vertex2(t), Vertex1(t))$
11:   $n_2 \leftarrow EdgeNormal(t, Vertex0(t), Vertex2(t))$
12:   $grad \leftarrow \frac{1}{2Area(t)} [ -p(c_0) * n_1 - p(c_1) * n_2 + p(c_2) * n_0]$ {Equation 4.14}
13:   $UpdatePosition(a, grad)$
14: end while
```

Algorithm 4 $UpdatePosition(a, grad)$

```
1: {$a$: the agent that should be moved}
2: {$grad$: the gradient that will guide the agent $a$}
3: if $Tag(Cell(Position(a))) \neq GOAL$ then
4:    $p \leftarrow Position(a) + \frac{grad}{\|grad\|} Speed(a)$
5:    $t \leftarrow Face(a)$
6:    if $Inside(p, t)$ then
7:       $Position(a) \leftarrow p$
8:    else
9:       $e \leftarrow CrossedEdge(t, Position(a), p)$
10:      $p' \leftarrow Intersection(e, Position(a), p)$
11:      $Position(a) \leftarrow p'$
12:      $Face(a) \leftarrow Face(Twin(e))$
13:   end if
14: end if
```

5 RESULTS

In order to evaluate the presented algorithm, we developed a set of tests. Initially, we compared the potential fields generated with our technique with the ones generated with the BVP Path Planner. By showing the similarity between these techniques, we show that our technique is able to produce smooth paths with low collision probability. We also present the paths obtained with some non-planar surfaces. We also present some tests that show that the 3DS-BVP maintains the local minima avoidance behavior of the BVP Path Planner. Finally, we present some performance tests.

Note that this chapter presents current capabilities of our approach. Chapter 6 will focus on the limitations of our approach in extreme cases, as well as the degenerated cases not covered in this work.

5.1 Comparing our method with the BVP Path Planner

One of the motivations to develop the proposed solution is to develop a path planner capable of generate smooth paths on arbitrary surfaces. The BVP Path Planner produces paths with this property due to Equation 3.1. This equation is an extension to the Laplace's Equation (Equation 4.1), which is able to produce smooth paths without local minima. The use of parameters ϵ and v adds a control to the potential fields without affecting the smoothness of the generated paths or let local minima occur. This way, we will use $\epsilon = 0$ through the tests in order to compare our approach with the undistorted potential fields generated by the BVP Path Planner.

To demonstrate the similarities between the proposed technique and the BVP Path Planner, a set of test cases were developed comparing the paths produced by both techniques. We also made a comparison between the potential values of our technique and the ones generated by the BVP Path Planner.

We designed a set of planar environments, where the BVP Path Planner and our technique were applied. Each environment has a set of obstacles and a goal. To use the BVP Path Planner, each environment was discretized into a regular grid, as explained in Section 3.1. To compare with our technique, the environments were discretized in two different ways:

- *Regular triangle grid*: the test environments were discretized into a set of equilateral triangles;

- *Noisy triangle mesh*: the regular triangle grid generated in the previous case was deformed by adding a random displacement to each vertex of the mesh. As the previous case used equilateral triangles, limiting the displacement of vertices to less than the 50% of the triangle edge size guarantees that the triangle mesh will continue valid. In our tests, we limited the random displacement to 40% of the size of triangle edges.

As the BVP Path Planner uses regular grids to discretize the environment, the tests using regular triangle grids were conducted in order to better compare our approach with the BVP Path Planner. From these tests, an optimal result should be an exactly replication of the potential fields produced by the BVP Path Planner. As the environment discretization is different, it is expected that, numerically, the potential values of each potential field be slightly different. Then, we assume that the closest the values of potential generated by our technique are to the ones generated by the BVP Path Planner, the better are the results.

The second set of tests, using noisy triangle meshes were conducted in order to better represent the shape of the triangles found in common non-planar triangle meshes. As the average triangle shape is not as regular as in the previous case, we should expect worse results from this test. Even so, as a result we assume that the closest are the values of potential generated by our technique to the ones generated by the BVP Path Planner, the better are the results.

Figure 5.1 illustrates two of these test cases. The test case 4 is represented by Figures 5.1(a) and (b), while the test case 8 is represented by Figures 5.1(a) and (d). As said, test 4 and 8 differs by the triangle mesh that it uses, so the result produced by the BVP Path Planner in 5.1(a) is the same for test cases 4 and 8. The green region on the left border of each image represents the goal of the environment. The yellow lines delimit the *free* cells from the *occupied* and *goal* cells. Figure 5.1(a) represents the environment discretized into a regular grid, and the white lines on it illustrate paths generated by the BVP Path Planner, starting at several positions of the environment. Figure 5.1(b) presents the same environment, but discretized into a regular triangle grid. The blue lines on Figure 5.1(b) and (d) represents paths generated with our technique. Figure 5.1(c) exposes both paths from Figure 5.1(a) and (b) overlaid in a single image, in order to visually compare then. Finally, Figure 5.1(d) exposes the same environment, discretized into a noisy triangle mesh. The test cases 1, 2, 3, 5, 6, and 7 are presented in Appendix 8.6.

Observing the paths produced by the 3DS-BVP in Figure 5.1, they are quite similar to the paths produced by the BVP Path Planner. By using the noisy triangle mesh, the path loses some smoothness due to the presence of low quality triangles, but they still mimics the results produced by the BVP Path Planner and by 3DS-BVP over a regular triangular grid.

We have also compared the potential value in each cell of the potential field produced by the 3DS-BVP with the potential field generated by the BVP Path Planner in the same position of the environment. Table 5.1 presents a summary of the differences found between these values in each test case.

We can see that our method generated potential fields very similar to those produced by the BVP Path Planner. In average, for the regular triangle grid, the difference was of 0.7%, with a standard deviation of nearly 0.7%. For the noisy triangle mesh, the differences were bigger, being nearly 1% on average and 1.1% of standard deviation.

Concerning the maximum differences, the regular triangle grids presented relatively small differences. The average maximum error was about 7%, while the greatest maximum difference was found in the test case 3 (8.957%). When looking at the noisy triangle meshes, the highest difference was 49.989%, found in the test case 7. Although this is a considerable big difference between the 3DS-BVP and the BVP path planner, it occurred in an environment that had an average difference of 1.867% and standard deviation

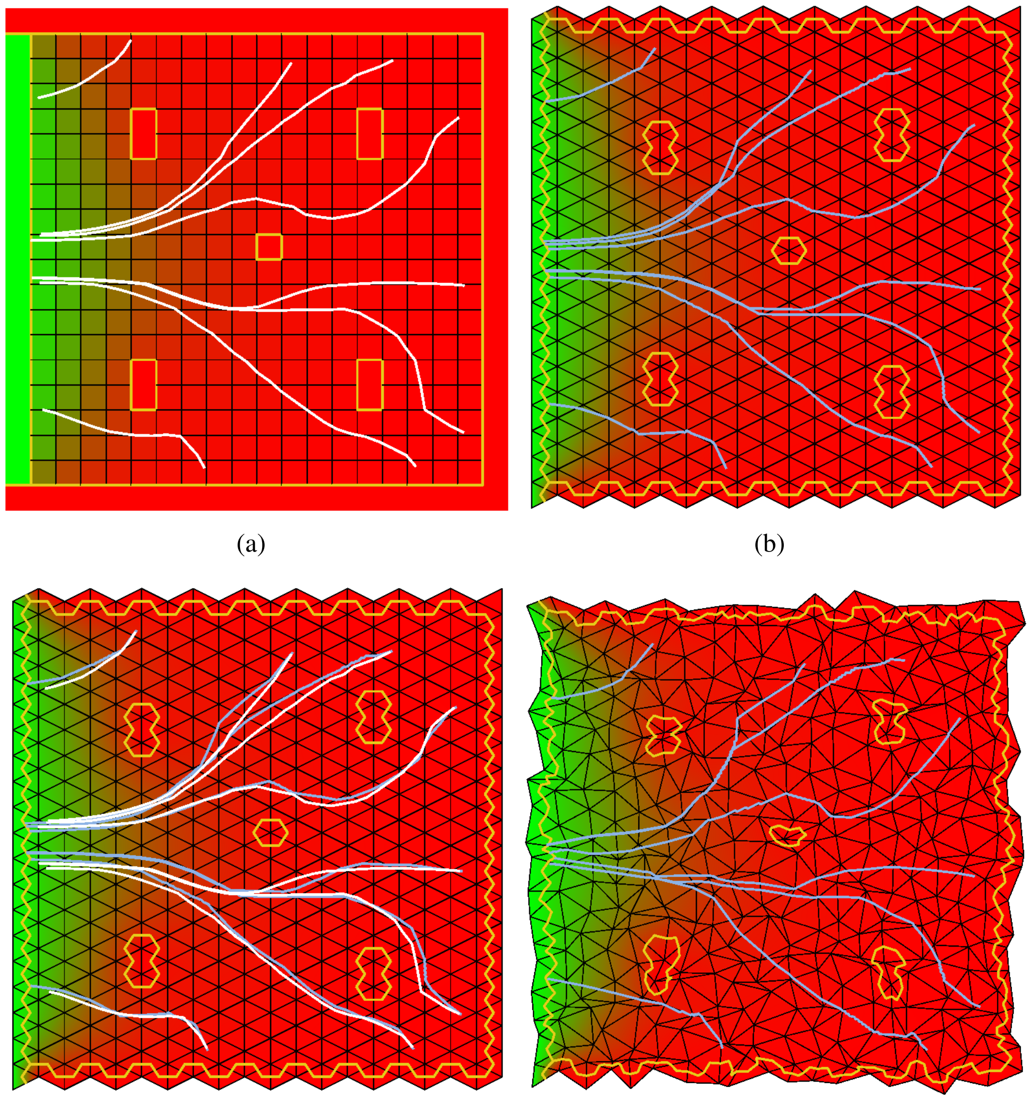


Figure 5.1: Comparison between the BVP Path Planner and the 3DS-BVP using regular and noisy triangle meshes. The gradual color transition from green to red represents the value of the potential in each point of the environment, from 0 to 1, respectively. (a) The environment discretized in a regular grid, with the paths computed with the BVP Path Planner; (b) the environment discretized in a regular triangle mesh, with paths computed with the 3DS-BVP; (c) comparison between the paths generated with the BVP Path Planner and the 3DS-BVP; (d) a discretization of the environment using a triangle mesh with noise, and respective paths generated with 3DS-BVP.

	Test Case	Average Error	Std. Deviation	Max. Error	#Cells
Regular	1	0.00394	0.00606	0.06224	2,500
	2	0.00303	0.00417	0.05802	10,000
	3	0.01844	0.01283	0.08957	10,000
	4	0.00604	0.00817	0.08911	10,000
	Average	0.00786	0.00781	0.07473	
Noisy	5	0.00903	0.01042	0.15687	2,500
	6	0.00469	0.00631	0.30755	10,000
	7	0.01867	0.01717	0.49989	10,000
	8	0.00792	0.01332	0.49913	10,000
	Average	0.01008	0.01180	0.36586	

Table 5.1: Comparison between the potential values generated with our technique and the BVP Path Planner. As the potential values from the potential fields were normalized, the average error for the regular case was about 0.7%, while in the noisy case it was about 1%. The average deviation was of 0.7% and 1.1%, respectivelly.

of 1.717%. This difference occurred due to the existence of highly deformed triangles resulting from the noise. The test case 8 produced very similar potential values, and generated the paths illustrated in Figure 5.1(d).

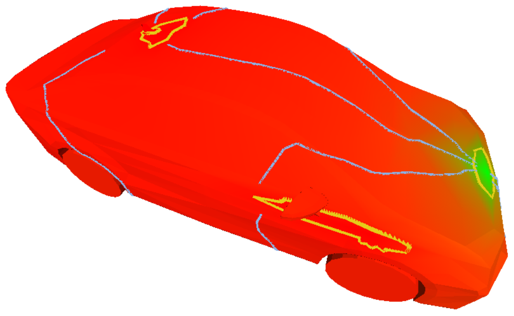
5.2 Path planning evaluation in arbitrary meshes

We also applied our algorithm on 3D models to analyze how paths are generated on these surfaces. In Figure 5.2, we used our algorithm to find paths on a car model. Despite being simple, this model helps to visualize the kind of paths generated by our algorithm. As in Figure 5.1, the green area on the surface of the model represent *goal* cells, and the yellow lines represents division between *free*, *occupied* and *goal* cells.

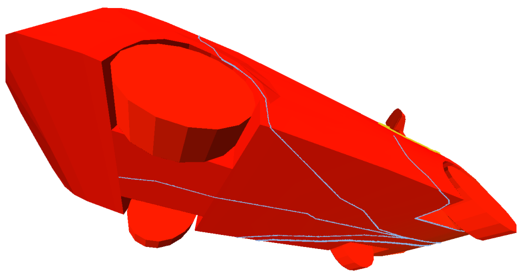
Particularly, Figure 5.2 is interesting because it clearly illustrates the drawback mentioned in Section 4.3. As one can see, the path significantly changes according to the initial position around the obstacle in Figure 5.2(d).

In another test, illustrated in Figure 5.3, we applied our algorithm on the Costa Minimal Surface (COSTA, 1984). This surface, which can be parametrically represented, is a complete minimal embedded surface with a genus with three punctures. It does not intersect itself and has no boundary (though we used a discrete version of it, with boundaries.) We generated several paths over this surface, from several distinct initial positions to a predefined goal position. In all cases with this surface the algorithm found a smooth path to reach the goal position, despite its complexity.

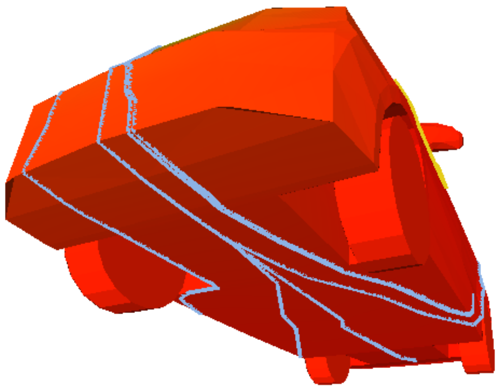
In another experiment, we used a model of the *Mother and Child* statue (see Figure 5.4) to generate paths over this surface. The model has several genus, which also makes it a good example of the kind of environment that our technique deals with. In Figures 5.4(a) and (c), our planner has generated quality and smooth paths to reach the goal position. In Figures 5.4(b) and (d) we used the same initial and goal positions of (a) and (c), respectively, but we also defined some regions where the path could not crossover, simulating obstacles over the surface. The algorithm demonstrated to be able to find quality and smooth paths.



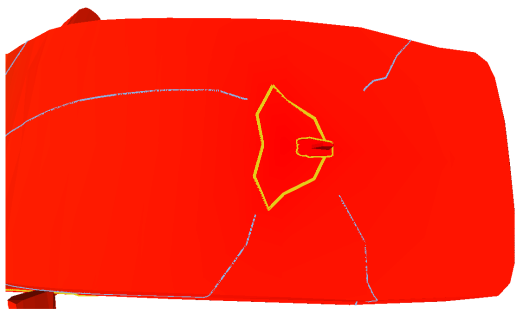
(a)



(b)



(c)



(d)

Figure 5.2: Paths produced over the model of a car. (a), (b), (c) and (d) are different views of the same paths produced with the same set of obstacles and goal. Note in (d) how different the paths are according to the initial position around the obstacles.

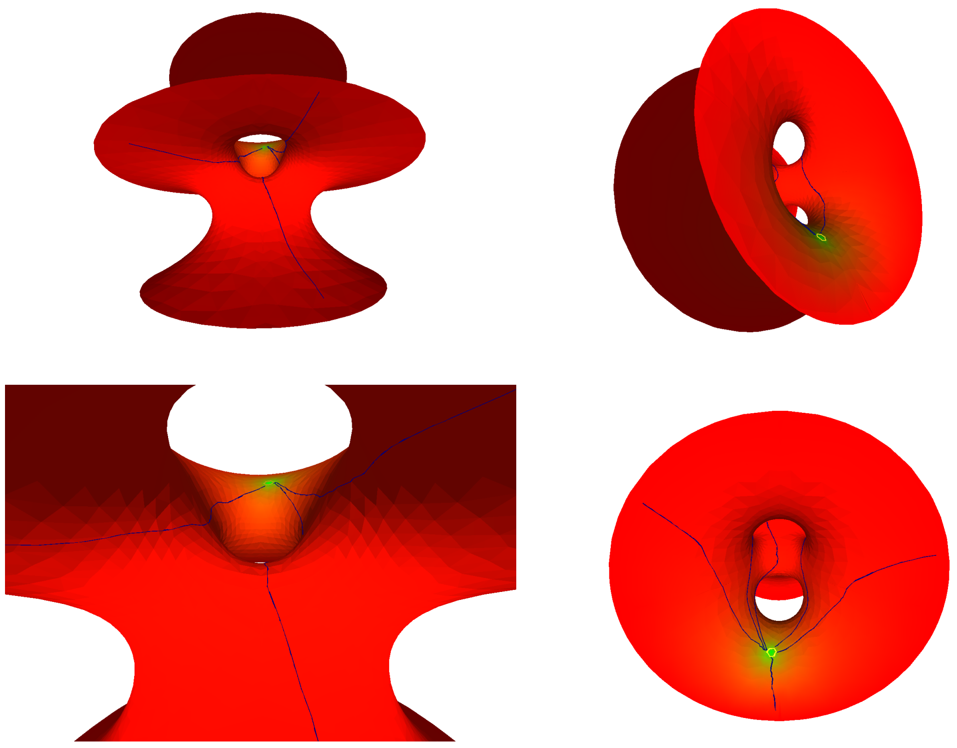


Figure 5.3: Paths produced over the Costa Minimal Surface (COSTA, 1984). Note how lines starting at different points on the surface smoothly reach the goal position (green).

5.3 Local Minima Avoidance

As explained in Section 2.2, a local minima is a region on the environment where the agent reached a stable configuration without reaching the goal position. Connolly et al. (CONNOLLY; BURNS; WEISS, 1990) showed that the Laplace's Equation does not present local minima. As said before, our technique is local minima free. This is due to the behavior of the 3DS-BVP, which mimics the potential fields generated by the BVP Path Planner.

This property can be easily perceived in Figure 5.5. In it, the same environment was presented in a regular and a noisy triangle mesh, built in the same way as the meshes in Section 5.1 was. In both cases the generated paths could avoid the local minima in the center of the environment.

An example of local minima avoidance in a 3D surface is already presented in Figure 5.4. In (a), the path planner found that the best path to follow is through the "arm" of the statue. But in (b), the 3DS-BVP was able to avoid local minima in that position of the statue, by following a different trajectory through the top of the statue. A similar behavior can be noted between (c) and (d).

5.4 Performance evaluation

We measured the performance of our algorithm with the models presented in Section 5.2. For our tests, we measured the time spent to compute the potential field, and the number of iterations needed to converge with different threshold errors. Each model was executed with a given threshold error for 5 times. The average times and number of

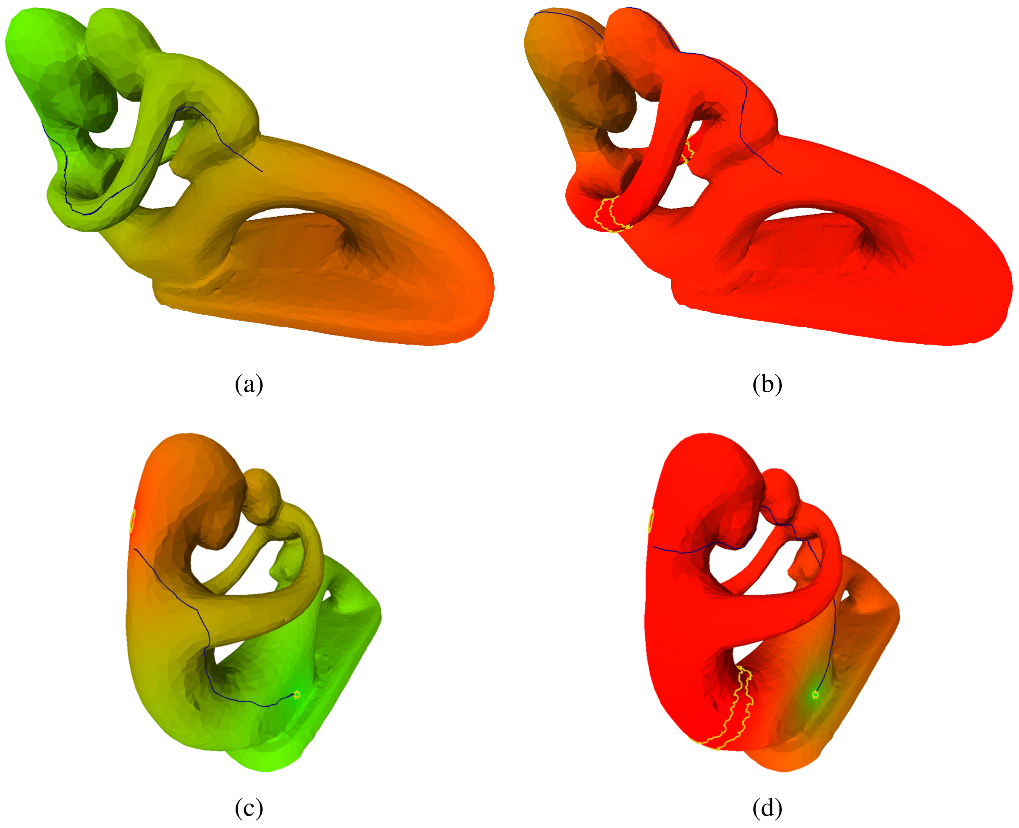


Figure 5.4: Several paths produced over the *Mother and Child* statue, which has several genus. Note that in (b) the initial and goal positions are the same as (a), but some obstacles resulted in a different path. The same occurs in (c) and (d).

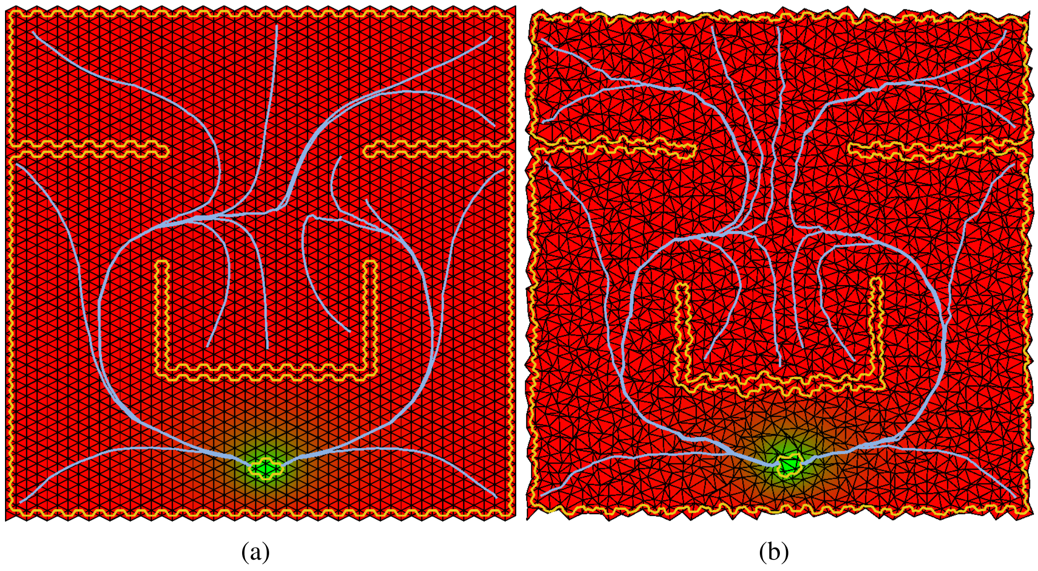


Figure 5.5: Local minima avoidance in a 2D environment with the 3DS-BVP Path Planner.

Model	Faces	Vertices/Cells	Error Thresh.	Iterations	Time (s)
Car	1,292	665	0.010	1,090	0.0473
			0.005	1,286	0.0551
			0.001	1,739	0.0743
Costa Surface	2,320	1,259	0.010	781	0.0587
			0.005	938	0.0670
			0.001	1,304	0.0928
Statue	10,000	4,994	0.010	16,195	5.8936
			0.005	19,267	7.0064
			0.001	26,401	9.6050

Table 5.2: Performance evaluation on three test cases.

iterations are presented in Table 5.2. The tests were executed in an Intel® Core(TM) i7 CPU 960 @ 3.20GHz, with 8GB of RAM memory, a graphics card NVidia GeForce GTX 470, running Windows® 7 Professional 64 Bits.

Despite the quality and smoothness of paths generated in the previous tests, our performance evaluation shows that there is still room for improvements. For smaller and simple models (the Car model, for example) or more complex but still small models (like the Costa Surface), our algorithm is able to solve the potential field and produce a quality path for iterative applications. Based on the times presented on Table 5.2, for these cases our algorithm could achieve an update rate of up to 21 updates per second using an error threshold equals to 0.01. For the Costa Surface model, using an error threshold of 0.001, the update rates are of about 10 updates per second.

Although these rates are sufficiently good for iterative applications, they also show that our algorithm is not yet able to answer queries for real time applications (i.e. that require update rates of 30 frames per second or more). Moreover, for use with more detailed models (like the Statue model) our algorithm still requires improvements in its performance, even for minimally interactive update rates.

Note that the measured times presented in Table 5.2 refers to the relaxation step only. Drawing the path from a given initial position to the goal does not impact on the algorithm performance, since the potential field is computed. This is because: (i) the computation of the gradient of a triangle (Equation 4.14) is very simple; (ii) the Equation 4.14 must be solved only once for each triangle in the path; (iii) the DCEL has the capability to answer the neighbor of a triangle in one simple query. This way, if the target position does not need to be updated frequently, once the relaxation step is complete, the path planner can be used in real time to draw paths.

6 LIMITATIONS AND DEGENERATED CASES

In Chapter 5 we presented a series of tests made to evaluate some aspects of our algorithm and validated it. However, we do not effectively exploit the limitations of our algorithm. In this chapter we will present the known degenerated cases (cases not handled by our algorithm due to anomalous geometry on the triangle mesh), as well as some cases where the algorithm presents some unexpected behavior (such as triangle meshes with acute triangles). Finally, we show that the 3DS-BVP presented the *flatness* problem, which is already presented in the BVP Path Planner.

6.1 Degenerated cases

The first degenerated case can be inferred from the Equation 4.4. This case will happen if at least two neighbor vertices of the triangle mesh have eventually the same coordinates. Suppose that a vertex v_0 has neighbors v_a and v_b. v_0 and v_a are in the same position, while v_b is the farthest neighbor of v_0. From this, v_0 will have the following properties:

- $D_{min}(v_0) = 0$, since v_0 and v_a have the same position;
- $D_{max}(v_0) = Dist(v_0, v_b)$, since v_b is the farthest vertex from v_0.

From this, if we solve the Equation 4.4 for v_a and v_b related to v_0, we get:

- $I(v_0, v_a) = 0 + Dist(v_0, v_b) - 0 = Dist(v_0, v_b)$
- $I(v_0, v_b) = 0 + Dist(v_0, v_b) - Dist(v_0, v_b) = 0$

This way, if two neighbor vertices v_0 and v_a have the same position, the influence of the vertex v_b at a distance equals to $D_{max}(v_0)$ will be equals to 0. If v_0 has other neighbors at a distance $D_{max}(v_0)$, these vertices will also have an influence equals to 0. This results that any potential from these vertices will not be propagated to the vertex v_0, inflicting undetermined behaviors to the final generated paths, since low quality paths to existing paths will not be found. The influence of any vertex v_n with $D_{min}(v_0) < Dist(v_0, v_n) < D_{max}(v_0)$ is not canceled.

A second degenerated case would happen if the input triangle mesh has an independent vertex, not connected with any real triangle in the mesh. Suppose that a vertex v is not connected to the rest of the triangle mesh. Then, the Equation 4.5 will result in $I_{total}(Cell(v)) = 0$. This will lead to an undetermined value result to of Equation 4.6. In fact, if a vertex is disconnected from the triangle mesh, it should be ignored in the algorithm, as it will not affect any produced path.

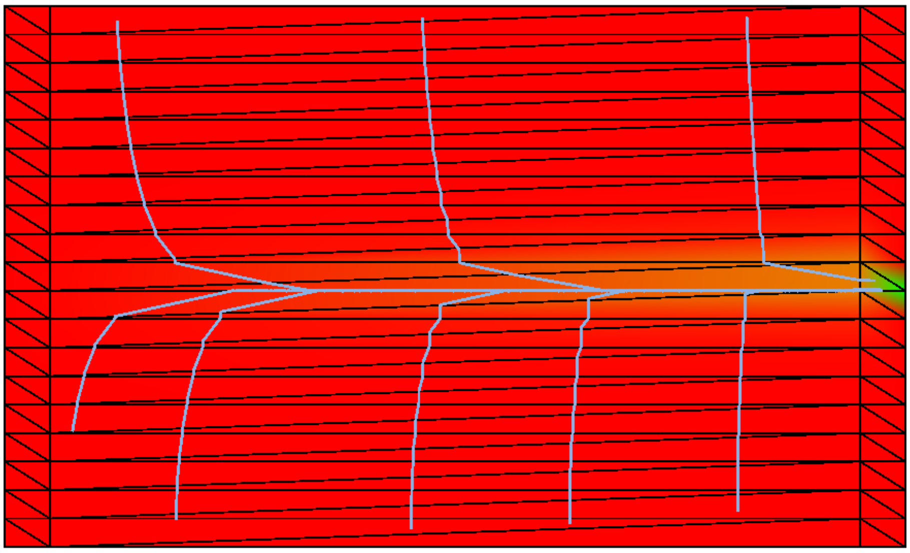


Figure 6.1: Paths generated on a low quality triangle mesh.

The last degenerated case occurs with degenerated triangles (with collinear vertices, for example.) If a triangle has its area equals to 0, the result of the Equation 4.14 is undefined, which results in a triangle with undefined gradient. As said in Section 4.2, the gradient of a triangle is what guides an agent to its goal. Thus, if the mesh has degenerated triangles, some paths may not be found due to the missing gradient. In order to avoid these problems, we suggest a regularization of the triangle mesh before the application of the path planning algorithm.

6.2 Limitations with low quality triangle meshes

As said when evaluating the quality of the paths on noisy meshes (see Section 5.1), the smoothness of the generated paths is affected by the quality of the triangle mesh. For this text, a good quality mesh is understood as a triangle mesh composed with a very small number of poorly shaped or degenerated triangles. Pébay (PéBAY; BAKER, 2001) developed a study on several different metrics to compute the quality of a triangle mesh.

In order to spot some of the cases where our algorithm could fail, either by not producing good paths or by not being able to find any path at all, we developed a set of tests. These tests where composed by triangle meshes with very specific characteristics, like very poorly shaped triangles for example. Although may exist other cases not covered by us that may lead our algorithm to an incorrect result, the cases presented here constitute a comprehensive set of cases that can be used for further development and expansion of the algorithm limitations.

On the first test, we developed an artificial planar mesh with several acute triangles. This was intended to analyze the behavior of the algorithm when this kind of triangle appears on a triangle mesh. In our test, only one cell at one border was tagged as *goal*, and several paths were generated on this mesh. The mesh and the generated paths are illustrated on Figure 6.1.

In this scenario our algorithm was still able to find paths on this triangle mesh. However, a transition between two triangles with high differences between their gradients generates abrupt changes in the path. One possible solution to this is to use better dis-

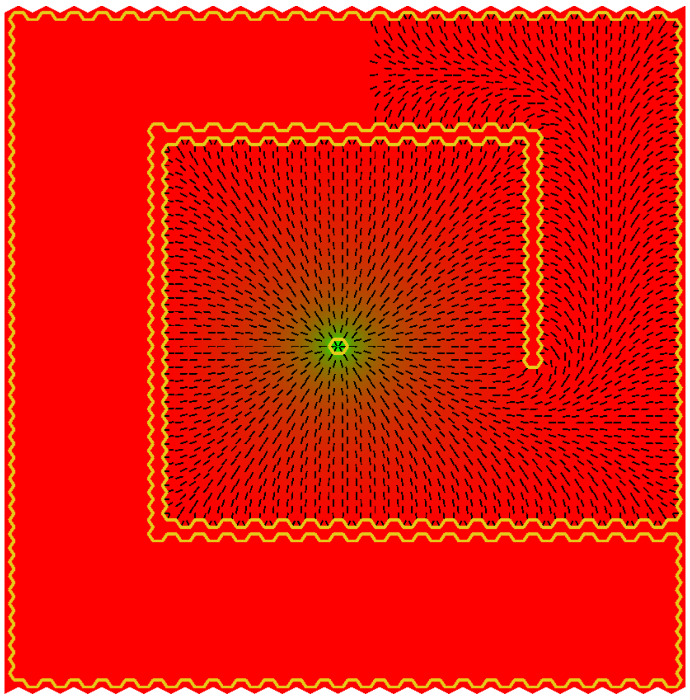


Figure 6.2: Environment where the flatness problem occurs. The black lines represent the not-null gradient.

cretizations for the environment, with more regular triangles. Another solution would be the addition of a smoothing phase after the production of the paths.

6.3 The Flatness Problem

As the BVP Path Planner, the 3DS-BVP presents the *flatness* problem. Presented by Silveira et al. (SILVEIRA et al., 2009), it arises from the finite numeric precision of current numeric floating point types. The Laplace's Equation and Equation 3.1 do not present local minima. But for some configurations of the boundary conditions, the exact result of Equation 3.1 for some cells require arbitrary precision in order to generate a non-null gradient. This happens due to the distribution of potential values generated by the equation itself, not by the scale used between different boundary conditions. Our method presented a similar problem in the kind of 2D environment that the BVP Path Planner presented this problem.

Figure 6.2 illustrates a case where the gradient of the potential field becomes null in some part of the environment. In this figure, the small black lines represent the valid gradient and its direction. It is possible to see that at some point of the hallway the gradient becomes null (i.e. the areas with no black lines).

Silveira et al. (SILVEIRA et al., 2009) proposed a palliative solution to this problem by placing intermediary goals on the hallway, in the points where the gradient starts to became null. Although we did not explored this approach, it seems to be easy to adapt his solution to the 3DS-BVP.

We cannot reproduce this problem in our available surfaces, such as the car model, the Costa Surface or the Statue. The way that the triangles of these meshes relate to each other in these models should be the factor that minimized the flatness problem. We believe that in other cases (possibly with more triangles and *occupied* cells) this problem should emerge more visibly.

7 CONCLUSIONS AND FUTURE WORK

In this chapter, we will present a summary of the path planner developed here, its main advantages and drawbacks. We will present also a set of possible future paths that can be followed to improve our work. Finally, we present other contributions related to this Master Thesis.

7.1 Conclusions

In this work, we presented the 3DS-BVP, an algorithm based on potential fields for path planning in arbitrary surfaces. This technique is inspired by the solution of the BVP Path Planner, developed by Silveira et al. (SILVEIRA et al., 2009). Our algorithm outperforms previous potential field based algorithms, as these techniques do not present a solution for the arbitrary 3D surface case. Moreover, it is an alternative to graph based algorithms, as well as geodesic distance fields based algorithms, by generating smooth paths that minimize collision with obstacles on the surface. This work was published in the *XXIV SIBGRAPI – Conference on Graphics, Patterns and Images* (FISCHER; NEDEL, 2011).

This algorithm is based on a numerical method developed in this work that approximates the results of the Laplace's Equation. The main advantage of this method is the use of a triangular discretization, which could easily be adapted to work in 3D arbitrary surfaces. The gradient descent of the resulting potential fields is used to guide agents on the surface, from an initial position to a specific goal position.

By mimicking the potential fields produced by the BVP Path Planner, our algorithm generates smooth paths free of local minima. The potential field approach also gives our algorithm the ability to find any existing path to a given goal position without the need to execute the query for each initial position. Finally, it is based on a simple representation of the 3D surface (a triangle mesh), and does not require any kind of preprocessing phase, which can lead to future development of techniques that support dynamic surfaces.

Comparing our approach to the BVP Path Planner, it showed that the paths were quite similar, presenting suavity while avoiding obstacles. When the potential fields are compared, the noisy meshes presented only an average difference of 1% and a standard deviation of 1.1%. The regular triangulation, as expected, presented better results, with an average difference of 0.786% and a standard deviation of 0.781%. The maximum differences found were of 7.4% and 36.5% on the regular and noisy cases, respectively. Although these differences were quite large, they do not appear to significantly influence the paths generated. In other words, these comparisons show the potential of our technique to reproduce the quality of the paths generated by the BVP Planner in a triangular representation of the environment.

As this method is applicable to both 2D and 3D meshes with minor modifications, this work also presented results of path generation over surfaces with different levels of complexity. The simplest surface was the model of a car. The second surface, with a significant complexity but a small number of triangles, was the Costa Minimal surface. The third surface, with a high complexity structure and a high number of triangles was the Mother and Child Statue. In all cases, our algorithm was able to find smooth paths.

Concerning the performance, our algorithm still has room for improvements. For the smaller models (the car, with 665 cells, and the Costa Minimal Surface, with $1,259$ cells) our algorithm was able to compute the potential field in less than 0.1 seconds, which is sufficiently good for interactive applications. However, it was not good enough for real time applications. Our algorithm took on average more than 0.04 seconds. A real time application running at 30 frames per second would require a potential field computational time of up to 0.033 seconds. When looking at surfaces with higher number of cells, our algorithm still requires a lot of improvement. In our tests, a surface with $4,994$ cells required at least 5.89 seconds to compute a potential field. As said in Section 5.4, these times are only related to the potential field computation. If the goal position do not need to be frequently updated, the algorithm can be used in real time as soon as the potential field is computed.

7.2 Future work

Although this work was successful on the development of a path planner for arbitrary surfaces, there are several improvements that could be developed in order to increase the capabilities of the 3DS-BVP.

One limitation of the 3DS-BVP is that it assumes that the agent has its height equals to 0 (i.e. it resides strictly on the surface). While in this case our algorithm will generate valid paths, it may not be true if the agent has height greater than 0. In this case, an agent that walks on the surface could collide with other points of the surface. For example, suppose the agent a walking over a surface s in the direction d, as illustrated in the Figure 7.1. As the agent has a height h_1, which is greater than h_2, the agent could not travel through that region of the surface, since a collision would happen there. If not solved, this problem could also happen in future multi-agent implementations of our algorithm. A possible solution for this problem was presented by Torchelsen et al. (TORCHELSEN et al., 2010), which requires an additional 3D grid.

Another addition that could be added to the algorithm would be the development of a local path planner approach, similar to the one introduced by Dapper (DAPPER; NEDEL; JUNIOR, 2007) in the BVP Path Planner (Section 3.3). As occurred in the BVP Path Planner, this could add a great increase on the performance for multi-agent environments, as well as allow dynamic obstacles on the environment. So far, the biggest problem to develop the local maps approach is the development of an algorithm that dynamically selects and keeps the part of the triangle mesh around a given point, while the point is moving over the triangle mesh. More research should be done to known if such algorithm already exists or must be developed.

There could be developed a method to define the most adequate error threshold to compare against the result of Equation 4.7. Although the exact solution is not required, at least a minimal error needs to be achieved during the relaxation process. If this error is larger than this, the potential may not be able to properly guide the agent towards its goal. If this threshold is smaller than the needed, the relaxation process may take unnecessary

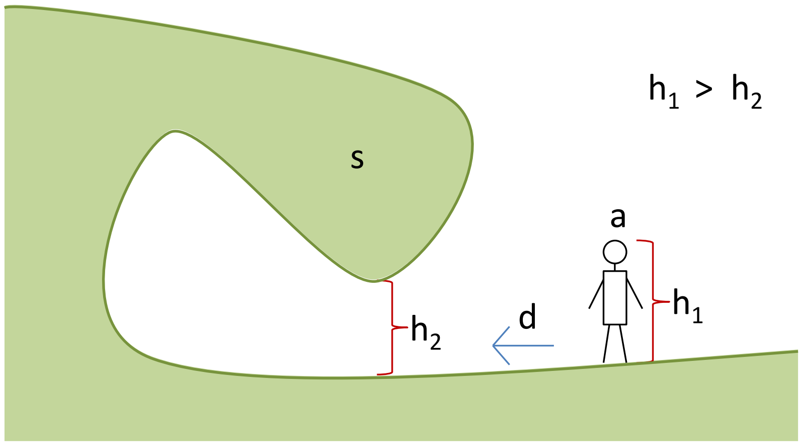


Figure 7.1: Collision between an agent a walking over a surface s and the surface itself.

iterations before achieving convergence.

Also, a third addition to the algorithm would be the improvement of the set of equations used in this work. This should be done in order to reduce the number of iterations needed to reach the convergence of the potential field. This will add and *implementation independent* increase of performance to the algorithm. One possible way to improve our equations could start by a study on the influence between two cells. A suggestion would be to add α, β and γ parameters to the terms of the Equation 4.4 and check how the modification of these parameters impact on the number of iterations. Another way to improve our algorithm would be conducted by an study on the initial value of *free* cells, as did Silveira et al. (SILVEIRA; PRESTES; NEDEL, 2008).

There are also *implementation dependent* optimizations that could be done. It seems that our algorithm is highly parallelizable, which would benefit greatly by a GPU implementation. Due to the distinct discretization methods (grid versus triangle meshes), the adaptation of optimizations from the BVP Path Planner to the 3DS-BVP is not simple. Besides the implementation of our equations in a GPU system, it will be required also efficient GPU data structures to hold the triangle mesh for the relaxation step.

Finally, the order that cells are evaluated reflects in the speed that the potential values from the obstacles and goals are propagated to the free cells. So, there should be an ideal order that makes the relaxation process faster.

7.3 Additional contributions

During the development of this work, several other contributions were made, through several published work. Although these contributions do not relate directly to the algorithm presented here, it was highly important to understand the BVP-Path-Planner and, thus, develop the 3DS-BVP.

In Fischer et al. (FISCHER; SILVEIRA; NEDEL, 2009), the processing power of current graphic processors (GPUs) was used to compute the potential fields for the local maps in the BVP Path Planner. There are two main contributions in this work: (i) a parallel version of the BVP-Path-Planner, implemented on the GPU utilizing NVIDIA CUDA (Compute Unified Device Architecture) (NVIDIA, 2008), and (ii) a strategy to reduce the number of memory transactions between the CPU and GPU. With the GPU strategy we achieve a speedup up to 56 times comparing with the previous technique. This

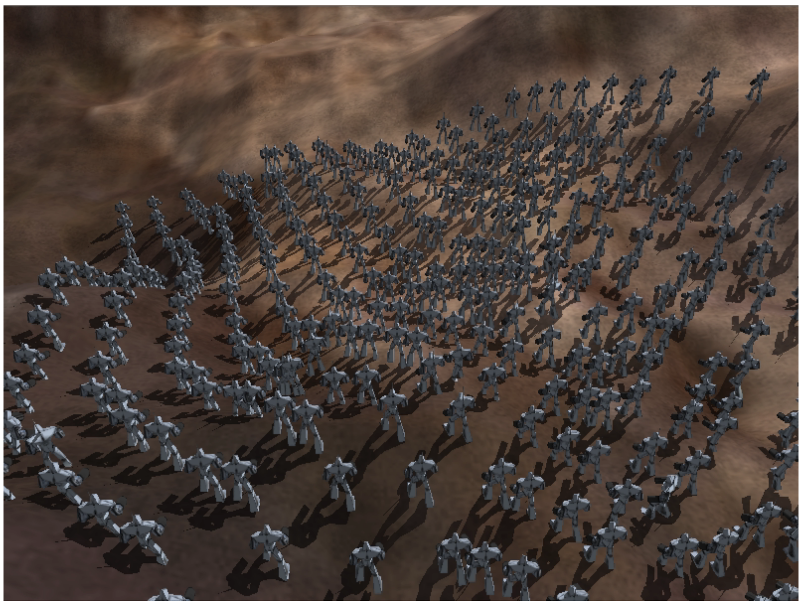


Figure 7.2: A picture of a scene with 500 autonomous agents walking using the GPU approach developed in Fischer et al. (FISCHER; SILVEIRA; NEDEL, 2009).

performance speedup allowed its use in situations with a large number of autonomous characters, which is a common situation found in games. Figure 7.2 shows a scene with 500 agents walking using the BVP-Path-Planner.

In Silveira et al. (SILVEIRA et al., 2010) the BVP Path Planner is presented in details, focusing on the behavior of the agents acting on the scene. The techniques developed during this work were implemented and evaluated within a RTS game engine.

In Fischer et al. (FISCHER et al., 2011) the solution for an complex interactive task is presented. In it, three techniques were combined: (i) a point-and-click approach for navigation in virtual environments, (ii) a ray cast selection approach to select objects, and (iii) a device-based approach to control the orientation of objects in a virtual scene. The transition between the techniques was developed in such a way that the user naturally changes between the techniques, without menus or buttons explicitly changing the current interaction technique. These techniques were applied in a virtual market, and the users were asked to find specific products between the ones available in the market shelves. Several user experiments were made to evaluate the developed techniques. The work was selected between the best eight articles, and was invited to submit a new version to the *Elsevier Computers & Graphics Journal*. The new version was submitted, and is currently in process of evaluation.

8 RESUMO EXPANDIDO

Navegação em superfícies tri-dimensionais é um problema relevante para muitas áreas como: visualização científica, onde um usuário precisa inspecionar diferentes tipos de objetos, como órgãos em uma aplicação médica, ou motores em um sistema CAD; em robótica, com a definição de caminhos e movimentos; e entretenimento, mais especificamente na área de jogos de video-game, onde a exploração de ambientes 3D complexos é muito mais desafiadora para o jogador do que simples ambientes planares.

Navegação é uma tarefa interativa complexa e é usualmente dividida em duas partes (BOWMAN et al., 2004): *planejamento de caminho* e *movimentação*. Enquanto a movimentação é o componente motor da navegação, as ações de baixo nível que fazem o usuário controlar a posição e orientação de seu ponto de vista, o planejamento de caminhos é a componente cognitivo, e inclui raciocínio de alto nível, planejamento e tomada de decisão. Ela inclui o entendimento do espaço e o planejamento de tarefas, como determinar o caminho a partir da posição atual para a posição objetivo.

Os algoritmos de planejamento de caminhos têm sido explorados por anos. Muitas soluções foram aplicadas na robótica e em ambientes virtuais, sendo que algumas delas focam na performance do algoritmo – normalmente, incluindo uma fase de pré-processamento (CALOMENI; CELES, 2006) –, e outras em prover caminhos de melhor qualidade. Apesar de que a maioria destes algoritmos resolverem o problema no plano Euclideano, alguns destes são robustos o suficiente para tratar sistemas com mais de dois graus de liberdade (como o planejamento de caminhos em 3D (CARSTEN; FERGUSON; STENTZ, 2006) ou planejamento de caminhos para robôs articulados com muitas juntas (BELGHITH et al., 2006)). Porém, planejamento de caminhos restritos à superfícies arbitrárias não são muito exploradas na literatura.

Métodos que focam especificamente no planejamento de caminhos 2D não podem ser trivialmente modificados para tratar superfícies arbitrárias. Uma abordagem possível é o uso de técnicas de projeção sofisticadas da superfície 3D no plano Euclideano, e então modificar o algoritmo de planejamento de caminhos 2D para trabalhar nesta projeção, o que não é uma tarefa trivial (devido à natureza da projeção). Algoritmos que tratam superfícies 3D dependem da sua natureza para serem adaptados a superfícies 3D, já que em um determinado ponto da superfície o algoritmo deve se comportar como se fosse um planejador de caminhos 2D. Abordagens baseadas em grafos são rápidas o suficiente para o uso em aplicações de tempo real, mas os caminhos gerados não são suaves como as outras abordagens. Em todos estes casos, o trabalho requerido para portar o algoritmo é relevante, e não é claro como estes algoritmos irão se comportar neste tipo de ambiente.

Neste trabalho foi desenvolvida uma solução para a segunda parte da navegação em superfícies arbitrárias, o planejamento de caminhos. Iremos apresentar uma nova técnica para planejamento de caminhos que trata o caso em superfícies arbitrárias, chamada

3DS-BVP, um acrônimo para *3D Surface Path Planner using Boundary Value Problems System –Sistema Planejador de Caminhos em Superfícies 3D usando Problemas de Valor de Contorno*. A técnica usa problemas de valor de contorno (PVC) para gerar campos potenciais usando uma discretização em malhas de triângulos. Utilizando o gradiente do campo potencial, o agente pode se guiar pelo ambiente. Sumariamente, as contribuições deste trabalho são:

- Um método numérico que gera campos potenciais utilizando uma discretização em malhas de triângulos;
- Um planejador de caminhos baseado em campos potenciais que calcula caminhos suaves em malhas de triângulos 3D.

Esta técnica é baseada em um planejador de caminhos que é capaz de gerar caminhos suaves com pouca probabilidade de colisão com obstáculos a partir de campos potenciais.

8.1 Trabalhos relacionados

Algoritmos planejadores de caminho tem sido utilizados para calcular o caminho a ser percorrido até uma posição objetivo em um ambiente virtual. Muitos algoritmos tem sido propostos para resolver este problema, e grande parte deles assume que o ambiente pode ser projetado em uma superfície 2D.

Kallmann (KALLMANN, 2005) usou triangulações de Delaunay com restrições para discretizar o espaço livre do ambiente em uma malha de triangulos e uma abordagem em grafo para buscar o caminho. Ele ainda propôs um método para o planejamento de caminhos em ambientes com largura mínima (KALLMANN, 2010). Apesar do fato destes métodos usarem malhas de triangulos como estrutura de dados, eles foram desenvolvidos apenas para uso em ambientes planares. Nosso método trata com indiferença a diferença entre superfícies planares e 3D.

Tecnicas baseadas em campos potenciais para navegação incluem o trabalho de Rosell e Iniguez (IñIGUEZ; ROSELLY, 2003), Trevisan et al. (TREVISAN et al., 2006), Treuille et al. (TREUILLE; COOPER; POPOVIć, 2006), e Park (PARK, 2010). Estas técnicas usam a posição dos obstáculos e dos agentes para calcular uma função. O resultado é um campo de onde as direções para uma determinada posição são derivadas. Estas técnicas diferem entre si pela função utilizada para calcular o campo e como as direções são derivadas, resultando em diferentes comportamentos para cada técnica. Por exemplo, o trabalho de Trevisan et al. Favorece o comportamento exploratório do agente, enquanto que o trabalho de Treuille et al. Favorece seu uso em simulações de multidões. Todas estas técnicas foram desenvolvidas para ambientes 2D. Algumas delas podem ser aplicadas para ambientes 3D ao adicionar uma dimensão às suas equações, mas isto degrada significativamente sua performance.

Um planejador de caminhos baseado em distâncias geodésicas em malhas de triangulos 3D foi recentemente proposto por Torchelsen et al. (TORCHELSEN et al., 2010). Este trabalho foca em sistemas multi-agente, e usa uma arquitetura CPU/GPU para tratar do desvio de colisões entre os agentes. A maior vantagem deste método é a alto desempenho alcançado. Por outro lado, os caminhos gerados são próximos aos menores caminhos, o que pode levar a uma alta probabilidade de colisão dos agentes com os obstáculos. O nosso planejador de caminhos produz caminhos suaves que, sempre que possível evita aproximar-se muito dos obstáculos.

Devido à performance e baixas necessidades de memória, as abordagens baseadas em grafos são as mais utilizadas na industria de vídeo games. Motores de jogos populares, como o Unreal Engine® e CryEngine® utilizam a abordagem. Nestes métodos, um grafo representa o ambiente e o algoritmo de Dijkstra (DIJKSTRA, 1959) (ou uma de suas derivações) é utilizada para encontrar o caminho entre dois nodos. A diferença entre estas abordagens (como a proposta por Kavraki et al. (KAVRAKI et al., 1996), Barraquand et al. (BARRAQUAND et al., 1997), Lavalle (LAVALLE, 1998), e Kang et al. (KANG; KIM; KIM, 2010)) é o algoritmo utilizado para amostrar o ambiente em um grafo e como este grafo é atualizado. Todos estes métodos parecem ser adaptáveis ao planejamento de caminhos em superfícies 2D e 3D, mas eles não tem sido explorados para planejamento de caminhos em superfícies arbitrárias.

8.2 O Planejador de Caminhos BVP

O planejador de caminhos BVP (TREVISAN et al., 2006) é um planejador 2D que gera caminhos utilizando a informação de potencial calculada a partir da solução numérica da Equação 3.1 com condições de contorno de Dirichlet, onde $\mathbf{v} \in \Re^2$ e $|\mathbf{v}| = \mathbf{1}$ corresponde a um vetor que insere uma perturbação no campo potencial; $\epsilon \in \Re$ corresponde à intensidade da perturbação produzida por $\mathbf{v}$; e $p(\mathbf{r})$ é o potencial na posição $\mathbf{r} \in \Re^2$, respectivamente. Ambos $\mathbf{v}$ e ϵ devem ser definidos antes de resolver esta equação. O gradiente descente destes potenciais representam rotas navegáveis a partir de qualquer ponto do ambiente até a posição objetivo. Trevisan et al. (TREVISAN et al., 2006) demonstra que esta equação não produz mínimos locais e gera caminhos suaves.

Para resolver numericamente um PVC, considera-se que a solução é discretizada em um grid regular ((TREVISAN et al., 2006; SILVEIRA et al., 2009)). Cada célula (i, j) é associada a uma região quadrada do ambiente e armazena um valor de potencial $p(i, j)$. Utilizando as condições de contorno de Dirichlet, as células associadas aos obstáculos no ambiente armazenam um valor de potencial igual a 1 (*alto potencial*) enquanto que as células contendo a posição objetivo armazenam um valor de potencial igual a 0 (*baixo potencial*).

Um alto valor de potencial previne que o agente avance em direção aos obstáculos, enquanto que o baixo valor de potencial gera uma atração ao agente. O método de relaxamento utilizado para calcular os campos potenciais nas áreas livres do ambiente é o Gauss-Seidel (GS). O método GS atualiza o potencial em uma célula c a partir da equação 3.2, onde $\mathbf{v} = (v_x, v_y)$, e p_c, p_b, p_t, p_r and p_l são células do grid, conforme ilustrado na Figura 3.1.

O método GS permite o uso de resultados parciais como uma aproximação do campo potencial (PRESTES et al., 2001). Como o resultado exato não é necessário, é possível controlar o erro acumulado $e(t)$ em cada iteração através de um limiar e_{max} de acordo com a Equação 3.3, onde $p(i, j)^t$ é o potencial em uma célula (i, j) na iteração t, $p(i, j)^{t-1}$ é o potencial na mesma célula na iteração anterior, e m e n são as dimensões da grade regular.

Após o calculo do campo potencial, o agente se move na direção do gradiente descente na sua posição atual (i, j).

8.3 Planejamento de caminhos em malhas de triângulos

Nesta proposta, o 3DS-BVP gera campos potenciais em malhas de triangulos. O método funciona de acordo com os seguintes passos: (1) discretizar o ambiente em um conjunto de células; (2) calcular o campo potencial; (3) calcular o gradiente deste campo potencial.

8.3.1 Discretização do ambiente

O primeiro passo do algoritmo 3DS-BVP funciona da seguinte forma. A cada vértice da malha de triangulos é associado um valor de potencial – da mesma forma que é feito no Planejador de Caminhos BVP. Então, cada triangulo é dividido em três regiões, conectando as medianas das arestas ao centroide do triangulo. Cada região do triangulo é associada ao vértice mais próximo e, para um dado vértice v de um triangulo t, a função $Region(v, t)$ retorna aquela região associada. Além disso, o algoritmo assume que cada vértice v na malha de triangulos é associada a um conjunto de triangulos $Triangles(v)$, onde cada triangulo em $Triangles(v)$ possui um vértice igual a v (ou seja, eles compartilham o mesmo vértice). A célula associada a um vértice v é definida pela função $Cell(v) = \{Region(v, t) | \forall t \in Triangles(v)\}$. A função $Vertex(c)$ retorna o vertice associado à célula c. A Figura 4.2 ilustra estes conceitos.

O conceito de *célula vizinha* é definida por duas células cujos vértices associados são conectados por uma única aresta. Isto significa que duas células c_1 e c_2 são vizinhas se a função $Link(Vertex(c_1), Vertex(c_2)) \neq null$ satisfaz. Assumindo que o conjunto C contem todas as células derivadas da malha de triangulos, a função $Neighbors(c_i) = \{c_j | \forall c_j \in C, i \neq j, Link(c_i, c_j) \neq null\}$ retorna o conjunto de células vizinhas para uma dada célula c_i.

8.3.2 Calculando o campo potencial

Para executar o relaxamento no conjunto de células, as condições de contorno precisam ser definidas. Estas condições de contorno são definidas de acordo com as posições dos obstáculos e o objetivo no ambiente. Assume-se que os obstáculos e a posição objetivos estão restritos à superfície. No Planejador de Caminhos BVP, cada célula recebe uma etiqueta em valor de potencial inicial, da seguinte forma:

- células associadas às regiões ocupadas do ambiente e as células associadas com as bordas da superfície recebem a marca *ocupada* e recebem um *alto valor de potencial*;

- a célula associada à posição objetivo recebe uma marca *objetivo* e um *baixo valor de potencial*;

- as células associadas às regiões navegáveis do ambiente recebem uma marca *livre* e recebem um valor intermediário entre *baixo* e *alto* valores de potencial, como no Planejador de Caminhos BVP.

Para atualizar o valor de potencial nas células *livres*, um conjunto de funções foi definido. Assumindo que a função $Dist(v_1, v_2)$ retorna a distancia geodésica entre os vértices v_1 e v_2, $D_{min}(c_i)$ e $D_{max}(c_i)$ retorna a menor e maior distancia, respectivamente, entre o vértice de uma célula c_i e as células vizinhas.

A influencia de uma célula c_j sobre uma célula c_i, relativa à vizinhança de c_i é definida pela função 4.4. Baseada na Equação 4.4, a Equação 4.5 calcula a soma das influências que uma célula recebe das células vizinhas. Esta equação é então utilizada na Equação 4.6, que é então utilizada para atualizar o potencial $p(c_i)$ de uma célula c_i *livre*.

O valor de potencial das células *livres* são atualizados utilizando Equação 4.6 até que a convergência seja alcançada, como no Planejador de Caminhos BVP. Um limiar e_{max} é utilizado para verificar se o potencial já convergiu. A Equação 3.3 é utilizada com o conjunto de células C para calcular o erro em uma dada iteração.

8.3.3 Calculando o gradiente do campo potencial

Com o campo potencial calculado, um agente deve ser capaz de seguir o gradiente descente de forma a alcançar a posição objetivo. A Equação 4.6 é capaz de reproduzir os resultados produzidos pela Equação de Laplace. É possível calcular o gradiente da Equação 4.6 de forma semelhante à realizada com a Equação de Laplace.

Uma possivel solução do calculo numérico para o gradiente da Equação de Laplace em malhas de triangulos irregulares envolve o uso da sua forma integral, apresentada na Equação 4.8. Após uma série de manipulações algébricas, é possível descrever uma aproximação para o gradiente em uma região pequena com a Equação 4.10. Esta equação, ao ser calculada de forma discreta em um triangulo t resulta na Equação 4.14, que é utilizada para aproximar o gradiente descente do campo potencial no centroide do triangulo t.

Para mover em direção à posição objetivo do ambiente, o agente a precisa seguir o gradiente descente do triangulo onde ele está. Um ponteiro para este triângulo é armazenado com o agente, para acesso rápido. Quando o agente caminha para fora do triangulo, é verificado qual das bordas do triangulo o agente atravessou, e o ponteiro é atualizado com o triangulo vizinho.

8.4 Campos potenciais em superfícies 3D

Em uma discretização de ambiente completamente planar, o planejador de caminhos apresentado até a Seção 8.3 comporta-se de forma bastante similar ao Planejador de Caminhos BVP. A posição objetivo deve ser definida com uma marca *objetivo* enquanto os obstáculos são definidos com uma marca *ocupado*. Os limites do ambientes também são marcados com uma marca *ocupado* e todas as células restantes são marcadas como *livres*. Todas as células recebem um valor de potencial de acordo com a sua marcação. Então, o método numérico avalia valores mais precisos para as células *livres*. Finalmente, o agente utiliza o gradiente do campo potencial para caminhar pelo ambiente.

Neste tipo de ambiente, o planejador de caminhos tem a vantagem de se utilizar de cálculos mais simples, já que todas as posições de vértices e normais podem ser manipuladas em 2D. Naturalmente, uma discretização 3D vai requerer a terceira coordenada para a posição dos vértices e para as normais.

Para malhas 3D abertas o planejador também funciona com mudanças mínimas, e a adição de um eixo coordenado não introduz mudanças significativas em nenhum dos métodos apresentados. Durante a relaxação, apenas a função $Dist(v_1, v_2)$ é modificada para retornar a distância geodésica em 3D. Adicionalmente, as normais usadas no cálculo do gradiente precisam ser armazenadas em 3D. Ainda assim, as normais mantêm-se paralelas ao plano definido pelo triangulo à qual elas pertencem.

A mudança mais significativa no algoritmo para tratar malhas 3D é o tratamento das condições de contorno. Nosso algoritmo requer que ao menos duas condições de contorno distintas sejam definidas: uma célula *objetivo* e uma célula *ocupada*. Assume-se que a célula *objetivo* sempre pode ser definida, então em malhas 3D devemos tomar um cuidado especial com as células *ocupadas*.

Se a malha 3D tiver bordas, os vértices das bordas irão forçar a existência de células *ocupadas*. Malhas de triângulos fechadas com obstáculos na superfície também irão forçar a existência destas células. Isto irá gerar um gradiente válido que previne que o agente saia dos limites da superfície, colida com obstáculos, e o guie para a posição objetivo.

Porém, se a malha for fechada e não tiver nenhum obstáculo na sua superfície, este algoritmo vai gerar apenas células *objetivo*. Se isto ocorrer, a relaxação vai parar de ser executada apenas quando todas as células tiverem seu valor de potencial equal ao *baixo valor de potencial*. Isto irá resultar em um gradiente nulo (o resultado da Equação 4.14 será $(0, 0)$), e o agente não terá como decidir a direção a ser seguida.

Para evitar a ocorrência de gradiente nulo nestes casos, a célula contendo a posição inicial de navegação do agente é marcada como *ocupada* e recebe um *alto valor de potencial*. Apesar de a célula não conter nenhum obstáculo, isto irá forçar com que o campo potencial tenha um gradiente capaz de guiar o agente até a posição objetivo. Além disso, como o agente estará deixando aquela posição, o obstáculo adicionado não irá modificar a existência de um caminho partindo da posição inicial até a posição objetivo.

Note que o caminho tomado pelo agente pode mudar significativamente de acordo com a posição inicial do agente dentro da célula modificada. Como a célula correspondente à posição inicial é modificada para receber a marcação *ocupada* e o *alto valor de potencial*, o gradiente na borda da célula vai ser significativamente diferente de acordo com o triangulo onde é avaliado. O fato de a malha ser fechada e não possuir outras bordas permite que ocorra a situação onde diversos caminhos diferentes possam ser tomados para alcançar o objetivo.

8.5 Resultados

Para avaliar este trabalho, foi desenvolvida uma série de testes. Primeiro, foi realizado uma comparação entre este método em uma malha de triângulos plana, demonstrando que esta técnica é capaz de produzir caminhos com pouca probabilidade de colisão, da mesma forma que o Planejador de Caminhos BVP faz. Então, serão apresentados alguns resultados obtidos com superfícies 3D arbitrárias.

8.5.1 Comparação do método com o Planejador de Caminhos BVP

Foram desenvolvidos um conjunto de casos de teste, onde cada teste é composto por um ambiente retangular com alguns obstáculos e objetivo. Nos testes 1 a 4 o ambiente foi discretizado em uma malha de triângulos regular. Nos testes 5 a 8 foram utilizados os mesmos ambientes dos primeiros 4 testes, mas com a adição de ruído modificando posição de cada vértice (representando melhor os triângulos encontrados em superfícies 3D). A Figura 5.1 ilustra estes casos.

Nestes casos, foi possivel perceber que os caminhos gerados pelo 3DS-BVP são bastante similares aos caminhos produzidos pelo Planejador de Caminhos BVP. Ao adicionar ruído, estes caminhos perderam um pouco de sua suavidade devido à presença dos triângulos de baixa qualidade, mas ainda assim replicam os resultados produzidos pelo Planejador de Caminhos BVP e o 3DS-BVP em malhas de triângulos regulares.

Também foi realizada uma comparação de valor de potencial em cada célula do campo potencial produzido pelo 3DS-BVP com os valores nas células do campo potencial produzido pelo Planejador de Caminhos BVP. A Tabela 5.1 apresenta um sumário destas diferenças. É possível perceber que, em média, o campo potencial gerado pelo nosso método é praticamente o mesmo do gerado pelo Planejador de Caminhos BVP, com uma diferença média abaixo de $0,8\%$ nos casos regulares, e de 1% nos casos com ruído.

A maior diferença encontrada foi de $49,989\%$, encontrada no caso de teste 7. Apesar de esta ser uma diferença relevante entre o 3DS-BVP e o Planejador de Caminhos BVP, ela ocorreu em um ambiente cuja diferença média foi de $1,867\%$ e desvio padrão de $1,717\%$. Esta diferença ocorreu devido à existência de triângulos muito deformados devido ao ruído adicionado. O caso de teste 8 produziu resultados similares.

8.5.2 Avaliação dos caminhos gerados em superficies arbitrárias

O algoritmo foi aplicado a algumas superfícies 3D para analizar como os caminhos são gerados nestas superfícies. Na Figura 5.3, foi aplicado o algoritmo na Costa Minimal Surface (COSTA, 1984), uma superfície paramétrica bastante complexa. Foram gerados diversos caminhos nesta superfície partindo de diversos caminhos distintos até um determinado objetivo. Em todos os casos nesta superfície, o algoritmo foi capaz de gerar um caminho suave.

Em outro experiment, foi utilizado o modelo da estátua *Mãe e Filho* (Figure 5.4) para gerar caminhos na sua superfície. A estátua possui diversas aberturas, o que a faz um bom exemplo do tipo de ambiente que a técnica é capaz de lidar. Conforme pode ser percebido, o planejador gerou caminhos suaves para alcançar a posição objetivo. Também foram realizados testes com algumas regiões onde o caminho não poderia cruzar, simulando obstáculos na superfície. O algoritmo demonstrou ser capaz de encontrar caminhos suaves na superfície.

8.5.3 Avaliação de performance

Foram realizadas medidas de performance no algoritmo em diversos casos, incluindo as superfícies apresentadas no Item 8.5.2. Foram medidos o tempo gasto para calcular o campo potencial e o número de iterações necessárias para convergir, Utilizando diversos limiares de erro. Os resultados estão apresentados na Tabela 5.2.

Apesar de a suavidade dos caminhos gerados nos testes anteriores, as medidas de desempenho demonstram que ainda existe espaço para melhorias. Para modelos complexos, mas pequenos, como a Costa Minimal Surface, o algoritmo é capaz de calcular o campo potencial e produzir caminhos de qualidade para aplicações como, por exemplo, o movimento de câmeras em ambientes virtuais. Para modelos maiores, como a estátua *Mãe e Filho*, o algoritmo ainda precisa de melhorias de performance.

8.5.4 Limitações e casos degenerados

Ao analizar a Equação 4.4 é possível concluir que se, eventualmente, dois vértices estiverem na mesma posição, $D_{min}(c_i)$ será igual a 0, e quaisquer dois vértices com distância $D_{max}(c_i)$ terão nenhuma influência sobre o vértice c_i. Isto pode resultar na interrupção da propagação do potencial de uma região para outra. Além disso, se a malha tiver um triângulo inválido (com os vértices colineares, por exemplo), o resultado da Equação 4.14 é indefinido.

Foram realizados diversos testes com malhas diferentes. Em um experimento com uma malha contendo vários triângulos, os caminhos tinham pouca suavidade. Isto ocorreu devido ao gradiente dos triângulos adjacentes apresentarem grandes diferenças entre eles.

8.6 Conclusões e trabalhos futuros

Neste trabalho foi apresentado o 3DS-BVP, uma técnica baseada em campos potenciais para planejamento de caminhos em superfícies arbitrárias. A principal vantagem desta técnica é a geração de caminhos livres de mínimos locais em superfícies 3D, sem a necessidade de uma parametrização 2D ou outro tipo de representação da superfície.

O campo potencial produzido pelo 3DS-BVP é capaz de gerar todos os caminhos possíveis até uma determinada posição objetivo em um ambiente, seguindo o gradiente descente. Os caminhos gerados em ambientes 2D demonstram ser bastante similares em qualidade aos gerados pelo Planejador de Caminhos BVP, com diferença média no campo potencial em torno de $0,8\%$. Esta é uma boa característica, demonstrando que o 3DS-BVP utiliza como base um campo potencial com características semelhantes àquelas produzidas pelo Planejador de Caminhos BVP.

A performance do algoritmo é a sua principal limitação, por não ser suficiente rápido para uso em aplicações de tempo real, como ambientes com diversos agentes em movimento. Acredita-se que esta limitação possa ser minimizada ao se melhorar as equações utilizadas pelo método e ao se desenvolver uma implementação paralela baseada em GPU dos métodos utilizados para calcular os campos potenciais.

Outras possibilicades de melhorar o algoritmo estão sendo analizadas. Uma possibilidade é melhorar o conjunto de equações que foram utilizadas, de forma a obter o campo potencial em menos iterações. Além disso, a ordem em que as células são avaliadas pode refletir na velocidade com que os valores de potencial das células ocupadas são propagadas para as células livres. Assim, deve haver uma ordem ideal que torna o processo de relaxação mais rápido. Outra possibilidade de melhorar o desempenho do algoritmo é a sua implementação em GPU. O método parece ser bastante paralelizável, assim como aquele realizado no trabalho de Fischer et al. (FISCHER; SILVEIRA; NEDEL, 2009).

Pretende-se aplicar este algoritmo principalmente em ambientes virtuais. Novas ferramentas para o controle de cameras virtuais e aplicações de modelagem podem ser desenvolvidas, de forma a ajudar a visualização e avaliação de modelos 3D. Além disso, muitos jogos de vídeo game, como Prey® e Super Mario Galaxy® o jogador e os inimigos caminham por superfícies arbitrárias para alcançar seus objetivos. Futuros jogos de vídeo game utilizando este tipo de ambiente podem também se beneficiar deste algoritmo.

REFERENCES

3D Realms. **Prey**. Available at <http://en.wikipedia.org/wiki/Prey_(video_game)>. Access in April 26, 2011.

AKLEMAN, E.; CHEN, J. Guaranteeing 2-manifold property for meshes. **Proceedings Shape Modeling International '99. International Conference on Shape Modeling and Applications**, [S.l.], p.18–25, 1999.

AUTODESK. **Maya**. Available at <http://usa.autodesk.com/maya/>. Access in March 29, 2011.

AUTODESK. **3D Studio Max**. Available at <http://usa.autodesk.com/3ds-max/>. Access in March 29, 2011.

BARRAQUAND, J. et al. A Random Sampling Scheme for Path Planning. **The International Journal of Robotics Research**, [S.l.], v.16, n.6, p.759–774, 1997.

BAUMGART, B. G. A polyhedron representation for computer vision. **Proceedings of the May 19-22, 1975, national computer conference and exposition on - AFIPS '75**, New York, New York, USA, p.589, 1975.

BELGHITH, K. et al. Anytime dynamic path-planning with flexible probabilistic roadmaps. **Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.**, [S.l.], p.2372–2377, 2006.

BERG, M. D. et al. **Computational Geometry: algorithms and applications**. 3rd.ed. [S.l.]: Springer-Verlag, 2008. 386p.

BOWMAN, D. A. et al. **3D User Interfaces: theory and practice**. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2004.

BROCK, O.; GRUPEN, R. Exploiting redundancy to implement multiobjective behavior. **2003 IEEE International Conference on Robotics and Automation**, [S.l.], p.3385–3390, 2003.

CALOMENI, A.; CELES, W. Assisted and automatic navigation in black oil reservoir models based on probabilistic roadmaps. **Proceedings of the 2006 symposium on Interactive 3D graphics and games - SI3D '06**, New York, New York, USA, p.175, 2006.

CARSTEN, J.; FERGUSON, D.; STENTZ, A. 3D Field D: improved path planning and replanning in three dimensions. **2006 IEEE/RSJ International Conference on Intelligent Robots and Systems**, [S.l.], p.3381–3386, Oct. 2006.

CHEW, L. P. Constrained Delaunay triangulations. **Proceedings of the third annual symposium on Computational geometry - SCG '87**, New York, New York, USA, p.215–222, 1987.

CONNOLLY, C.; BURNS, J.; WEISS, R. Path planning using Laplace's equation. **Proceedings., IEEE International Conference on Robotics and Automation**, [S.l.], p.2102–2106, 1990.

CORMEN, T. H. et al. **Introduction to Algorithms**. 2.ed. [S.l.]: McGraw-Hill Higher Education, 2001. 1202p.

COSTA, C. J. Example of a complete minimal immersion in IR3 of genus one and three-embedded ends. **Bulletin of the Brazilian Mathematical Society**, [S.l.], v.15, n.1-2, p.47–54, Mar. 1984. 10.1007/BF02584707.

CRYTEK. **CryENGINE 3**. Available at <http://www.crytek.com/cryengine>. Access in March 29, 2011.

DAPPER, F.; NEDEL, L. P.; JUNIOR, E. P. e. S. **Planejamento de movimento para pedestres utilizando campos potenciais**. 2007. Dissertação — Universidade Federal do Rio Grande do Sul.

DELOURA, M. **Game Programming Gems**. [S.l.]: Charles River Media, 2000.

DIJKSTRA, E. W. A note on two problems in connexion with graphs. **Numerische Mathematik**, [S.l.], v.1, n.1, p.269–271, Dec. 1959.

EPICGAMES. **Unreal Engine**. Available at <http://www.unrealengine.com/>. Access in March 29, 2011.

FISCHER, L. et al. Finding Hidden Objects in Large 3D Environments: the supermarket problem. **2011 XIII Symposium on Virtual Reality**, [S.l.], p.79–88, May 2011.

FISCHER, L. G.; SILVEIRA, R.; NEDEL, L. GPU Accelerated Path-Planning for Multi-agents in Virtual Environments. **2009 VIII Brazilian Symposium on Games and Digital Entertainment**, [S.l.], p.101–110, 2009.

FISCHER, L.; NEDEL, L. Semi-automatic navigation on 3D triangle meshes using BVP based path-planning. **XXIV SIBGRAPI - Conference on Graphics, Patterns and Images**, Los Alamitos, 2011.

GUIBAS, L. J.; STOLFI, J. **Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams**. New York, New York, USA: ACM Press, 1983. 221–234p.

HART, P.; NILSSON, N.; RAPHAEL, B. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. **IEEE Transactions on Systems Science and Cybernetics**, [S.l.], v.4, n.2, p.100–107, 1968.

HARTMAN, C.; BENES, B. Autonomous boids. **Computer Animation and Virtual Worlds**, [S.l.], v.17, n.3-4, p.199–206, July 2006.

HSU, D.; LATOMBE, J.-C.; MOTWANI, R. Path planning in expansive configuration spaces. **Proceedings of International Conference on Robotics and Automation**, [S.l.], v.3, p.2719–2726, 1997.

HUSSEIN, A.; ELNAGAR, A. Motion planning using Maxwell's equations. **IEEE/RSJ International Conference on Intelligent Robots and System**, [S.l.], p.2347–2352, 2002.

IñIGUEZ, P.; ROSELLY, J. Probabilistic Harmonic-function-based Method for Robot Motion Planning. **Proceedings of the 2003 IEEE/RSJ Intl. Conference on Intelligent Robots and Systems**, [S.l.], p.382–387, 2003.

KALLMANN, M. Path Planning in Triangulations. **Proceedings of the IJCAI Workshop on Reasoning, Representation, and Learning in Computer Games**, Edinburgh, Scotland, p.49–54, 2005.

KALLMANN, M. Shortest paths with arbitrary clearance from navigation meshes. **Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation**, Madrid, Spain, p.159—-168, 2010.

KANG, S.-J.; KIM, Y.; KIM, C.-H. Live path: adaptive agent navigation in the interactive virtual world. **The Visual Computer**, [S.l.], v.26, n.6, p.467–476, Apr. 2010.

KAVRAKI, L. et al. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. **IEEE Transactions on Robotics and Automation**, [S.l.], v.12, n.4, p.566–580, 1996.

KHATIB, O. Real-Time Obstacle Avoidance for Manipulators and Mobile Robots. **The International Journal of Robotics Research**, Thousand Oaks, CA, USA, v.5, n.1, p.90–98, Mar. 1986.

LAVALLE, S. M. **Rapidly-Exploring Random Trees: a new tool for path planning**. 1998.

LENGYEL, J. et al. Real-time robot motion planning using rasterizing computer graphics hardware. **ACM SIGGRAPH Computer Graphics**, [S.l.], v.24, n.4, p.327–335, Sept. 1990.

LINDEMANN, S. R.; LAVALLE, S. M. Current Issues in Sampling-Based Motion Planning. In: DARIO, P.; CHATILA, R. (Ed.). **Robotics Research**. [S.l.]: Springer Berlin / Heidelberg, 2005. p.36–54. (Springer Tracts in Advanced Robotics, v.15).

MOI3D. **Moment of Inspiration 3D**. Available at <http://moi3d.com/>. Access in September 14, 2011.

MULLER, D. Finding the intersection of two convex polyhedra. **Theoretical Computer Science**, [S.l.], v.7, n.2, p.217–236, 1978.

NIEUWENHUISEN, D.; KAMPHUIS, A.; OVERMARS, M. H. High quality navigation in computer games. **Science of Computer Programming**, [S.l.], v.67, n.1, p.91–104, June 2007.

NINTENDO. **Super Mario Galaxy**. Available at <http://pt.wikipedia.org/wiki/Super_Mario_Galaxy>. Access in March 30, 2011.

NVIDIA. **NVIDIA CUDA Programming Guide 2.0**. [S.l.]: NVidia, 2008. Available at <http://www.nvidia.com/cuda>. Access in October 5, 2011.

PARK, M. J. Guiding flows for controlling crowds. **The Visual Computer**, [S.l.], v.26, n.11, p.1383–1391, Jan. 2010.

PéBAY, P. P.; BAKER, T. J. A Comparison Of Triangle Quality Measures. **10th International Meshing Roundtable**, [S.l.], p.327–340, 2001.

PIEGL, L. A.; TILLER, W. **The NURBS Book (Monographs in Visual Communication)**. [S.l.]: Springer, 1996.

PRESTES, E. et al. Exploration technique using potential fields calculated from relaxation methods. **Proceedings 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems. Expanding the Societal Role of Robotics in the the Next Millennium**, [S.l.], v.4, p.2012–2017, 2001.

REYNOLDS, C. W. Flocks, herds and schools: a distributed behavioral model. **ACM SIGGRAPH Computer Graphics**, [S.l.], v.21, n.4, p.25–34, Aug. 1987.

ROSELL, J.; INIGUEZ, P. Path planning using Harmonic Functions and Probabilistic Cell Decomposition. **Proceedings of the 2005 IEEE International Conference on Robotics and Automation**, [S.l.], p.1803–1808, 2005.

SILVEIRA, R. et al. Natural steering behaviors for virtual pedestrians. **The Visual Computer**, [S.l.], v.26, n.9, p.1183–1199, Nov. 2009.

SILVEIRA, R. et al. Path-Planning for RTS Games Based on Potential Fields. In: BOULIC, R.; CHRYSANTHOU, Y.; KOMURA, T. (Ed.). **Motion in Games**. [S.l.]: Springer Berlin / Heidelberg, 2010. p.410–421. (Lecture Notes in Computer Science, v.6459).

SILVEIRA, R.; PRESTES, E.; NEDEL, L. P. Managing coherent groups. **Computer Animation and Virtual Worlds**, [S.l.], v.19, n.3-4, p.295–305, 2008.

SPRING. **Spring Engine**. Available at <http://springrts.com/>. Access in June 26, 2011.

STENTZ, A. The focussed D* algorithm for real-time replanning. **Proceedings of the 14th international joint conference on Artificial intelligence - Volume 2**, San Francisco, CA, USA, p.1652–1659, 1995.

TANENBAUM, A. S. **Computer Networks (4th Edition)**. [S.l.]: Prentice Hall, 2002.

TORCHELSEN, R. P. et al. Approximate on-Surface Distance Computation using Quasi-Developable Charts. **Computer Graphics Forum**, [S.l.], v.28, n.7, p.1781–1789, Oct. 2009.

TORCHELSEN, R. P. et al. Real-time multi-agent path planning on arbitrary surfaces. **I3D ’10: Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games**, Washington, D.C., p.47–54, 2010.

TREUILLE, A.; COOPER, S.; POPOVIć, Z. Continuum crowds. **ACM SIGGRAPH 2006 Papers**, Boston, Massachusetts, v.25, n.3, p.1160–1168, July 2006.

TREVISAN, M. et al. Exploratory Navigation Based on Dynamical Boundary Value Problems. **Journal of Intelligent and Robotic Systems**, [S.l.], v.45, n.2, p.101–114, May 2006.

UNITYTECHNOLOGIES. **Unity: game development tool**. Available at <http://unity3d.com/>. Access in March 29, 2011.

VALVE. **Source Engine**. Available at <http://developer.valvesoftware.com/wiki/Navigation_Meshes>. Access in June 28, 2011.

YAN, H. et al. Path planning based on Constrained Delaunay Triangulation. **Intelligent Control and Automation, 2008. WCICA 2008. 7th World Congress on**, [S.l.], p.5168–5173, 2008.

ZHAO, C.; FAROOQ, M.; BAYOUMI, M. Collision-free path planning for a robot with two arms cooperating in the 3-D work space. **Proceedings of IEEE International Conference on Robotics and Automation**, [S.l.], v.3, p.2835–2840, 1996.

APPENDIX A - PROOF OF EQUATION 4.13

In Section 4.2 we said that if the vectors n_0, n_1 and n_2 lie in the same plane defined by a triangle t and have the same lenghts of the associated edges of t, it is possible to write

$$n_1 + n_2 - n_0 = (0, 0, 0). \tag{8.1}$$

In order to prove this, we will define the vectors n_0, n_1 and n_2 assuming the triangle t, depicted in Figure 8.1. This triangle is defined by its vertices $v_0 = (x_0, y_0, z_0)$, $v_1 = (x_1, y_1, z_1)$ and $v_2 = (x_2, y_2, z_2)$. The normal of t is

$$\hat{n} = \frac{(v_1 - v_0) \times (v_2 - v_0)}{\|(v_1 - v_0) \times (v_2 - v_0)\|}. \tag{8.2}$$

It is possible to write the vectors n_0, n_1 and n_2 as

$$\begin{aligned} n_0 &= R(v_1 - v_0), \\ n_1 &= R(v_1 - v_2), \\ n_2 &= R(v_2 - v_0) \end{aligned} \tag{8.3}$$

where R is a rotation matrix around the vector $\hat{n}$ with angle $\frac{\pi}{2}$. Combining the equations 8.1 and 8.3 we can write:

$$n_1 + n_2 - n_0 = R(v_1 - v_2) + R(v_2 - v_0) - R(v_1 - v_0) = (0, 0, 0). \tag{8.4}$$

As R is a linear transformation, 8.4 can be written as

$$R\left[(v_1 - v_2) + (v_2 - v_0) - (v_1 - v_0)\right] = (0, 0, 0) \tag{8.5}$$

As the terms of Equation 8.5 cancels each other, the rotation matrix R is multiplied by the vector $(0, 0, 0)$, which results in the same vector $(0, 0, 0)$, completing the proof:

$$R(0, 0, 0) = (0, 0, 0). \tag{8.6}$$

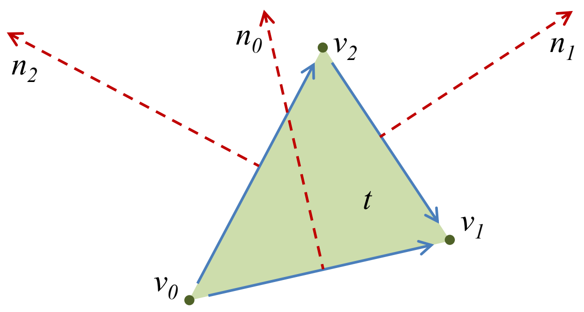


Figure 8.1: The triangle t, its vertices and associated *edge normals*, used in Equation 4.14.

APPENDIX B - TEST CASES

On the Section 5.1 we presented a study comparing the 3DS-BVP and the BVP Path Planner. In Figure 5.1, we presented the test cases 4 and 8. Test case 4 presented an environment and paths computed using the BVP Path Planner and the 3DS-BVP. The test case 8 presented the same environment as the test case 4, but using a noisy triangle mesh to compute the paths on the 3DS-BVP. This section presents all the other test cases used during this test (except the test cases 4 and 8, which are already presented in Section 5.1).

As in Figure 5.1, figures 8.2, 8.3, 8.4, 8.5, 8.6 and 8.7 use a gradual color transition from green to red to represent the value of the potential in each point of the environment, from 0 to 1, respectively. In all figures, the white lines represent the path generated by the BVP Path Planner, and the blue lines represent the path generated by the 3DS-BVP. Yellow lines represent separation between *occupied*, *goal* and *free* cells.

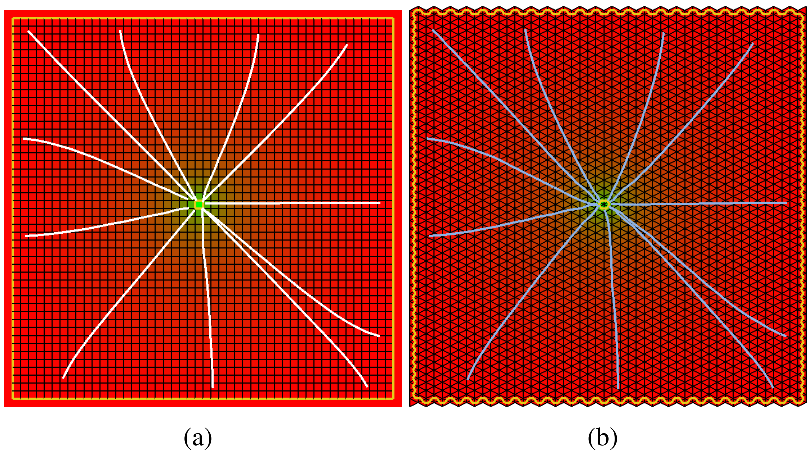


Figure 8.2: Comparison between the BVP Path Planner and the 3DS-BVP, test case 1. (a) BVP Path Planner. (b) 3DS-BVP.

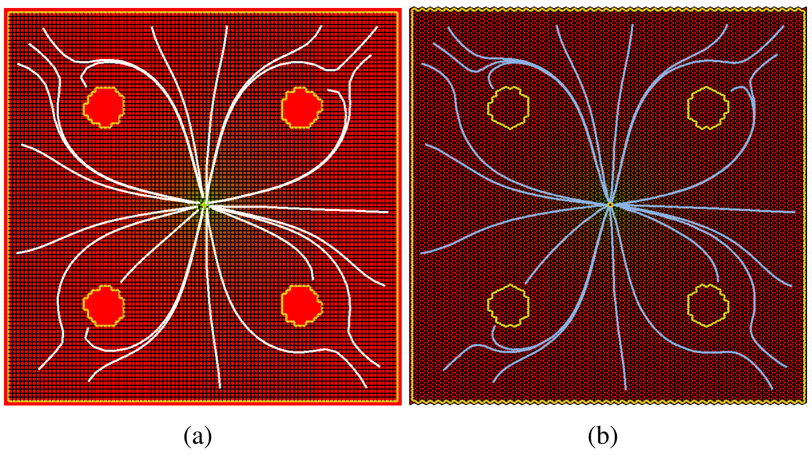


Figure 8.3: Comparison between the BVP Path Planner and the 3DS-BVP, test case 2. (a) BVP Path Planner. (b) 3DS-BVP.

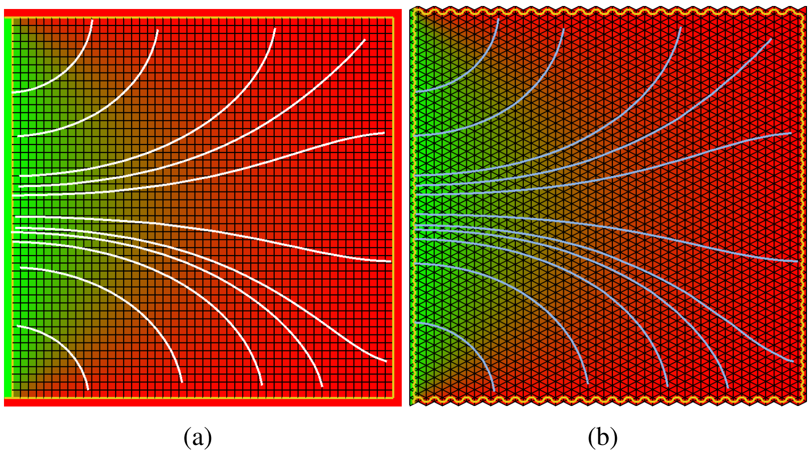


Figure 8.4: Comparison between the BVP Path Planner and the 3DS-BVP, test case 3. (a) BVP Path Planner. (b) 3DS-BVP.

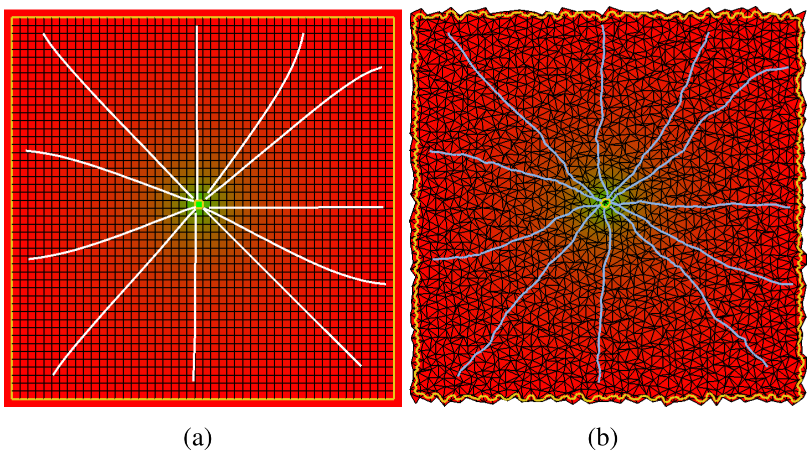


Figure 8.5: Comparison between the BVP Path Planner and the 3DS-BVP, test case 5. (a) BVP Path Planner. (b) 3DS-BVP, with noisy mesh.

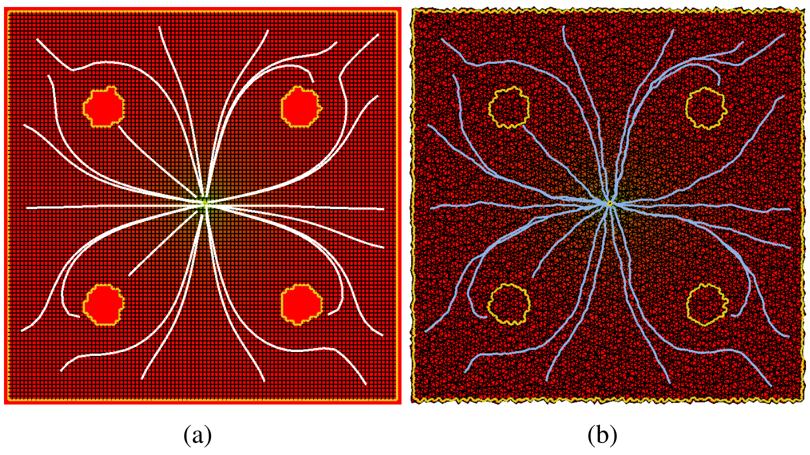


Figure 8.6: Comparison between the BVP Path Planner and the 3DS-BVP, test case 6. (a) BVP Path Planner. (b) 3DS-BVP, with noisy mesh.

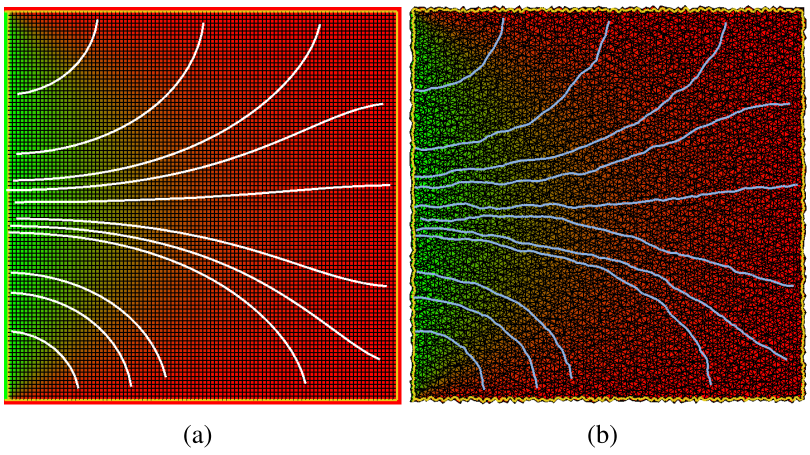


Figure 8.7: Comparison between the BVP Path Planner and the 3DS-BVP, test case 7. (a) BVP Path Planner. (b) 3DS-BVP, with noisy mesh.

APPENDIX C - ARTICLES PUBLISHED DURING THIS WORK

During the development of this work, several articles were published. The following pages contains the full text of these articles.

Semi-Automatic Navigation on 3D Triangle Meshes Using BVP Based Path-Planning

Leonardo Fischer, Luciana Nedel
Institute of Informatics
Federal University of Rio Grande do Sul – UFRGS
Porto Alegre, Brazil
{lgfischer, nedel}@inf.ufrgs.br

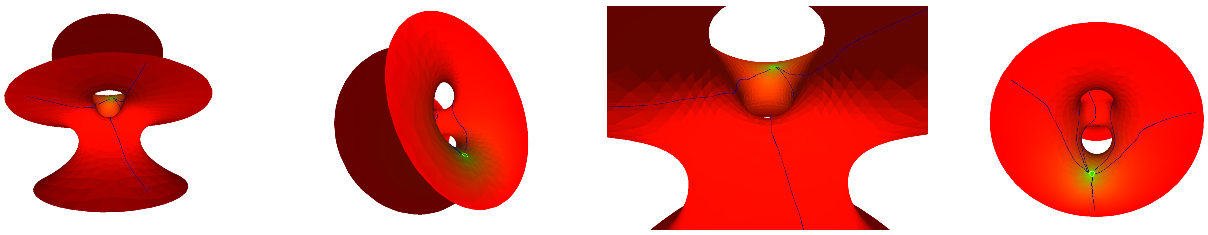


Fig. 1. Paths produced on the Costa Minimal Surface [1]. Note how lines starting in different points of the surface smoothly reach the target point (green).

***Abstract*—Efficient path-planning methods are being explored along the years to allow the movement of robots or virtual agents in planar environments. However, there is a lot of space to improve the quality of paths restricted to 3D surfaces, with holes and bends for instance. This work presents a new technique for path-planning on 3D surfaces called 3DS-BVP. This path planner is based on Boundary Value Problem (BVP), which generates potential fields whose gradient descent represents navigation routes from any point on the surface to a goal position. Resulting paths are smooth and free from local minima. The 3DS-BVP works on complex surfaces of arbitrary genus or curvature, represented by a triangle mesh, without the need of 2D parametrizations. Our results demonstrate that our technique can generate paths in arbitrary surfaces with similar quality as those generated by BVP-based methods in planar environments. Our approach can be applied in the development of new tools to automate the navigation on 3D surfaces, like the camera control in the exploratory visualization of 3D models.**

***Keywords*-3D path-planning; navigation; surfaces exploration.**

I. Introduction

Navigation on three-dimensional surfaces is a relevant problem for many application areas, such as: scientific visualization, where a user needs to inspect different objects, as organs in a medical application or engines in a CAD system; robotics, with the automatic definition of paths and motions for robots; and entertainment, more specifically on the video games domain, where the exploration of complex 3D worlds are much more challenging for the player than planar ones.

Navigation is a complex interactive task and is usually divided in two parts [2]: travel, and wayfinding. While the travel is the motor component of navigation, the low-level actions that a user makes to control the position and orientation of his/her viewpoint, the wayfinding is the cognitive component, and includes high-level thinking, planning, and decision making. It includes spatial understanding and planning tasks, such as determining a path from the current location to a goal location.

Path-planning algorithms are being explored for years. Several solutions were applied into robotics and virtual environments, with some of them focusing on a high performance path finding – normally including a pre-processing phase [3] –, and others on providing better paths. Although most of these solutions focus on the problem on the Euclidean plane, some of them are robust enough to handle systems with more than two degrees of freedom (as 3D path-planning [4] or path-planning for robotic arms with several joints [5]). However, path-planning methods restricted to arbitrary surfaces is not well explored in the literature.

Methods that focus specifically on 2D path-planning cannot be trivially modified to handle arbitrary surfaces. One possible approach is to use a sophisticated projection technique from the 3D surface to the Euclidean plane, and then modify the 2D path-planning algorithm to work on this projection, which is not a trivial task (due to the nature of this projection). Algorithms that handle 3D environments depend on their nature to be adapted to 3D surfaces, as in a given point of the surface the algorithm should behave as a 2D path planner. Graph based approaches are fast enough for real-time applications, but the generated paths are not as smooth as other approaches. In all these cases, the required work for porting is not negligible, and it is not clear how these algorithms will behave in this kind of environments.

In this work we developed a solution for the second part of the navigation problem on arbitrary surfaces, the wayfinding.

We present a new path-planning technique that handles the arbitrary 3D surface case, so called *3DS-BVP*, an acronym to *3D Surface Path Planner using Boundary Value Problems system*. The technique uses boundary value problem (BVP) systems to generate potential fields using a triangle mesh discretization. Using the gradient of the potential field, the agent can be guided through the environment. Briefly, the main contributions of this paper are:

- A numerical method that generates potential fields using a triangle mesh discretization;
- A path planner based on potential fields that draws smooth paths on 3D triangular surface meshes.

Our technique is based on a path-planning algorithm that is able to generate smooth paths with low probability of collision with obstacles, using potential fields.

The remainder of this papers is organized as follows. Section II presents the related work on path-planning for interactive applications and robotics. Sections III and IV describe the technique, while Section VI presents the results achieved and some discussion about it. Finally, in Section VII our conclusions and future works are discussed.

II. Related Work

Path-planning algorithms are being used to find a path to be followed from a given position to a goal one on a virtual environment. Many algorithms have been proposed to solve this problem, and the most part of them assume that the free space can be projected on a 2D surface.

Kallmann [6] used constrained Delaunay triangulations to discretize the free space of the environment in a triangle mesh, and a graph approach to search for free paths. Afterwards, he has also proposed a method that search for paths in an environment with specific clearance [7]. Despite the fact that these methods use triangle meshes as main data structure, they are developed only for planar environments. Our method do not make any difference between planar and 3D surfaces.

Techniques based on potential fields for navigation include the work of Rosell and Iniguez [8], Trevisan et al. [9], Treuille et al. [10], and Park [11]. These techniques use positions of obstacles and agents to compute a function. The result is a field from where the directions to a target position are derived. These techniques differ from each other in the function that is used to compute the field and how directions are derived, resulting in different behaviors for each technique. For example, the work of Trevisan et al. favors the exploratory behavior of an agent, while the work of Treuille et al. favors its use with crowds of autonomous agents. All these techniques were developed for 2D environments. Some of them can also be applied for 3D environments by adding one dimension to their equations, but this will significantly degrade its performance.

A path planner based on geodesic distances on triangular 3D meshes was recently proposed by Torchelsen et al. [12]. This work focuses on multi-agent systems, an uses a CPU/GPU architecture to handle the collision avoidance between the agents. The main advantage of this method is the high performance achieved. On the other side, the paths generated are close to the shortest ones, which can lead to a high probability of collision paths with static obstacles. Our potential field approach produces smooth paths that whenever is possible avoids getting very close to the obstacles.

Due to its performance and low memory requirements, graph-based approaches are the most common in the game industry. Popular game engines as Unreal Engine® and CryEngine® made use of it. In these methods, a graph represents the environment and the Dijkstra algorithm [13] (or one of its derivations) is used to find a path between two nodes. The difference between the approaches (as the ones proposed by Kavraki et al. [14], Barraquand et al. [15], Lavalle [16], and Kang et al. [17]) is the algorithm used to sample the graph from the environment and how it is updated. All these methods seem to be easily adaptable for 2D and 3D path-planning, but they are not being explored for arbitrary surface path-planning.

III. BVP Path Planner

The BVP Path Planner [9] is a 2D Path Planner that generates paths using the potential information computed from the numeric solution of

$$\nabla^2 p(\mathbf{r}) = \epsilon \mathbf{v} \cdot \nabla p(\mathbf{r}) , \qquad (1)$$

with Dirichlet boundary conditions, where $\mathbf{v} \in \Re^2$ and $|\mathbf{v}| = \mathbf{1}$ corresponds to a vector that inserts a perturbation in the potential field; $\epsilon \in \Re$ corresponds to the intensity of the perturbation produced by $\mathbf{v}$; and $p(\mathbf{r})$ is the potential at position $\mathbf{r} \in \Re^2$, respectively. Both $\mathbf{v}$ and ϵ must be defined before computing this equation. The gradient descent on these potentials represents navigational routes from any point of the environment to the goal position. Trevisan et al. [9] shows that this equation does not produce local minima and generates smooth paths.

To solve numerically a BVP, we consider that the solution space is discretized in a regular grid ([9], [18]). Each cell (i, j) is associated to a squared region of the environment and stores a potential value $p(i, j)$. Using the Dirichlet boundary conditions, the cells associated to obstacles in the environment store a potential value of 1 (*high potential*) whereas cells containing the goal position store a potential value of 0 (*low potential*).

A high potential value prevents the agent from running into obstacles whereas a low value generates an attraction basin that pulls the agent. The relaxation methods employed to compute the potentials of free space cells is the Gauss-Seidel (GS). The GS method updates the potential of a cell c through:

$$p_c = \frac{p_b + p_t + p_r + p_l}{4} + \frac{\epsilon((p_r - p_l)v_x + (p_b - p_t)v_y)}{8} \qquad (2)$$

where $\mathbf{v} = (v_x, v_y)$, and p_c, p_b, p_t, p_r and p_l are cells of a grid, as illustrated in Figure 2.

The GS method allows the use of partial results as an approximation of the potential field [19]. Since the exact solution is not necessary, we can control the accumulated error $e(t)$ at each iteration through a tolerance threshold e_{max}

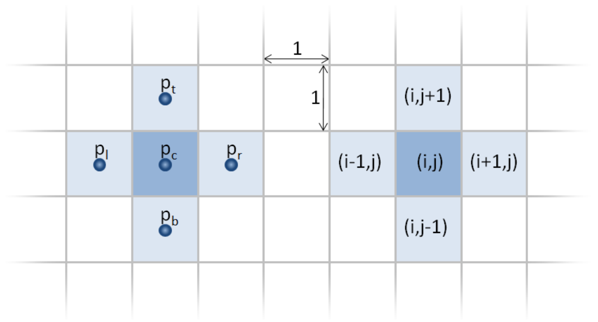


Fig. 2. Cells of a grid, when Equation 2 is evaluated for the cell p_c.

according to:

$$e(t) = \sum_{i=1}^{m} \sum_{j=1}^{n} |p(i,j)^t - p(i,j)^{t-1}| \geq e_{max}. \qquad (3)$$

where $p(i,j)^t$ is the potential of the cell (i,j) at the iteration t, $p(i,j)^{t-1}$ is the potential of the same cell in the previous iteration and m and n are the grid dimensions.

After the potential computation, the agent moves following the direction of the gradient descent of this potential at its current position (i,j).

IV. Path-Planning on Triangle Meshes

In our proposal, the 3DS-BVP generates potential fields that produce smooth paths in triangle meshes. The method works accordingly to the steps: (1) discretize the environment in a set of cells; (2) calculate the potential field; (3) compute the gradient of the potential field.

A. Discretizing the environment

Triangle meshes are widely used in computer graphics industry to model objects and virtual environments, what means that our technique can be easily applied to existing 3D models and surfaces. Then, we assume that the environment is represented by a triangle mesh that is used as a triangular grid to compute the potential field.

The triangular grid is represented by a *Doubly-Connected Edge List* (*DCEL*) data structure [20] due to its simplicity, capability to find the neighbors of a vertex, face or edge in constant time, and ability to handle closed and open triangle meshes. For the scope of this work, an *open mesh* is a mesh of triangles that is topologically equivalent to a 2D plane. Meshes with *holes* in its surface are also considered *open meshes*. A *closed mesh* is *not* topologically equivalent to a plane, in such a way that it should be *broken* into two or more open meshes to build a mapping function from the mesh to a plane. *Closed meshes* also don't have holes in its surface.

As shown in Figure 3, an edge in the DCEL connects two vertices and is actually defined by a pair of half-edges, each one being a *twin* of the other. A face is then defined by a sequence of half-edges starting at the *boundary* half-edge of the face, usually in a counter-clockwise order. The half-edges

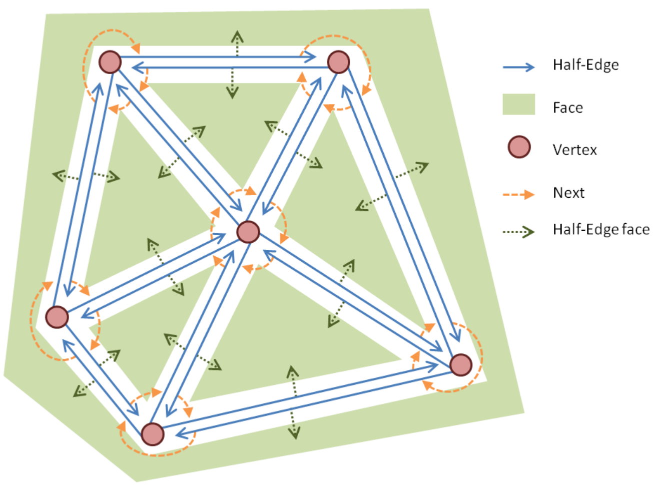


Fig. 3. DCEL data structure. Edges are represented by pairs of parallel blue arrows, vertices by red circles, and faces by green polygons.

are connected among themselves through the *next* and *previous* pointers.

The first step of the 3DS-BVP algorithm works as follows. To each vertex of the *DCEL* data structure, a potential value is assigned – in the same way of the BVP path planner cell in Section III. Then, each triangle of the grid is divided into three regions by connecting the medians of the edges of the triangle to its centroid (see Figure 4). Each triangle region is associated to the near vertex and, for a given vertex v of a triangle t, the function $Region(v,t)$ returns that associated region. Also, the algorithm assumes that each vertex v in the triangle mesh is associated with a set of triangles $Triangles(v)$, where each triangle in $Triangles(v)$ has one vertex equals to v (i.e. they share the same vertex). The cell associated to a given vertex v is then defined by function $Cell(v) = \{Region(v,t)|\forall t \in Triangles(v)\}$. The function $Vertex(c)$ returns the vertex associated to the cell c. Figure 5 illustrates these concepts.

We also define the concept of *neighbor cell* as two cells that have their associated vertices connected through a single pair of half-edges. This means that two cells c_1 and c_2 are neighbors if the function $Link(Vertex(c_1), Vertex(c_2))! = null$ satisfies. Assuming that the set C contains all the cells derived from the triangle mesh, the function $Neighbors(c_i) = \{c_j|\forall c_j \in C, i! = j, Link(c_i,c_j)! = null\}$ returns the set of neighbor cells for a given cell c_i.

B. Calculating the potential field

In order to execute the relaxation over the set of cells, the boundary conditions must be defined. These boundary conditions are set according to the positions of obstacles and the goal in the environment. We assume that obstacles and goal positions are constrained to the surface. As in the BVP Path Planner, each cell receives a tag and an initial potential value, as follows:

- cells associated with occupied areas of the environment and cells associated with *limiting vertices* (vertices in the

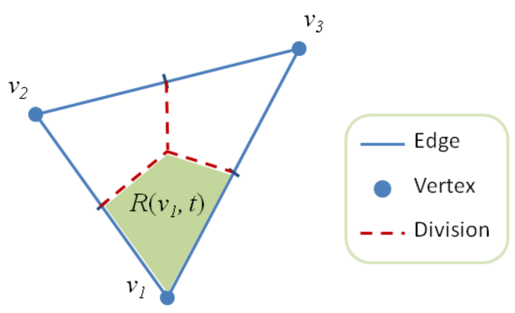


Fig. 4. Triangles are divided in order to build the cell mesh. In highlight, the area returned by the function $Region(v_1, t)$.

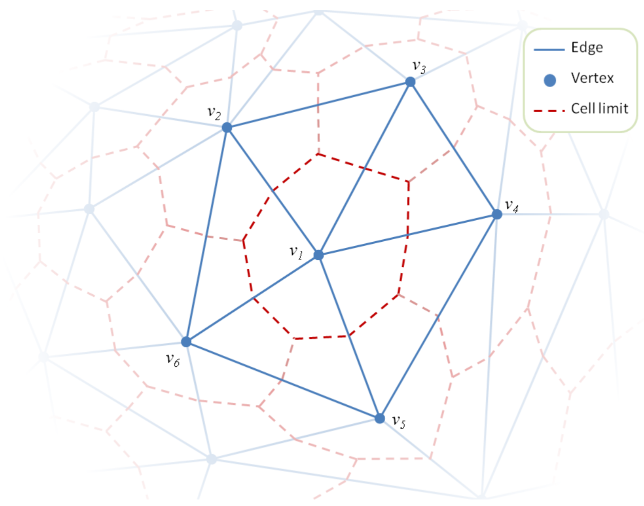


Fig. 5. A triangle mesh and its corresponding cell division. The five highlighted triangles ($\{v_1, v_2, v_3\}$, $\{v_1, v_3, v_4\}$, $\{v_1, v_4, v_5\}$, $\{v_1, v_5, v_6\}$ and $\{v_1, v_2, v_6\}$) share the same vertex v_1 and, therefore, they are in the set $Triangles(v_1)$. Applying the function $Region(v_1, t)$ for each triangle t in $Triangles(v_1)$ results in the cell associated to the vertex v_1, which is highlighted with the dotted line around v_1.

border of the mesh or in the limit of a hole in the mesh) are tagged as *occupied* and receive the *high potential* value;

- the cell associated with the goal position receives the *goal* tag and the *low potential* value;
- cells associated with *free* navigable areas of the environment are tagged as *free* and receive a mean value between the *low* and *high potential* values, as in the BVP Path Planner.

In order to update the potential value of the *free* cells, a set of functions need to be defined. Assuming that the function $Dist(v_1, v_2)$ returns the Euclidean distance between the vertices v_1 and v_2, $D_{min}(c_i)$ and $D_{max}(c_i)$ return the minimum and maximum distance, respectively, between the vertex of a cell c_i and the vertices in its neighborhood.

The influence of a cell c_j over the cell c_i, relative to the neighborhood of c_i is given by

$$\begin{aligned} I(c_i, c_j) &= D_{min}(c_i) + D_{max}(c_i) \\ &\quad - Dist(Vertex(c_i), Vertex(c_j)) \end{aligned} \qquad (4)$$

Equation 4 is an heuristical measure of how much the potential of one cell c_j has over its neighbor c_i. As $D_{min}(c_i)$ and $D_{max}(c_i)$ are computed from the same input set, it is guaranteed that $D_{min}(c_i) <= D_{max}(c_i)$. The closer the cell c_j is to c_i, closer the value of $I(c_i, c_j)$ is to D_{max}. The farther a cell c_j is from the cell c_i, closer the value of $I(c_i, c_j)$ is to D_{min}. Then, the result of $I(c_i, c_j)$ can be interpreted as how close a cell c_j is to c_i, in a scale between $D_{min}(c_i)$ and $D_{max}(c_i)$.

Based on Equation 4, the function

$$I_{total}(c_i) = \sum_{j=1}^{\#Neighbors(c_i)} I(c_i, c_j) \qquad (5)$$

computes the sum of the influences that a cell receives from its neighbors. This function is then used in the equation

$$p(c_i) = \sum_{j=1}^{\#Neighbors(c_i)} p(c_j) \frac{I(c_i, c_j)}{I_{total}(c_i)}, \qquad (6)$$

which is used to update the potential $p(c_i)$ of a *free* cell c_i.

The potential values of the *free* cells are updated using Equation 6 until the convergence sets in, as in the BVP Path Planner. A threshold error e_{max} is used to verify if the potential field has converged. Equation 3 is used over the whole set of cells C to compute the error at a given iteration.

C. Computing the gradient of the potential field

Calculating the potential field, an agent should be able to follow the gradient descent in order to reach the goal position. Equation 6 is able to mimics the results produced by the Laplace's Equation. We can then calculate the gradient of Equation 6, as we use to do with the Laplace's Equation.

One possible solution for the calculus of the gradient of the Laplace's Equation in an unstructured triangular mesh involves the use of the integral form of that equation, which can be obtained by integrating the equation over some volume Ω, and then applying the Gauss Divergence Theorem. The integral form of the Laplace's Equation and its equivalent after applying the Gauss Divergence Theorem is

$$\int_{\Omega} \nabla^2 p(r) d\Omega = \oint_{\partial\Omega} \nabla p(r) \cdot \hat{n} dA = 0. \qquad (7)$$

The relation between the gradient of the function $p(r)$ over a volume and the integral on the surface of the volume of the function times the normal area vector is

$$\int_{\Omega} \nabla p(r) d\Omega = \oint_{\partial\Omega} p(r) \hat{n} dA. \qquad (8)$$

The relation above, when computed in sufficiently small volumes can be used to compute an approximation of the gradient. This results in

$$\nabla p(r) \approx \frac{1}{\Omega} \oint_{\partial\Omega} p(r)\hat{n} dA. \tag{9}$$

Assume that we want to compute the approximation of the gradient in the triangle $t = \{v_0, v_1, v_2\}$. For this, we calculate the normal vector of each edge of each triangle and multiply each one by the length of the respective edge. The area of the triangle t is calculated by the function $Area(t)$. Function $p(r)$ is already computed in each vertex.

The approximation of Equation 9 on the triangle t yelds the equation

$$\nabla p(t) \approx \frac{1}{2Area(t)} \left[-p(v_0)\hat{n}_1 - p(v_1)\hat{n}_2 + p(v_2)\hat{n}_0 \right]. \tag{10}$$

that is then used to approximate the gradient of the potential field at the centroid of the triangle t.

To move towards the target position in the environment, an agent a must follow the gradient descent of the triangle where he is on. This triangle is kept as a pointer to the current face in the DCEL strucutre. When the agent walks out of the current face, the functions provided by the DCEL data structure can be used to check which one of the edges was crossed. The crossed edge will be represented by a half-edge h, and the function $Face(Twin(h))$ will be set as the current face of the agent. The function $UpdatePosition(a, g)$ is used to update the agent position and the current face based on the gradient g.

V. Potential Fields in 3D Surfaces

In a completely planar environment discretization, the path planner presented in Section IV behaves in a similar way to the BVP Path Planner (see Section III). The goal position must be checked with a *goal* tag while the obstacles are tagged as *occupied*. The limits of the environment are also tagged as *occupied* and all the remaining cells are tagged as *free*. All cells receive some potential value, according to what was specified in the Sub-section IV-B. Then, the relaxation step evaluates adequate values for *free* cells. And finally, the agent can use the gradient of the potential field to build a path to follow.

In this kind of environment, the path planner can take advantage of a simpler math, since all the vertex positions and normals can be manipulated in 2D. Naturally, a 3D discretization will require a third coordinate for the position of vertices and normals.

For 3D open meshes the path planner works with minimal modification, and the addition of a coordinate axis does not introduce significant changes in any of the methods presented in the previous section. During the relaxation, only the function $Dist(v_1, v_2)$ changes and must compute the Euclidean distance in 3D. In addition, the normals used to compute the gradient must also be stored in 3D. But the normals keep parallel to the plane defined by the triangle which they belong.

The most relevant modification in the algorithm to deal with 3D meshes is the handling of boundary conditions. Our algorithm requires that at least two different boundary conditions must be set: one *goal* cell and one *occupied* cell. We assume that the *goal* is always defined, so for 3D meshes we should take a special care with the *occupied* cells.

If the 3D mesh is *open* (as defined in Section IV-A), the *limiting vertices* will force the existence of *occupied* cells, as explained in Section IV-B. *Closed meshes* with obstacles on its surface will also force the existence of *occupied* cells. This will generate a gradient that prevents the agent from leaving the surface limits, collides with obstacles, and guides to the goal position.

However, if the mesh is *closed* and do not have any obstacle on its surface, then our algorithm will generate only *goal* cells. If this occurs, the relaxation step will stop only when all cells have their potential value equal to *low potential*. This will lead to a null gradient on the mesh (the result of Equation 10 will be $(0, 0)$), and the agent will have no clue about which direction it should follow.

To avoid the occurrence of null gradients, the cell containing the initial position of the agent is tagged as *occupied* and receives the *high potential* value. Although the cell does not contain any obstacle, this will force the potential field to have a gradient capable of guiding the agent from its initial position to the goal. Also, as the agent is leaving that position away, the obstacle added will not modify the existence of a path from the initial to the goal position.

Meanwhile, this approach has a drawback. The path taken by the agent may change significantly according to the initial position of the agent inside the modified cell. Because the cell that corresponds to the initial position is modified to receive the *occupied* tag and *high potential* value, the gradient on the cell borders will be significantly different according to the triangle where the position is. The fact that the mesh is closed and do not have other boundary conditions allows the situation where many different paths can be taken to reach the same goal position.

VI. Results

In order to evaluate our work, we produced a set of tests. First, we present a comparison of our method in a planar triangular mesh, showing that our technique is able to produce smooth paths with low collision probability, in the same way that the BVP Path Planner does. Then, we present some of the results that we obtained with 3D arbitrary surfaces.

A. Comparing our method with the BVP Path Planner

We designed a set of test cases, where each test case is composed of a squared environment with some obstacles and goals in it. In the test cases 1 up to 4 the environment were discretized in a regular triangle grid. The test cases 5 up to 8 used the same environments as the first cases, but with addiction of noise to disturb each vertex position in the triangle

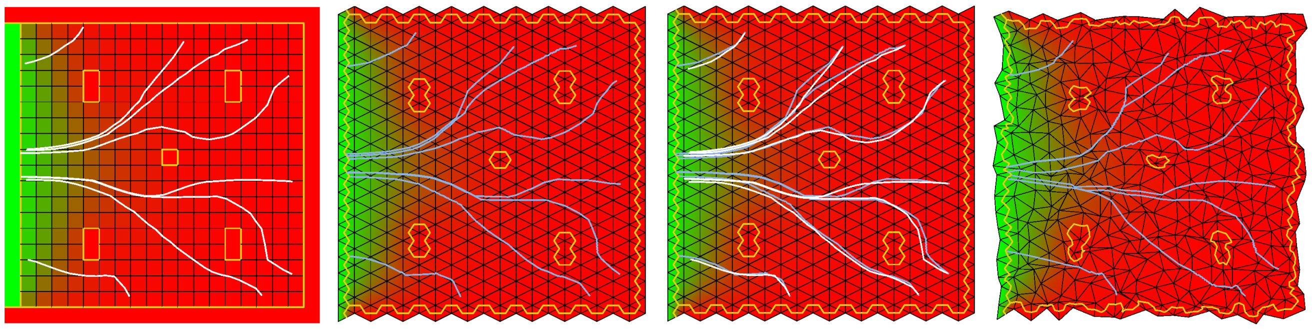


Fig. 6. Planar environment. The gradual color transition from green to red represents the value of the potential in each point of the environment, from 0 to 1, respectively. (a) The environment discretized in a regular grid, with the paths computed with the BVP Path Planner; (b) the environment discretized in a regular triangle mesh, with paths computed with the 3DS-BVP; (c) comparison between the paths generated with the BVP Path Planner and the 3DS-BVP; (d) a discretization of the environment using a triangle mesh with noise.

Test Case	Average Error	Std. Deviation	Max. Error	#Cells
1	0.00394	0.00606	0.06224	2,500
2	0.00303	0.00417	0.05802	10,000
3	0.01844	0.01283	0.08957	10,000
4	0.00604	0.00817	0.08911	10,000
5	0.00903	0.01042	0.15687	2,500
6	0.00469	0.00631	0.30755	10,000
7	0.01867	0.01717	0.49989	10,000
8	0.00792	0.01332	0.49913	10,000
Average	0.00897	0.00981	0.22030	

TABLE I
COMPARISON BETWEEN THE POTENTIAL FIELD GENERATED WITH OUR TECHNIQUE AND THE BVP PATH PLANNER. AS THE POTENTIAL FIELDS GENERATED WAS NORMALIZED, THE AVERAGE ERROR IN ALL CASES WAS ABOUT 0.8%, WITH A STANDARD DEVIATION OF ABOUT 0.9% OF THE RESULT PRODUCED BY THE BVP PATH PLANNER.

grid (better representing the triangles found in 3D surfaces). Figure 6 illustrates these test cases. The green region on the border represents the goal and the yellow lines delimit the obstacles. The white lines are the path generated by the BVP Path Planner, starting at random positions on the environment, while the blue lines are the path generated by our technique. Figure 6(a) illustrates our test cases using the regular grid, while Figure 6(b) and (d) used a triangle grid. Figure 6(c) combines the paths produced by the BVP Path Planner and the 3DS-BVP in a single image.

We can see that the paths produced by the 3DS-BVP in Figure 6 are quite similar to the paths produced by the BVP Path Planner. By adding noise, these paths loses some smoothness due to the presence of low quality triangles, but still mimics the results produced by the BVP Path Planner and by 3DS-BVP on regular triangle grid.

We have also compared the potential value in each cell of the potential field produced by the 3DS-BVP with the potential field generated by the BVP Path Planner. Table I presents a summary of the differences found between these values. We can see that, in average, the potential field generated by our method is almost the same potential field produced by the BVP Path Planner, with a difference of only 0.8% on average.

The highest difference was 49.989%, found in the test case 7. Although this is a considerable difference between the 3DS-BVP and the BVP Path Planner, it occurred in an environment that had an average difference of 1.867% and standard deviation of 1.717%. This difference occurred due to the existence of highly deformed triangles resulting from the noise. The test case 8 produced very similar potential values, and generated the paths illustrated in Figure 6(d).

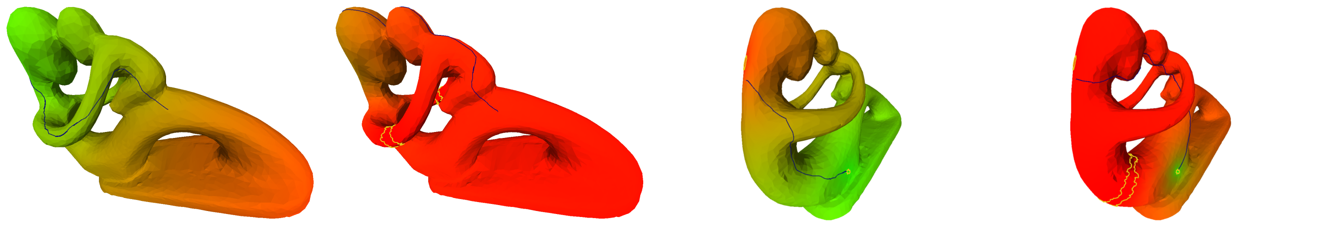
B. *Path-planning evaluation in arbitrary meshes*

We applied our algorithm on some 3D models to analyze how paths are generated on these surfaces. In Figure 1 we applied our algorithm on the Costa Minimal Surface [1], a complete minimal embedded surface with a genus with three punctures. We generated several paths on this surface, from several distinct initial positions to a predefined goal position. In all cases within this surface, the algorithm found a smooth path to reach the goal position.

In another experiment, we used a model of the *Fertility* statue (Figure 7) to generate paths on its surface. The *Fertility* has several genus, which also makes it a good example of the kind of environment that our technique deals with. In Figures 7(a) and (c), our planner has generated quality and smooth paths to reach the goal position. In Figures 7(b) and (d) we used the same initial and goal positions as (a) and (c), respectively, but we also defined some regions where the path could not crossover, simulating obstacles on the surface. The algorithm demonstrated to be able to find quality and smooth paths.

C. *Performance evaluation*

We measured the performance of our algorithm in several cases, including the surfaces presented in Section VI-B. We measured the time spent to compute the potential field, and the number of iterations needed to the convergence with a threshold error $e_{max} = 0.001$. Results are presented in Table II. The tests were executed in a Intel® Core i7 870, 2.93GHz, 4GB Ram and NVidia ® GeForce GTX 470.



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

Fig. 7. Several paths produced on a complex model (*Fertility* mesh), with several genus. Note that in (b) the initial and goal positions are the same as (a), but some obstacles resulted in a different path. The same occurs in (c) and (d).

Model	Faces	Vertices/Cells	Time (s)	Iterations
Car	1,292	665	0.2953	1,006
Costa	2,320	1,259	0.3948	854
Statue	10,000	4,994	41.2660	19,434

TABLE II
PERFORMANCE EVALUATION ON THREE TEST CASES.

Although the quality and smoothness of paths generated in the previous tests, our performance evaluation shows that there is still room for improvement. For smaller but complex models, like the Costa Minimal Surface, our algorithm is able to solve the potential field and produce a quality path for applications, like the motion of cameras in virtual environments. For more detailed models, like the Fertility, our algorithm still needs some improvements in its performance.

D. Limitations and degenerated cases

Analysing Equation 4 one can conclude that if, eventually, two vertices are in the same position, $D_{min}(c_i)$ will be equal to 0, and any vertex with a distance of $D_{max}(c_i)$ will have no influence over the vertex c_i. This may result in an interruption of the propagation of the potential from one region to another. Also, it is clear that if the mesh has an invalid triangle (with collinear vertices, for instance), the result of the Equation 10 is undefined.

We tested our method with many different meshes. In an experiment with several very long non-equilateral triangles, the paths lost its smoothness. This happens because the gradient of the adjacent triangles presented big differences between them.

VII. Conclusions and Future Work

We presented the 3DS-BVP, a technique based on potential fields for path-planning in arbitrary surfaces. As the main advantage of our technique, it generates smooth paths free from local minima on 3D surfaces, without the need of a 2D parametrization, or some other surface representation.

The potential field produced by the 3DS-BVP generates all the possible paths from a point in the environment to a goal position following the descent gradient. Paths generated on 3D surfaces shows to be quite similar to the quality of BVP Path Planner, with a potential field with a difference of only 0.8% to it on average. This is a good feature, because the 3DS-BVP uses as core a potential field with similar characteristics of the ones produced by the BVP Path Planner.

As its main drawback, its performance is not sufficiently good for real-time applications, as environments with several moving agents. We believe that this drawback can be minimized by improving our solution method and by developing GPU based methods to compute the potential field.

Some possibilities to improve the algorithm performance are being analyzed. One possible way is to improve the set of equations that we used, in order to obtain the potential field using less iterations. Also, the order that cells are evaluated reflects in the speed that the potential values from the obstacles and goals are propagated to the free cells. So, there should be an ideal order that makes the relaxation process faster. Another possible performance optimization is to implement our algorithm using GPU. Our method appears to been highly parallelizable, as the one shown in a previous work [21].

We intend to apply this algorithm mainly in virtual environments. New tools to control the virtual camera in CAD and modeling applications can be developed, in order to help the evaluation and visualization of 3D models. In video games like Prey® and Super Mario Galaxy® the player and its enemies walk on arbitrary surfaces to reach their objectives. Future video games using this kind of environment can also have benefits from our algorithm.

Acknowledgment

The authors would like to thank Renato Silveira for his ideas and help with the text, as well as CNPq-Brazil through projects 483947/2010-5, 580156/2008-7, 309092/2008-6 and Microsoft Brazil Interop Labs. for partially supporting this work.

References

[1] C. J. Costa, "Example of a complete minimal immersion in IR3 of genus one and three-embedded ends," *Bulletin of the Brazilian Mathematical Society*, vol. 15, no. 1-2, pp. 47–54, Mar. 1984.

[2] D. A. Bowman, E. Kruijff, J. J. LaViola, and I. Poupyrev, *3D User Interfaces: Theory and Practice*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2004.

[3] A. Calomeni and W. Celes, "Assisted and automatic navigation in black oil reservoir models based on probabilistic roadmaps," in *Proceedings of the 2006 symposium on Interactive 3D graphics and games - SI3D '06*, ser. I3D '06. New York, New York, USA: ACM Press, 2006, pp. 175–182.

[4] J. Carsten, D. Ferguson, and A. Stentz, "3D Field D: Improved Path Planning and Replanning in Three Dimensions," in *Proceedings of the 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS '06)*, 2006, pp. 3381–3386.

[5] K. Belghith, F. Kabanza, L. Hartman, and R. Nkambou, "Anytime Dynamic Path-planning with Flexible Probabilistic Roadmaps." in *ICRA'06*, 2006, pp. 2372–2377.

[6] M. Kallmann, "Path Planning in Triangulations," in *Proceedings of the IJCAI Workshop on Reasoning, Representation, and Learning in Computer Games*, Edinburgh, Scotland, 2005.

[7] ——, "Shortest paths with arbitrary clearance from navigation meshes," in *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. Madrid, Spain: Eurographics Association, 2010, pp. 159–168.

[8] J. Rosell and P. Iniguez, "Path planning using Harmonic Functions and Probabilistic Cell Decomposition," in *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*. IEEE, 2005, pp. 1803–1808.

[9] M. Trevisan, M. A. P. Idiart, E. Prestes, and P. M. Engel, "Exploratory Navigation Based on Dynamical Boundary Value Problems," *Journal of Intelligent and Robotic Systems*, vol. 45, no. 2, pp. 101–114, May 2006.

[10] A. Treuille, S. Cooper, and Z. Popović, "Continuum crowds," *ACM SIGGRAPH 2006 Papers*, vol. 25, no. 3, pp. 1160–1168, Jul. 2006.

[11] M. J. Park, "Guiding flows for controlling crowds," *The Visual Computer*, vol. 26, no. 11, pp. 1383–1391, Jan. 2010.

[12] R. P. Torchelsen, L. F. Scheidegger, G. N. Oliveira, R. Bastos, and J. a. L. D. Comba, "Real-time multi-agent path planning on arbitrary surfaces," in *I3D '10: Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*. Washington, D.C.: ACM, 2010, pp. 47–54.

[13] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, Dec. 1959.

[14] L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars, "Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, 1996.

[15] J. Barraquand, L. Kavraki, J.-C. Latombe, R. Motwani, T.-Y. Li, and P. Raghavan, "A Random Sampling Scheme for Path Planning," *The International Journal of Robotics Research*, vol. 16, no. 6, pp. 759–774, 1997.

[16] S. M. Lavalle, "Rapidly-Exploring Random Trees: A New Tool for Path Planning," 1998. [Online]. Available: http://msl.cs.uiuc.edu/~lavalle/papers/Lav98c.pdf

[17] S.-J. Kang, Y. Kim, and C.-H. Kim, "Live path: adaptive agent navigation in the interactive virtual world," *The Visual Computer*, vol. 26, no. 6, pp. 467–476, Apr. 2010.

[18] R. Silveira, F. Dapper, E. Prestes, and L. Nedel, "Natural steering behaviors for virtual pedestrians," *The Visual Computer*, vol. 26, no. 9, pp. 1183–1199, Nov. 2009.

[19] E. Prestes, M. A. Idiart, P. M. Engel, and M. Trevisan, "Exploration technique using potential field calculated from relaxation methods," *Intelligent Robots and Systems*, vol. 4, pp. 2012–2017, 2001.

[20] M. D. Berg, O. Cheong, M. V. Kreveld, and M. Overmars, *Computational Geometry: Algorithms and Applications*, 3rd ed. Springer-Verlag, 2008.

[21] L. G. Fischer, R. Silveira, and L. Nedel, "Gpu accelerated path-planning for multi-agents in virtual environments," in *2009 VIII Brazilian Symposium on Games and Digital Entertainment*. IEEE, 2009, pp. 101–110.

Finding hidden objects in large 3D environments: the supermarket problem

Leonardo Fischer, Guilherme Oliveira, Daniel Osmari, Luciana Nedel
Institute of Informatics
Federal University of Rio Grande do Sul – UFRGS
Porto Alegre, Brazil
{*lgfischer, gnoliveira, dkosmari, nedel*}@*inf.ufrgs.br*

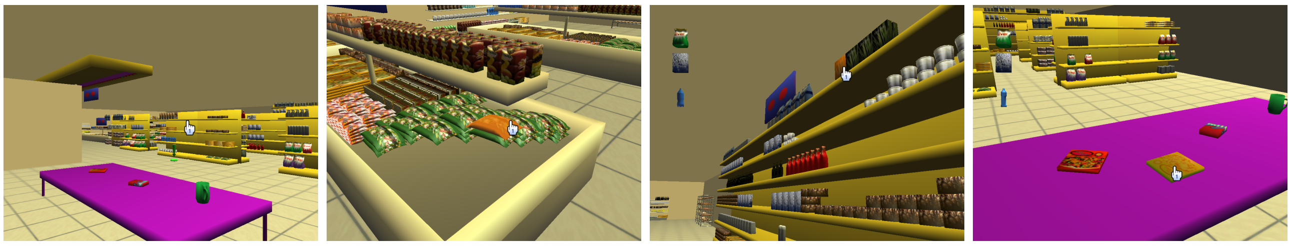

(a) Objects to be found (b) Searching (c) Selecting (d) Manipulating

Figure 1. Four snapshots of the supermarket: (a) copies of the target objects on the purple table; (b) and (c) target objects being selected; (d) a selected object being reoriented to match the same orientation of the copy in the table. Notice, in the top-left corner of images (c) and (d), the list of products that the user is carrying with him.

***Abstract*—Many everyday tasks involve searching for something, selecting, and manipulating it with some cognitive purpose. Even if it is done in a quite automatic manner, due to our experience to deal with real objects, the mapping of these actions to a virtual world is not trivial.**

In this paper we present a solution for this problem, which is composed of three parts: a semi-automatic navigation technique based on path-planning; a selection technique comprising the overexposure of nearby objects; and a manipulation technique based on natural gestures. These three techniques were combined in a smooth way to solve the searching, selecting and manipulating problem. A certain degree of parallelism between them was accepted and is handled by the system. We evaluate our results by user testing in a supermarket scenario. Subjects were invited to walk the hallways of a virtual supermarket looking in the shelves for specific products – sometimes hidden by other objects – that they need to pick up and put over a table in a pre-defined position and orientation. Results showed that our solution fulfill the needs of this kind of complex task, in a natural, funny and attractive way.

***Keywords*-3D interaction; virtual worlds;**

I. Introduction

In the last decade, the popularization of graphics processing units (GPUs) and the emergence of high quality and low cost commodity devices (e.g. Nintendo Wii Remote®, PlayStation Move®, Microsoft Kinect®, and Apple iPhone®) that enable 3D interaction for everyone, motivated the implementation of fully interactive 3D applications.

Examples of this kind of applications may be found in different domains [1], as medicine, for instance. A minimally invasive virtual surgery involves the navigation inside the body, the identification of the target organ, and the manipulation of the tissues (e.g. for suture). In the public safety and military field, the user is invited to walk in a virtual environment, find a bomb and defuse it, precisely manipulating small pieces. Regarding entertainment, thanks to the release of modern video games consoles, many game titles involving natural interaction can be found in the shops shelves.

Although the great part of interactive applications require complete and integrated solutions, research projects usually handle only one interaction technique such as selection, manipulation, or navigation, neglecting the integration with other techniques [2]. The composition of different interaction techniques in a single application is almost as difficult as to propose a new one. An application that require more than one interactive task certainly have dozens of possible solutions. But which technique is really effective to accomplish the task? Combining the best choices for each task that need to be supported may not result in the best solution, because the transition and coupling between the techniques may not be as smooth or natural as expected.

In this paper we propose a solution for a complete interactive 3D application that mimics problems people use to face in real life and that is being explored in industry for simulation software. The user should search for a specific object – sometimes total or partially hidden – in a virtual environment, get it, and put it in a specific position and orientation. We present the interaction techniques we chose for navigation, selection and manipulation and how we combine them into a unique solution that allows smooth transition between them.

The remainder of this paper is structured as follows. Section II reviews some works on 3D interaction and the best solutions for every kind of task. In Section III we detail the problem that motivates this work. The four following sections present an overview of the proposed solution (Section IV), the semi-automatic navigation technique adopted (Section V), the selection technique with the overexposure of objects (Section VI), and the manipulation technique (Section VII). Section VIII describes the user tests and Section IX the results. Section X discusses the lessons learned and Section XI presents our conclusions and future works.

II. Related Work

Users of immersive virtual environments usually intend to manipulate objects that are part of the scene. Reaching objects may also require navigation through the virtual environment since they might not be close to the user. Manipulation requires a previous selection of an object of interest, and navigation is also frequently based on selecting a target point to indicate the path that the user wants to follow.

Developing simple and comfortable selection and manipulation techniques for 3D environments has been a research issue for many years, and there are several possibilities depending on the application-specific tasks, different devices and interaction metaphors. At a high level, these techniques can be classified in two categories according to the interaction metaphor used: the exocentric and the egocentric metaphors [3]. In the exocentric metaphor, the proportions between the user and the objects are not maintained, assuming that the user interacts with the environment from outside of its reference system. This is known as *god's eye viewpoint*. In the egocentric metaphor, the user is part of the virtual world, maintaining the dimensional coherence between him/her and the objects being manipulated. This class is further subdivided into two metaphors: *virtual hand*, where the user reaches and grabs the object of interest with a virtual hand, and *virtual pointer*, where the user interacts with the target object by pointing at it.

With virtual hand techniques, the users can select and directly manipulate virtual objects by *touching* them with their own hands. These techniques are mainly implemented in two different ways: classical virtual hand technique [3], and Go-Go technique [4] that improves the simple virtual hand by allowing the user to interactively change the length of the virtual arm.

The interaction by pointing allows the selection and manipulation of objects located far from the user reaching area. One of the first pointing-based implementations where developed in the 80s by Bolt [5]. Then, many others were proposed, differing in the shape of the pointer, the definition of the virtual pointer direction and the methods of disambiguating the object the user wants to select. The most common example of virtual pointer is the ray-casting technique [6] and its variations, like ray-casting with fishing reel [7], spotlight [8], and aperture selection [9].

Due to the difficulty of conceiving a single best technique for all possible scenarios, some hybrid techniques were proposed. Some of them are very popular as: HOMER (Hand Centered Object Manipulation Extending Ray casting) [7], scaled-world grab [8] and Voodoo Dolls [10].

Navigation in three-dimensional environments is a recurring problem and includes both *travel* and *wayfinding*. Travel corresponds to the motor component of navigation, or the control of the user's position and orientation in the virtual world. Wayfinding involves thinking, planning and choosing a path to follow from one point to another. In other words, represents the high-level part of the navigation task.

A large number of applications require some form of egocentric navigation through a simulated three-dimensional environment, either restricted to the ground – degenerating to an effectively planar situation – or involving free movement in the three-dimensional space.

Several devices and techniques have been tried to solve this problem, most of them presenting limitations. Some techniques require the combination of two or more devices to allow the number of degrees of freedom (DOF) needed for effective navigation, as in the use of the traditional mouse usually coupled with a keyboard for additional DOFs. Others techniques impose adaptation problems to potential users, like the three-dimensional mice, spaceballs, etc. Less common techniques, like the ones based on gesture and body motion, suffer from their own drawbacks, often requiring large and/or expensive setups [11], sometimes limiting the user's natural movements not associated with navigation, or being prone to cause user fatigue in long sessions [12].

Other navigation techniques require that the user physically walks through a real environment [13], [14]. These systems achieve a high sense of immersion by tracking the user position and orientation. A restriction is that they require a large available space so the user can walk freely.

A common solution for travel tasks relies on the continuous control of the orientation of the viewpoint and motion by the user. For example, the user may wear a head mounted display (HMD) where the orientation of his head is tracked [13]. In these cases, the user has the hands free to interact with the virtual environment, but the motion of the viewpoint is highly coupled to view direction in cases where the path is not pre-computed.

In automatic techniques, the user indicates where he/she wants to go. The system is responsible to find a path and moves the viewpoint automatically [15], [16]. Mackinlay et al. [15] proposed an interaction technique where the viewpoint position and speed is controlled based on the user selected point of interest. Hachet et al. [16] proposed improvements by adding some widgets to the technique. In techniques based on sketches, the user draw a path on the environment, and the system controls the motion over that path. Hagedorn and Dollner [17] used a sketch based navigation approach in a touch sensitive display to explore a 3D model of a city.

In this section we addressed the main 3D interaction techniques. However, it is difficult to determine which technique is the best one. In the last few years, many experimental evaluations have been published and some guidelines are accepted nowadays [18]: the interaction technique and the device used should match; one may use pointing techniques for selection and virtual hand for manipulation tasks; when possible, reduce the number of DOFs; consider the use of both natural and magic techniques; use an appropriate combination of technique, display, and input devices; for navigation tasks, avoid teleportation; and consider using multimodal input.

Despite these recommendations, that certainly help the user choose the interactive technique and devices, the achievement of a good interface is not assured. Everytime a new technique, a new device or a new application is proposed, an empirical evaluation should be done to validate the choice.

III. The Supermarket Problem

A supermarket intuitively illustrates the problem presented in Section I and is explored in this work. A person who goes to the supermarket for shopping is exposed to a large and complex environment full of objects (e.g. shelves, products, and so on). Her tasks involves: walk along corridors, with or without a specific destination, but always looking for something; reach products on shelves, putting it inside the shopping cart; and take the products off the shopping cart and pay for it, where shopping ends. Sometimes, people manipulate a specific product to evaluate if it will be bought or not. This sequence of actions is done repeatedly, for many different products, arranged in very different ways into distinct containers.

To guide the development of this work, we use the same specifications given by the organizers of the IEEE 3DUI (Symposium on 3D User Interfaces) interaction contest of 2010 [2]. The task is to find a set of three objects in a supermarket and move them to a table. Copies of the target objects were put on a purple table (see Figure 1(a)), and the objects itself were hidden behind other objects in the aisles marked with a pane with red dots (red dots can be seen in the Figures 1(b) and 1(c)). Users should be able to move along the corridors, find the object similar to the one on the table, and place it beside its copy in roughly the same orientation (Figure 1(d)).

The supermarket scene model comprises hundreds of distinct products, including milk bottles, soap packages, pizzas, cereal boxes, and so on. The products that should be found by users are like others in the shelves, but highlighted with a different color.

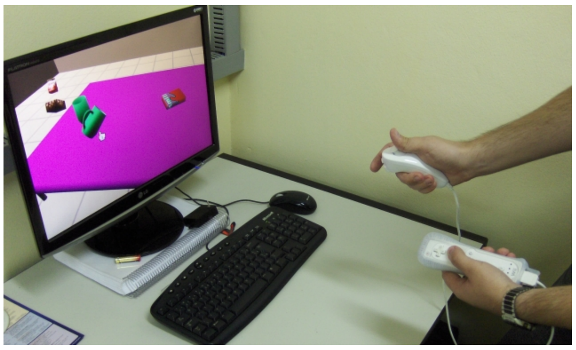


Figure 2. Using the Nintendo Wii Remote® and Nunchuck® to interact with the supermarket scenario. In this stage the Nunchuck® is being used to rotate the selected object so its orientation can match the one of the fixed copy on the table.

IV. Overview of the Solution

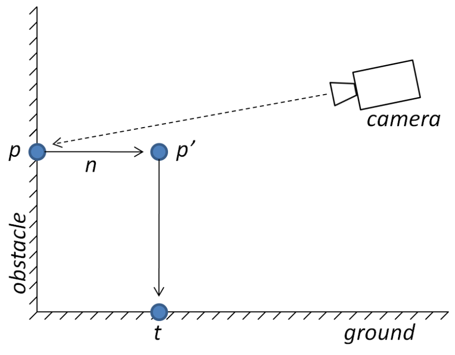
The kind of action to be accomplished in a virtual environment like a supermarket involves three main tasks: navigation, selection, and manipulation. Since the environment is complex and full of objects, we decided to conceive a solution as simple as possible, that minimizes the use of buttons, and avoids any kind of menu to minimize the cognitive overload to the user.

Since some shelves of the virtual supermarket are full of products and the one we are looking for can be partially occluded by others, we decided to avoid techniques that introduce more visual elements to the scene, as the virtual hand. Likewise, the use of ray-casting-based selection of a volume, if not very well calibrated, can allow the selection of several objects at a time by mistake while trying to select the right object.

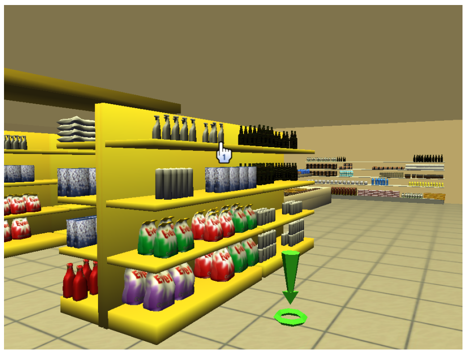
In a supermarket, it is common to walk through a corridor looking to the products on the shelves. Then, the user should be capable of navigate in one direction while looking to another one.

Finally, we also considered two additional points: (i) magic solutions are better than the ones that mimic reality [19], (ii) for the sake of efficiency and simplicity, the same input device should be used for navigation, selection and manipulation.

Therefore, our solution is based on the use of the Nintendo Wii Remote® and the Nunchuck®, a spatially convenient device for 3D interaction [20]. The Wii Remote® is used as a pointing device – through the infrared (IR) sensors – to indicate objects or positions, allowing navigation and selection. Actions are performed when the Wii Remote® buttons are pressed. The manipulation task requires the use of the analog stick of the Nunchuck®, as well as the



(a)



(b)

Figure 3. Computation of the target point for automatic navigation: calculus of the target point (a), and an example of a user pointing to a shelf and the resulting navigation target as a green arrow pointing to the ground (b).

accelerometers of the Wii Remote®. We implement versions of these interaction techniques based on keyboard and mouse input. Figure 2 shows a user in action with the Wii Remote® and Nunchuck®.

We tried to avoid differentiating between the usage of the two main buttons – A and B – of the Wii Remote®. Some users tend to hold the controller as a TV remote (favoring the A button) while others handle it like a gun (favoring the B button). Both buttons perform the same action whenever possible. To confirm and cancel actions we chose the buttons A and B, respectively, as usually assigned in official Wii games. We believe the differentiation on these contexts is more efficient than relying on a menu-based interaction.

The intuition of the solution follows an egocentric metaphor and uses a first person approach, where the camera represents the user eyes. The user takes the Wii Remote® in hands and use it to point in the screen where he/she intend to move, confirming that position by pressing the A or B button. The camera is then smooth and automatically moved to the selected position. During the travel, the user has full control on the gaze.

If, instead of pointing to a new position, the user points to a product and press button A or B, this product will be selected, disappearing of its current position and being shown in the left side of the screen, in a kind of virtual shopping cart. If, however, the user is too far from the product he/she is pointing, it is not selected. Instead, the camera moves to a position near the product. In such a way, we are combining navigation and selection tasks in a single command that is sensitive to the context. This behavior avoids errors due to the small size of far objects.

When the user is near a shelf, some products move a little bit from its current place in a *magic* way. This movement, that we called *explosion of products*, helps the user to see products that are behind others.

Finally, to place a product over the table, the user must point to the table and press button A or B. The product orientation is changed with the help of the Nunchuck® stick and rolling the Wii Remote®.

Our solution was implemented in C++, using OGRE (Open Source 3D Graphics Engine) renderer [21]. We used Blender [22] for modeling.

V. Navigation: Smoothly Moving the Viewer to the Pointing Object

In this section, we will detail our proposal for navigation. Also we will overview the traditional FPS (First-Person-Shooter) navigation technique – used in many video games – which we will compare against our own technique.

We propose a point-and-click solution where the user controls a virtual pointer to show where he/she intends to go on the virtual environment. When the desired position is under the virtual pointer, the user must press a button confirming that target position. Then, the system uses an automatic navigation technique to move the camera to the new position.

We use the IR sensor of the Wii Remote® to know the screen position where the user is pointing to. The user controls a pointer that moves freely on the screen. When the pointer goes near the edges of the screen, the user is able to control the orientation of the camera. For example, if the cursor is next to the upper edge, the camera looks up, and if the cursor is next to the left edge, the camera turns to the left. Orientating the camera does not change its position in the virtual environment. In the case of using the mouse for pointing (instead of the Wii Remote®), the cursor remains always fixed on the center of the screen. Moving the mouse reorients the camera as in a First-Person-Shooter (FPS) game.

With the screen position of the pointer, the camera position and its orientation, the ray-cast pointing technique is used to compute the position p on the surface of the first 3D model in the scene behind the pointer, assuming perspective projection. We also calculate the normal n of that surface. With the position p and the direction n, a point t is computed by creating an intermediate point p' at a certain distance d from the point p in the direction n, and then projecting ortogonally the point p' on the ground. The point t will serve as the target point for the navigation. Figure 3 shows how the target point t is computed and presents an example in our application.

We compute the target point this way to avoid that the user stops its navigation on the exact point where he is pointing to. In real life, when a person is in the supermarket and wants to buy a product,she goes to a position where the object is reachable within a distance of her arm. If the point p is directly projected on the ground to compute the target point t, the user gets so close to the point p that he will need to turn around and walk back some steps to see clearly the desired point. If the user points directly to the ground, the resulting target position t will be the same as the point p. In this case, the user can control the exact position where he wants to go.

The computation of the target point is done every time the cursor or the camera changes its position or orientation. As a feedback clue for the user, we draw a green arrow on the scene over the target point t. The Figure 3(b) shows a picture of this case, with the cursor controlled by the user and the final target point indicated by the green arrow.

Every time the user press the button A or B on the Wii Remote®, the current target point t is used as a new target position to guide the camera movement. We use the potential field algorithm proposed by [23] to calculate a path from the current position to the target one, since it generates smooth motions for the camera.

Once a path is computed, the system starts moving the camera over it. The system does not control directly the speed of the camera. Instead, the system controls the acceleration, deceleration and maximum speed, so the camera does not start or stop its movement abruptly. During the camera movement, the system automatically controls the camera position over the computed path. The camera motion is constrained to the ground and its gaze remains free, so the user can look around.

In order to evaluate our approach, we implemented a second camera control system, similar to the ones found in FPS games. In this implementation, the analog stick of the Nunchuck® controls the camera position on the virtual environment. Push the analog stick forward makes the camera move forward, parallel to the ground, and pull back it makes the camera do the inverse movement. Push the analog stick to the left or right makes the camera strafe (i.e. walk sideways) to the left or to the right, respectively. Letting the analog stick go back to its original position makes the camera stop its motion.

Like in our technique, the orientation of the camera is controlled with the IR sensor of the Wii Remote®. A cursor moves freely on the screen, and when it is near one of the screen edges, the camera turns in that direction.

It is also possible to control the camera with a standard keyboard and mouse. In this case the camera is controlled the same way as it is in most FPS games. The *W* and *S* keys make the camera move forward or backwards, and the *A* and *D* keys make the camera strafe to the left or to the right, respectively. The arrow keys can be used in the same way, with the *up*, *left*, *down* and *right* arrow keys acting as the *W*, *A*, *S* and *D* keys, respectively. The virtual pointer is fixed in the center of the screen, and any motion in the mouse changes the orientation of the camera, i.e., moving the mouse to the right makes the camera also turn right.

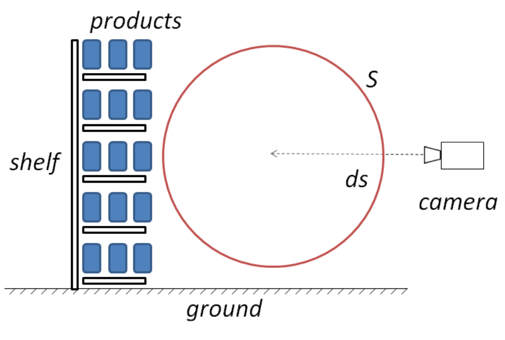
VI. Selection

The same cursor used to navigate is also used for selection. A product of the supermarket can be selected by just pointing the cursor to it and pressing the A or B button on the Wii Remote®. The selected product is removed from the shelf and added to the selected products list, that appears at the left side of the screen. The first product selected is placed at the top-left corner of the screen. The second is placed right below the first one, and so on. Figure 1(d) shows a list of products already selected by the user.

The system lets the user pick a product only if it is close enough. We used a limit distance of 1.2 meters in our tests. This limitation is perceived by the user because we change the color of the products that are in this range slightly. This restriction in the selection range is used to avoid two problems: (i) the first one is to pick products that are too far from the current user position, increasing selection mistakes. In this case the product appears so small that is hard to the user to point exactly to it. (ii) the second problem does not concern the selection itself, but a conflict with navigation. At long distances, it is easy for a user to click in a product instead of a wall or shelf when trying to navigate through the market. So, when the user clicks a product at a distance greater than the threshold, the system interprets this action as a navigation request, instead of selection of that product.

There is another problem inherent of the supermarket scenario that rises in the selection stage. Products vary in shape and size, are placed over shelves of different designs and orientations, and are organized in layers. Thus, several items are occluded by the front most layer of products of the shelf. A customer may want to freely browse the products, even the ones in the back of the shelf, even if it has only the same kind of product. A simple example is to check the product's expiration date. It is a problem to select the desired item if it is in the back layer, or if it has a thin shape, like a pizza box.

To enhance the search for hidden products we defined a behavior for the items based on the idea of *explosion diagrams*. These diagrams are very common in manuals of assembling instructions. They depict the assembling of a complex object by representing the whole object as a set of



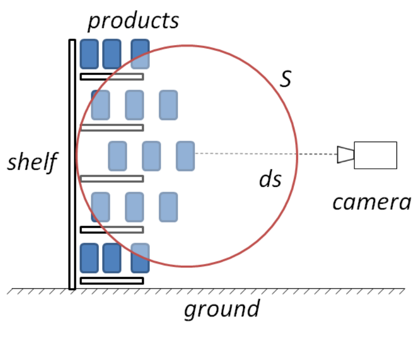


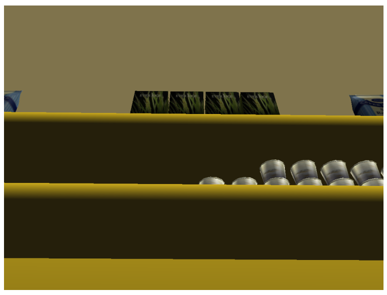
Figure 4. Diagram of the *explosion of products*. (a) Camera far from the products and nothing happens. (b) Camera closer to the products, activating the spreading of products. Observe in the image (b) how the products are moved from its original position (in the figure (a)) to a new position, related to the sphere s.

its minor components, which are slightly moved subcomponents from their original position in the opposite direction from the one that they are attached. This way, the diagram exposes hidden components and make easier for the user to understand how the whole system is composed. Recent works [24], [25] have addressed the automatic creation of such diagrams.

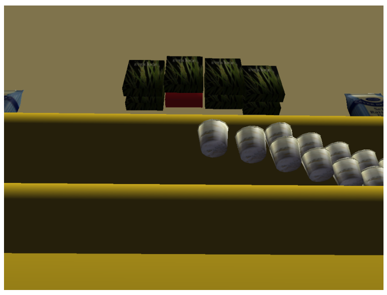
Like an explosion diagram, we move the products from the shelves to expose the hidden items, making the searching process easier. While several different schemes could be used to modify the layout of the items, our choice was to move them along the opposite direction from the one they would normally be placed in the shelf. This drawer-like behavior is quite intuitive and shows good results even when a high amount of items are moved.

First, we construct a sphere s centered in front of the camera, at a distance d_s. We put all products that are inside s in a set p_s. Then, for each product in p_s we compute a direction of displacement. This direction is related to the shelf where the product is placed, so it must be defined to each shelf from the start. That also implies that the items must be logically associated to their shelves. We move the products of p_s, in the displacement direction, with an offset proportional to the distance of the product to the center of s (see Figure 4).

The animation of the position of the products of p_s results in a better distribution of the products on the screen: products that are in front of the shelves still can be easily selected, and products that were hidden by others are now partially visible and selectable. Since the modified layout of the products takes an *explosion shape* we named this process *explosion of products*. Figure 5 shows how some hidden products can be discovered with the *explosion of products* feature enabled.



(a) *Explosion of products* disabled. Some products are hidden behind the front one.



(b) *Explosion of products* allows occluded products to be seen.

Figure 5. The effect of the *explosion of products* feature on the products position, improving the selection of the hidden ones.

VII. Manipulation

The user needs to point to the purple table and hold down the button A or B on the Wii Remote® to start the manipulation task. If the user points to a product that is already over the table, the manipulation is activated on that product. Otherwise, if the user points to an empty space over the table and a product has already been selected from the shelves, the first product from the selected products list (in the upper left side of the screen on the Figure 1(c)) is removed from that list and placed over the table, exactly on the position pointed by the user when the button has been pressed. Then, the manipulation is activated on this product. However, if the user presses the button pointing to an empty space over the table, and there is no products selected, the

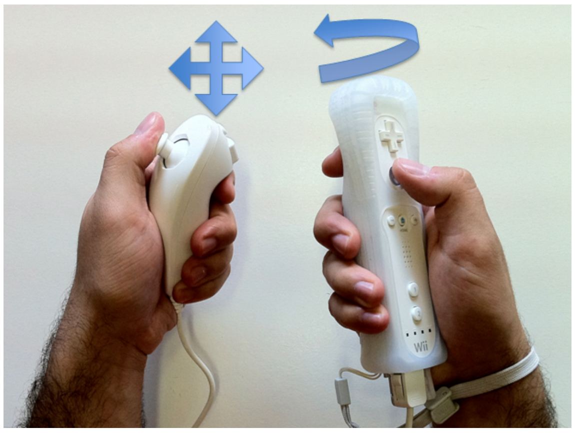


Figure 6. Gestures used to manipulate a product using the Wii Remote® and Nunchuck®.

action is ignored.

Note that the manipulation is enabled only while the user is pressing a button on the Wii Remote®. If the button is released, the system automatically goes back to the navigation/selection mode. In this case, if the user has released the button and is not yet satisfied with the orientation of the product, he/she can again point to the object, select it by pressing the button and manipulate it as he/she want, while the button holds down.

Keeping the button pressed, the orientation of the product can be modified using the Nunchuck® stick and the Wii Remote® accelerometers. The axes of the camera reference system were used as a basis for the product rotation: the camera up vector defines the Y axis of rotation, the direction of view is used as the Z axis, and the cross product between Y and Z is used as the X axis. When the user moves the analog stick of the Nunchuck® to the left-right, or up-down, the product rotates around its Y or X axis respectively. Rolling the Wii Remote® results in a rotation of the product around the Z axis. Figure 6 illustrates the gestures associated.

VIII. User Tests

An experiment was designed with the purpose of understanding the possible advantages of our complete solution. More specifically, we were interested in testing the efficiency, acceptance, comfort and adaptability of the proposed technique for applications that involve the selection of objects spread – sometimes partially occluded by others – in large environments. We also wanted to verify if the Wii Remote® really fits to the techniques proposed and to measure how efficient is its use if compared with the mouse.

A. Tasks and subjects

We designed three tasks with the objective of evaluate navigation and selection separately and then both together. The manipulation was not taken into account in these tests.

TASK 1 was conceived to evaluate the navigation system. The user was invited to navigate through the supermarket aisles to reach a series of waypoints (shown as rotating blue stars) positioned over the scene. Only a single waypoint is visible at each time. When the user reaches a waypoint it disappears, and the next waypoint appears. The new waypoint is visible from the position of the previous waypoint, but requires the user to turn around to find it. Users are aware of this fact. The positions of the waypoints are fixed for all tests and users. This test was executed once for each combination of input devices (Wii Remote® or mouse) and interaction technique (point-and-click or FPS). Each user repeated the test four times.

TASK 2 was proposed to test the impact of the use of the *explosion of products* in the selection and how it works with the Wii Remote® and mouse. The user starts the test in front of a shelf and has to select one specific product. The product distinguishes of the others by its color and is out of the user reach. The user needs first to orient himself to face the product, and then to walk a very small distance. Our reasoning is that, when selecting a product the user might also need to walk to achieve a better angle. So the ability to move while trying to select a product is relevant. When the user selects the first product he/she is moved to another shelf for another instance of the test. The test finishes when the user completes three selections. This task was executed twice for each input device (Wii Remote® or mouse), with and without the *explosion of products* feature, totalizing four tests.

TASK 3 is the full task and consists of navigating through the supermarkets shelves, finding and selecting three specific products, navigating back to the purple table, and placing the objects over the table. The test finishes when the three products are placed on the table (the user is aware of the locations of the products). This test was executed twice, once for each input device (Wii Remote® or mouse), with the use of the explosion animation.

The experiment was performed by a group of 38 subjects, 33 male and 5 female, aging from 19 to 37 years old (average = 22.8; standard deviation = 3.9), all of them undergraduate students in Computer Science. Each of them has done the three tasks with the different conditions previewed, resulting in 10 tests per user, and a total of 380 tests.

In a pre-test form, we made a set of questions to the users, asking them to characterize themselves in a scale from 1 to 5, meaning *very little experience* and *highly experienced*, respectively. Each question asked about the user experience with computer games, 3D environments, FPS games, and use of the Wii Remote®. We summarized these answers in the Table I, presenting the average and standard deviation.

B. Procedure and variables

Before the experiment starts, users were invited to fill a self characterization questionnaire, instructed on how to

Table I
CHARACTERIZATION OF THE SUBJECTS, ACCORDING TO THE ANSWERS IN THE PRE-TEST QUESTIONAIRE.

Question	Average	Std. dev.
Experience with computer games	4.10	0.83
Experience with 3D virtual environments	3.42	1.00
Experience with FPS games	3.78	1.04
Experience with Wii Remote®	2.02	1.26

operate the devices, and encouraged to perform as many practice trials as wished so that they could feel confident during the tests. Then the three sets of tasks are performed, in order.

For each task, the execution of its tests was randomized in order to try to prevent bias. From each one of the twelve runs we collected the execution times in a log file. The users also filled post-experiment surveys for qualitative analysis including the following data: which technique is more efficient for navigation (point-and-click with the Wii Remote®, point-and-click with the mouse, FPS with mouse and keyboard), which technique is more intuitive and easiest to learn for navigation (point-and-click with the Wii Remote®, point-and-click with the mouse, FPS with mouse and keyboard), which technique is more fun to navigate (point-and-click with the Wii Remote®, point-and-click with the mouse, FPS with mouse and keyboard), which technique is more efficient for selection (point-and-click with Wii Remote® or mouse), which technique is more intuitive and easiest to learn for selection (point-and-click with the Wii Remote®, point-and-click with the mouse, FPS with mouse and keyboard), and which selection technique is more fun (point-and-click with the Wii Remote®, point-and-click with the mouse, FPS with mouse and keyboard). We also offered a space for free comments and opinions.

IX. RESULTS

The tasks described in Section VIII-A and tested with the 38 subjects allowed the capture of objective and subjective data that are presented below. Completion times for the tasks were recorded in log files and used as input for ANOVA (Analysis of Variance) test.

Regarding navigation, we tested four different configurations and measured times for completion. Mean times and standard deviation for the four configurations tested were calculated and can be seen in Figure 7. The use of the mouse is always more efficient than the Wii Remote®, presenting a mean time significantly smaller ($F = 12.4140$; $p < 0.0005$). Comparing the performance with the two navigation techniques tested, we can see that the FPS technique is more efficient than the point-and-click, with a mean time significantly smaller ($F = 25.0174$; $p < 0.0000016$).

From the subjective data, that represents the opinion of the subjects, we can observe that 76% of the users found the use of mouse and keyboard is more intuitive than the

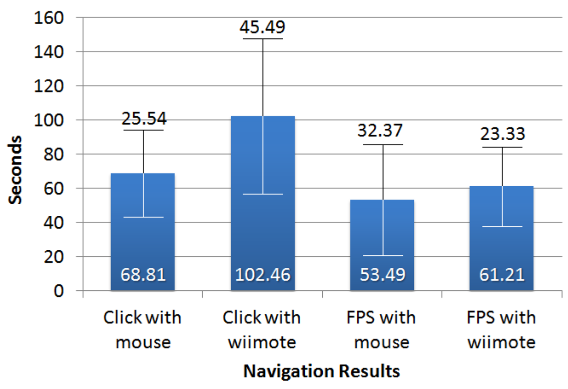


Figure 7. Navigation task performance in four different conditions: point-and-click technique with mouse; point-and-click with Wii Remote®; FPS with mouse and keyboard; and FPS with Wii Remote®. The numbers at the bottom of the bars represents the mean time, while the numbers at the top of the bars means the standard deviation.

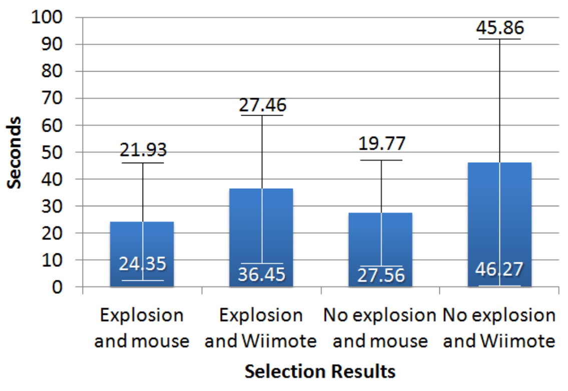


Figure 8. Selection task performance in four different conditions: using the *explosion of products* associated to the mouse or Wii Remote®; without the explosion effect and using the mouse or Wii Remote®. The numbers at the bottom of the bars represents the mean time, while the numbers at the top of the bars means the standard deviation.

Wii Remote®. For the other hand, 86% of these same users declared that the Wii Remote® is more fun than the mouse.

With respect of selection tests, mean times and standard deviation for the four configurations tested can be seen in Figure 8. The use of the *explosion of products* presents the best mean times when associated with the use of a mouse or Wii Remote®. However, these mean times have no statistic significance ($F = 1.6421$; $p < 0.202$).

In the last set of tests, subjects where asked to accomplish the *TASK 3*, as described in Section VIII-A. The mean times and standard deviation are shown in Figure 9. As it can be seen, the mouse and keyboard is more efficient than the Wii Remote®, presenting a mean time significantly smaller ($F = 8.7025$; $p < 0.0042$).

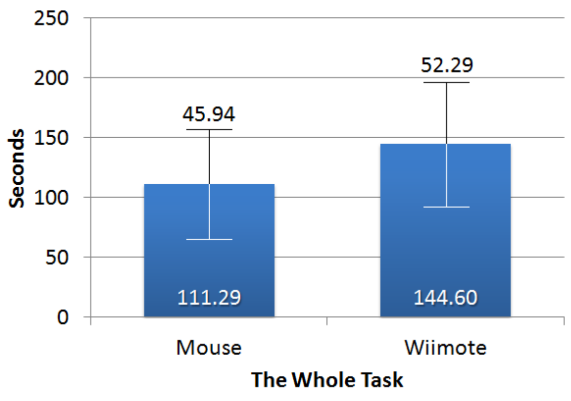


Figure 9. Mean performance achieved in the completion the whole task. The numbers at the bottom of the bars represents the mean time, while the numbers at the top of the bars means the standard deviation.

X. Discussion

Some of the results obtained with the user tests were more or less expected by us. The higher performance of the users with the mouse, if compared with the Wii Remote®, is quite obvious. Mouse and keyboard are working tools as common as pen and paper for computer science students. On the other hand, 72% of the subjects declared to have no or few previous experience with the Wii Remote®. Considering this unfair comparison, we believe that the performance with the Wii Remote® is sufficiently good to consider the use of this device in adverse situation where, for example, the user cannot sit down in front of the computer and use the mouse and keyboard as comfortably as they use to do.

The best performance achieved with the FPS navigation technique, instead of the use of the point-and-click technique was a surprise for us. Observing the subjects during the tests, we noted that they clicked a lot when using the point-and-click technique. In fact, much more than expected. This behavior can be, in such a way, explained by the profile of the users; 62% of the users declared to be hard users of FPS games. We also think that the point-and-click technique is not well suited for the supermarket scenario, because of its characteristics (small aisles with high shelves). In an open-air scenario, where greater distances are seen, the user can take more advantage of the automatic definition of a valid path to follow.

The performance improvement achieved with the use of the *explosion of products* effect for selection were also expected. We could not reach significance with the ANOVA test, but we intend to do more tests with users, varying the position and size of the products to be selected, in order to understand these results.

As mentioned in Section VIII-A, in this study we did not evaluate the manipulation task. However, during the execution of the complete task (*TASK 3*), the subjects had also the opportunity of manipulate the products over the purple table. Many users mentioned in the pos-test questionnaire that the Wii Remote® and the Nunchuck® were more comfortable than the mouse (that was considered very uncomfortable), but they also claimed that the Wii Remote® was not sufficiently accurate. In fact, the rotation given by the Wii Remote® accelerometers is not very precise at all, what probably motivated Nintendo to develop another device – the MotionPlus – based on gyroscopes. Unfortunately, we could not use it during the development of this work.

XI. Conclusion

In this paper we presented a complete and integrated solution to solve everyday interactive 3D tasks that involves navigation, selection and manipulation of objects that are, frequently, hidden by others. In our proposal, we tried to keep almost the same interactive devices and techniques for all the tasks, avoiding the exchange of devices and reducing the cognitive overload inherent to the interaction.

The results demonstrate that our approach is promising, but more work should be done in order to have a robust solution. As future work, we intend to reimplement the manipulation technique using more robust sensors, as the gyroscopes present in new commodity devices (e.g. iPhone, and Nintendo Motion Plus). New tests should also be done including manipulation and considering subjects with little experience in 3D games and less trained in the interactive techniques commonly used in this kind of application.

Acknowledgment

The authors would like to thank CNPq-Brazil through projects 483947/2010-5, 309092/2008-6 and 580156/2008-7 for partially supporting this work, as well as all the volunteers for their kind participation in the experiments. A special thanks to Leandro Tibola, Leonardo Santagada and Rafael Kovaleski for their help in the design and implementation of the first version of the software, and Vitor Pamplona and Marilena Maule for their comments on this text.

References

[1] A. Craig, W. R. Sherman, and J. D. Will, *Developing Virtual Reality Applications: Foundations of Effective Design*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2009.

[2] P. Figueroa, Y. Kitamura, S. Kuntz, L. Vanacken, S. Maesen, T. D. Weyer, S. Notelaers, J. R. Octavia, A. Beznosyk, K. Coninx, F. Bacim, R. Kopper, A. Leal, T. Ni, and D. A. Bowman, "3dui 2010 contest grand prize winners," *IEEE Computer Graphics and Applications*, vol. 30, pp. 86–96, c3, 2010.

[3] I. Poupyrev, S. Weghorst, M. Billinghurst, and T. Ichikawa, "Egocentric object manipulation in virtual environments: Empirical evaluation of interaction techniques," in *Proceedings of the EUROGRAPHICS'98 Conference*. Eurographics and Blackwell Publishing Ltd., 1998, pp. 41–52.

[4] I. Poupyrev, M. Billinghurst, S. Weghorst, and T. Ichikawa, "The go-go interaction technique: non-linear mapping for direct manipulation in vr," in *UIST '96: Proceedings of the 9th annual ACM symposium on User interface software and technology*. New York, NY, USA: ACM, 1996, pp. 79–80.

[5] R. A. Bolt, ""put-that-there": Voice and gesture at the graphics interface," in *Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '80. New York, NY, USA: ACM, 1980, pp. 262–270. [Online]. Available: http://doi.acm.org/10.1145/800250.807503

[6] M. R. Mine, "Virtual environment interaction techniques," Chapel Hill, NC, USA, Tech. Rep., 1995.

[7] D. A. Bowman and L. F. Hodges, "An evaluation of techniques for grabbing and manipulating remote objects in immersive virtual environments," in *SI3D '97: Proceedings of the 1997 symposium on Interactive 3D graphics*. New York, NY, USA: ACM, 1997, pp. 35–38.

[8] M. R. Mine, F. P. B. Jr., and C. H. Séquin, "Moving objects in space: exploiting proprioception in virtual-environment interaction," in *SIGGRAPH*, 1997, pp. 19–26.

[9] A. Forsberg, K. Herndon, and R. Zeleznik, "Aperture based selection for immersive virtual environments," in *UIST '96: Proceedings of the 9th annual ACM symposium on User interface software and technology*. New York, NY, USA: ACM, 1996, pp. 95–96.

[10] J. S. Pierce, B. C. Stearns, and R. Pausch, "Voodoo dolls: seamless interaction at multiple scales in virtual environments," in *I3D '99: Proceedings of the 1999 symposium on Interactive 3D graphics*. New York, NY, USA: ACM, 1999, pp. 141–145.

[11] R. P. Darken, W. R. Cockayne, and D. Carmein, "The omni-directional treadmill: a locomotion device for virtual worlds," in *UIST '97: Proceedings of the 10th annual ACM symposium on User interface software and technology*. New York, NY, USA: ACM, 1997, pp. 213–221.

[12] F. Sparacino, C. Wren, A. Azarbayejani, and A. Pentland, "Browsing 3-d spaces with 3-d vision: body-driven navigation through the internet city," in *International Symposium on 3D Data Processing Visualization and Transmission*. Los Alamitos, CA, USA: IEEE Computer Society, 2002, pp. 224–233.

[13] G. Bruder, F. Steinicke, and K. H. Hinrichs, "Arch-explore: A natural user interface for immersive architectural walk-throughs," *3D User Interfaces*, vol. 0, pp. 75–82, 2009.

[14] E. Foxlin, "Pedestrian tracking with shoe-mounted inertial sensors," *IEEE Computer Graphics and Applications*, vol. 25, pp. 38–46, 2005.

[15] J. D. Mackinlay, S. K. Card, and G. G. Robertson, "Rapid controlled movement through a virtual 3d workspace," in *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM, 1990, pp. 171–176.

[16] M. Hachet, F. Dècle, S. Knödel, and P. Guitton, "Navidget for easy 3d camera positioning from 2d inputs," in *Proceedings of IEEE 3DUI - Symposium on 3D User Interfaces*, 2008, best paper award. [Online]. Available: http://iparla.labri.fr/publications/2008/HDKG08

[17] B. Hagedorn and J. Döllner, "Sketch-based navigation in 3d virtual environments," in *Smart Graphics*, ser. Lecture Notes in Computer Science, A. Butz, B. Fisher, A. Krüger, P. Olivier, and M. Christie, Eds., vol. 5166. Springer Berlin / Heidelberg, 2008, pp. 239–246. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-85412-8_23

[18] D. A. Bowman, E. Kruijff, J. J. Laviola, and I. Poupyrev, "An introduction to 3-d user interface design," in *Presence: Teleoperators and Virtual Environments*, 2001, pp. 96–108.

[19] D. A. Bowman, E. Kruijff, J. J. LaViola, and I. Poupyrev, *3D User Interfaces: Theory and Practice*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2004.

[20] C. A. Wingrave, B. Williamson, P. D. Varcholik, J. Rose, A. Miller, E. Charbonneau, J. Bott, and J. J. L. Jr., "The wiimote and beyond: Spatially convenient devices for 3d user interfaces," *IEEE Computer Graphics and Applications*, vol. 30, pp. 71–85, 2010.

[21] G. Junker, *Pro OGRE 3D Programming (Pro)*. Berkely, CA, USA: Apress, 2006.

[22] R. Hess, *The Essential Blender: Guide to 3D Creation with the Open Source Suite Blender*. San Francisco, CA, USA: No Starch Press, 2007.

[23] R. Silveira, F. Dapper, E. Prestes, and L. Nedel, "Natural steering behaviors for virtual pedestrians," *Visual Comput.*, 2009.

[24] M. Tatzgern, D. Kalkofen, and D. Schmalstieg, "Compact explosion diagrams," in *NPAR '10: Proceedings of the 8th International Symposium on Non-Photorealistic Animation and Rendering*. New York, NY, USA: ACM, 2010, pp. 17–26.

[25] D. Kalkofen, M. Tatzgern, and D. Schmalstieg, "Explosion diagrams for augmented reality," in *Proceedings of the IEEE Virtual Reality Conference (VR 09)*. IEEE Computer Society, 2009, pp. 71–78.

Path-Planning for RTS Games Based on Potential Fields

Renato Silveira, Leonardo Fischer, José Antônio Salini Ferreira, Edson Prestes, and Luciana Nedel

Universidade Federal do Rio Grande do Sul, Brazil,
{rsilveira, lgfischer, jasferreira, prestes, nedel}@inf.ufrgs.br,
Home page: `http://www.inf.ufrgs.br`

Abstract. Many games, in particular RTS games, are populated by synthetic humanoid actors that act as autonomous agents. The navigation of these agents is yet a challenge if the problem involves finding a precise route in a virtual world (path-planning), and moving realistically according to its own personality, intentions and mood (motion planning). In this paper we present several complementary approaches recently developed by our group to produce quality paths, and to guide and interact with the navigation of autonomous agents. Our approach is based on a BVP Path Planner that generates potential fields through a differential equation whose gradient descent represents navigation routes. Resulting paths can deal with moving obstacles, are smooth, and free from local minima. In order to evaluate the algorithms, we implemented our path planner in a RTS game engine.

Keywords: Path-planning, navigation, autonomous agent

1 Introduction

Recent advances in computer graphics algorithms, especially on realistic rendering, allow the use of synthetic actors visually indistinguishable from real actors. These improvements benefit both the movie and game industry which make extensive use of virtual characters that should act as autonomous agents with the ability of playing a role into the environment with life-like and improvisational behavior. Despite a realistic appearance, they should present convincing individual behaviors based on their personality, moods and desires. To behave in such way, agents must act in the virtual world, perceive, react and remember their perceptions about this world, think about the effects of possible actions, and finally, learn from their experience [7].

In this complex and suitable context, navigation plays an important role [12]. To move agents in a synthetic world, a semantic representation of the environment is needed, as well as the definition of the agent initial and goal position. Once these parameters were set, a path-planning algorithm must be used to find a trajectory to be followed.

However, in the real world, if we consider different persons – all of them in the same initial position – looking for achieving the same goal position, each path followed will be unique. Even for the same task, the strategy used for each person to reach his/her goal depends on his/her physical constitution, personality, mood, reasoning, urgency, and so on. From this point of view, a high quality algorithm to move characters across virtual environments should generate expressive, natural, and occasionally unexpected steering behaviors. In contrast, especially in the game industry, the high performance required for real-time graphics applications compels developers to look for most efficient and less expensive methods that produce good and almost natural movements.

In this paper, we present several complementary approaches recently developed by our group [4, 6, 14, 15, 18] to produce high quality paths and to control the navigation of autonomous agents. Our approach is based on the BVP Path Planner [14], that is a method based on the numeric solution of the boundary value problem (BVP) to control the movement of agents, while allowing the individuality of each one.

The topics presented in this article are: (*i*) a robust and elegant algorithm to control the steering behavior of agents in dynamic environments; (*ii*) the production of interesting and complex human-like behaviors while building a navigational route; (*iii*) a strategy to handle the path-planning for group of agents and the group formation-keeping problem, enabling the user to sketch any desirable formation shape; (*iv*) a sketch based global planner to control the agent navigation; (*v*) a strategy to deal with the BVP Path Planner in GPU; (*vi*) a RTS game implementation using our technique.

The remaining of this paper is structured as follows. Section 2 reviews some related works on path-planning techniques that generate quality paths and behaviors. Section 3 explains the concepts of our planner and how we deal with agents and group of agents. Section 4 proposes an alternative to our global path planner, enabling the user interaction. Improvements in the performance are discussed in Section 5. Some results, including a RTS game implementation using our technique, is shown in Section 6. Section 7 presents our conclusions and some proposals for future works.

2 Related Work

The research on path-planning has been extensively explored on the games domain where the programmer should frequently deal with many autonomous characters, ideally presenting convincing behavior. It is very difficult to produce natural behavior by using a strategy focusing on the global control of characters. On the other hand, taking into account the individuality of each character may be a costly task. As a consequence, most of the approaches proposed in computer graphics literature do not take into account the individual behavior of each agent, compromising the planner quality.

Kuffner [8] proposed a technique where the scenario is mapped onto a 2D mesh and the path is computed using a dynamic programming algorithm like

Dijkstra. He argues that his technique is fast enough to be used in dynamic environments. Another example is the work developed by Metoyer and Hodgings [11], that proposes a technique where the user defines the path that should be followed by each agent. During the motion, the path is smoothed and slightly changed to avoid collisions using force fields that act on the agent.

The development of randomized path-finding algorithms – specially the PRM (Probabilistic Roadmaps) [10] – allowed the use of large and more complex configuration spaces to efficiently generate paths. There are several works in this direction [3, 13]. In most of these techniques, the challenge becomes more the generation of realistic movements than finding a valid path. Differently, Burgess and Darken [2] proposed a method based on the *principle of least action* which describes the tendency of elements in nature to seek the minimal effort solution. The method produces human-like movements through realistic paths using properties of fluid dynamics.

Tecchia et al. [16] proposed a platform that aims to accelerate the development of behaviors for agents through local rules that control these behaviors. These rules are governed by four different control levels, each one reflecting a different aspect of the behavior of the agent. Results show that, for a fairly simple behavioral model, the system performance can achieve interactive time. Treuille et al. [17] proposed a crowd simulator driven by dynamic potential fields which integrates global navigation and local collision avoidance. Basically, this technique uses the crowd as a density field and constructs, for each group, a unit cost field which is used to control people displacement. The method produces smooth behavior for a large amount of agents at interactive frame rates. Based on local control, van den Berg [1] proposed a technique that handles the navigation of multiple agents in the presence of dynamic obstacles. He uses a concept, called *velocity obstacles*, to locally control the agents with few oscillation.

As mentioned above, most of the approaches do not take into account the individual behavior of each agent, his internal state or mood. Assuming that realistic paths derive from human personal characteristics and internal state, thus varying from one person to another, we [14] recently proposed a technique that generates individual paths. These paths are smooth and dynamically generated while the agent walks. In the following sections, we will explain the concepts of our technique and the extensions implemented to handle several problems found in RTS games.

3 BVP Path Planner

The BVP Path Planner, originally proposed by Trevisan et al. [18], generates paths using the potential information computed from the numeric solution of

$$\nabla^2 \, p(\mathbf{r}) \; = \; \epsilon \mathbf{v} \cdot \nabla p(\mathbf{r}) \; , \tag{1}$$

with Dirichlet boundary conditions. In Equation 1, $\mathbf{v} \in \Re^2$ and $|\mathbf{v}| = \mathbf{1}$ corresponds to a vector that inserts a perturbation in the potential field; $\epsilon \in \Re$ corresponds to the intensity of the perturbation produced by $\mathbf{v}$; and $p(\mathbf{r})$ is the potential at position $\mathbf{r} \in \Re^2$. Both $\mathbf{v}$ and ϵ must be defined before computing this equation. The gradient descent on these potentials represents navigational routes from any point of the environment to the goal position. Trevisan et al. [18] showed that this equation does not produce local minima and generates smooth paths.

To solve numerically a BVP, we can consider that the solution space is discretized in a regular grid. Each cell (i, j) is associated to a squared region of the environment and stores a potential value $p(i, j)$. Using Dirichlet boundary conditions, the cells associated to obstacles in the real environment store a potential value of 1 (*high potential*) whereas cells containing the goal store a potential value of 0 (*low potential*).

A high potential value prevents the agent from running into obstacles whereas a low value generates an attraction basin that pulls the agent. The relaxation method usually employed to compute the potentials of free space cells is the Gauss-Seidel (GS) method. The GS method updates the potential of a cell c through:

$$p_c = \frac{p_b + p_t + p_r + p_l}{4} + \frac{\epsilon((p_r - p_l)v_x + (p_b - p_t)v_y)}{8} \tag{2}$$

where $\mathbf{v} = (v_x, v_y)$, and $p_c = p_{(i,j)}$, $p_b = p_{(i,j-1)}$, $p_t = p_{(i,j+1)}$, $p_r = p_{(i+1,j)}$ and $p_l = p_{(i-1,j)}$. The potential at each cell is updated until the convergence sets in.

After the potential computation, the agent moves following the direction of the gradient descent of this potential at its current position (i, j),

$$(\nabla \mathbf{p})_c = \left(\frac{p_l - p_r}{2}, \frac{p_b - p_t}{2} \right).$$

In order to implement the BVP Path Planner, we used global environment maps (one for each goal) and local maps (one for each agent). The global map is the Global Path Planner which ensures a path free of local minima, while the local map is used to control the steering behavior of each agent, also handling dynamic obstacles.

The entire environment is represented by a set of homogeneous meshes $\{\mathcal{M}_k\}$, where each mesh $\mathcal{M}_k$ has $L_x \times L_y$ cells, denoted by $\{\mathcal{C}^k_{i,j}\}$. Each cell $\mathcal{C}^k_{i,j}$ corresponds to a squared region centered in environment coordinates $r = (r_i, r_j)$ and stores a particular potential value $\mathcal{P}^k_{i,j}$. The potential associated to each cell is computed by GS iterations (Equation 2) and then used by the agents to reach the goal. In order to delimit the navigation space, we consider that the environment is surrounded by static obstacles.

3.1 Dealing with Individuals

Each agent a_k has one local map m_k that stores the current local information about the environment obtained by its own sensors. This map is centered in the current agent position and represents a small fraction of the global map. The size of the local map influences the agent behavior. A detailed analysis of the influence of the size of the local map on the behavior of the agent can be found in [14].

The local map is divided in three regions: the update zone (*u-zone*); the free zone (*f-zone*); and the border zone (*b-zone*), as shown in Figure 1. Each cell corresponds to a squared region, similar to the global map. For each agent, a goal, a particular vector $\mathbf{v}_k$ that controls its behavior, and a ϵ_k should be stated. If $\mathbf{v}_k$ or ϵ_k is dynamic, then the function that controls it must also be specified.

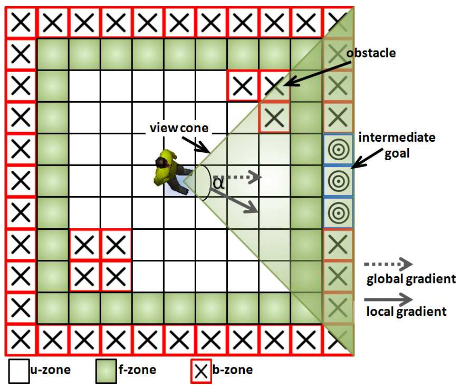


Fig. 1. Agent Local Map. White, green and red cells comprise the *update*, *free* and *border zones*, respectively. Blue, red and light blue cells correspond to the intermediate goal, obstacles and the agent position, respectively.

To navigate into the environment, an agent a_k uses its sensors to perceive the world and to update its local map with information about obstacles and other agents. The agent sensor sets a view cone with aperture α. Figure 1 sketches a particular instance of the agent local map. The *u-zone* cells that are inside the view cone and correspond to obstacles or other agents have their potential value set to 1. Dynamic or static obstacles behind the agent do not interfere in its future motion.

For each agent a_k, we calculate the global descent gradient on the cell of the global map containing its current position. The gradient direction is then used to generate an intermediate goal in the border of the local map, setting the potential values to 0 of a couple of *b-zone* cells, while the other *b-zone* cells are considered as obstacles, with their potential values set to 1. The intermediate goal helps the agent a_k to reach its goal while allowing it to produce a particular motion.

F-zone cells are always considered free of obstacles, even when there are obstacles inside. The absence of this zone may close the connection between the current agent cell and the intermediate goal due to the possible mapping of obstacles in front of the intermediate goal. When this occurs, the agent gets lost because there is no information coming from the intermediate goal to produce a path to reach. *F-zone* cells handle the situation, always allowing the propagation of the information about the goal to the cells associated to the agent position.

After the sensing and mapping steps, the agent k updates the potential value of its map cells using Equation 2 with its pair $\mathbf{v}^k$ and ϵ^k. Hereinafter, it updates its position according to:

$$\Delta\, \mathbf{d} = \upsilon\ (\cos(\varphi^t), \sin(\varphi^t))\ \Psi(|\varphi^{t-1} - \zeta^t|) \tag{3}$$

with

$$\varphi^t = \eta\ \varphi^{t-1} + (1-\eta)\ \zeta^t \tag{4}$$

where υ defines the maximum agent speed, $\eta \in [0, 1)$, φ is the agent orientation and ζ is the orientation of the gradient descent computed from the potential field stored on its local map in the central position. Function $\Psi : \Re \rightarrow \Re$ is

$$\Psi(x) = \begin{cases} 0 & \text{if } x > \pi/2 \\ \cos(x) & \text{, otherwise} \end{cases}$$

If $|\varphi^{t-1} - \zeta^t|$ is higher than $\frac{\pi}{2}$, then there is a high hitting probability and this function returns the value 0, making the agent stops. Otherwise, the agent speed will change proportionally to the collision risk.

In order to demonstrate that the proposed technique produces realistic behaviors for humanoid agents, we compared the paths generated by our method with real human paths [14] . Associating specific values to the parameters ϵ and $\mathbf{v}$ in the agent local map, the path produced by the BVP Path Planner almost mimics the human path. Figure 2 shows a path generated by dynamically changing ϵ and $\mathbf{v}$. Up to the half of the path, the parameters $\epsilon = 0.1$ and $\mathbf{v}= (0, -1)$ were used. Half the path forward, the parameters were changed to $\epsilon = 0.7$ and $\mathbf{v}= (1, 0)$. These values were empirically obtained. We can visually compare and observe that the calculated path is very close to the real one.

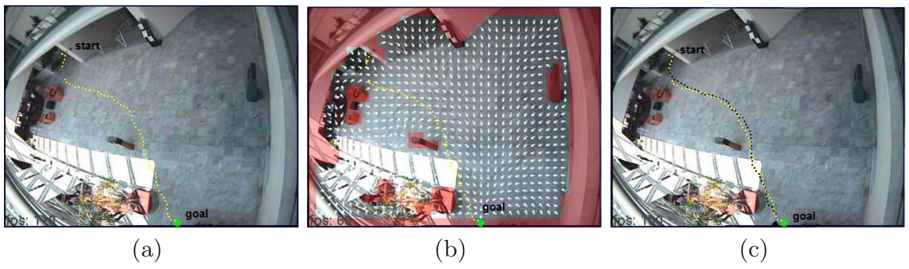


Fig. 2. Natural path generated by the BVP Path Planner: (a) the environment with the real person path (yellow); (b) the environment representation discretized by our planner; (c) the agent path (in black) calculated by our planner after adjusting the parameters.

3.2 Dealing with Groups

The organization of agents in groups has two main goals: to allow the control of groups or armies by the user, and to decrease the computational cost through the management of groups instead of handling individuals. In a previous work [15], an approach to integrate group control in the BVP Path Planner was proposed. This approach also support the group formation-keeping while agents move following a given path. Kallmann [9] recently proposed a path planner to efficiently produce quality paths that could also be used for moving groups of agents. Our technique, illustrated in Figure 3, focus on the group cohesion and behavior while enabling the user to interact with the group.

In this approach, each group is associated to a specific formation and a map, called *group map*. The user should provide – or dynamically sketch – any desired shape for the group formation. This formation is then discretized into a set of points that can be seen as attraction points associated to agents – one point attracting each agent towards him. Analogous to an agent local map, the group map is then projected into the environment and its center is aligned with the center of the group of agents in the environment. The center of the formation shape is also aligned with the center of the group map.

Obstacles and goals are mapped to this map in the same way that we have done for the local maps. However, in the group map there is no view cone. Each agent in a group is mapped to its respective position on the group map as an obstacle. In order to obtain the information about the proximity of an agent in relation to the obstacles, we divide the group map into several regions surrounding the obstacles. Each cell in a region has a scalar value that is used to weight the distance between an agent and an obstacle. When the agent is in a cell associated to any of these regions, it will be influenced not only by the force exerted by the formation, but also by the gradient descent computed at its position in the group map. After that, the vector field is extracted and the agent motion is then influenced by two forces: formation force, and the *group map* vector field. As both forces influence in the path definition, they should be properly established, in order to avoid undesired or non realistic behaviors (for more details on combining these forces, refers to [15].

The motion of the agents which are inside the group map is established using these forces while the motion of the entire group is produced by moving the group map along the global map. For this, we consider the group map center point as a single agent and any path planner algorithm can be used to obtain a path free of collisions.

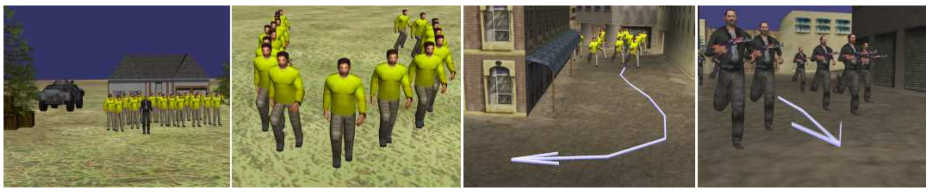


Fig. 3. Group control: agents can keep a formation or move freely; the user can interact and sketch trajectories to be followed by the groups.

4 User Interaction with RTS Games

As seen in Section 3, our technique uses a global map to make a global path-planning, and local or group maps to avoid the collision with dynamic elements of a game (e.g. units, enemies, moving obstacles). This fits very well to most RTS games, where the player selects some units and click on a desired location in order to give a “go there” command.

This kind of game-play commonly requires several mouse clicks to give a specific behavior. RTS players commonly want that a group of units overcome an obstacle by following a specific path that conforms to his/her own strategy. Define a strategy in a high level of abstraction is generally hard to be done with only a few mouse clicks.

Based on the ideas of a previous work from our group [5], we suggested an interaction technique where the units are controlled by a sketch based navigation scheme, as an alternative to the global map . The player clicks with the mouse on the screen and draws the desired path for the currently selected units. The common technique of just clicking on the goal location is also supported. This way of sketching the path to be followed is the same one used in paint-like applications, and can be easily adapted to touch screen displays, like Microsoft Surface®and Apple iPad®, for instance.

4.1 Basic Implementation

The technique is divided in two steps: an *input capture* step, where the user draws the path to be followed; and a *path following* step, where the army units run to their goals. In the first step, the path can be drawn by dragging the mouse with a button pressed, or by dragging his/her finger over a touch screen surface. When the user releases the mouse button or removes his/her finger from the screen surface, a list of 2D points is taken and projected against the battlefield, resulting in a list of 3D points.

In the second step, the points generated on the first step are put in a list and used as intermediary goals for the units. Following this list, each point is used as the position goal for all selected units. When the first unit reaches this goal, it is discarded and the next one in the list is used. This continues until the first unit of the group reaches the last goal.

4.2 Splitting the Army

Imagine a situation where the player has walking soldiers and tanks available to attack, and the enemy headquarters is protected by one of these sides by a swamp. Only walking soldiers can walk through it. Then, the player may choose to attack the enemy by one side with the tanks, and the protected side with the walking soldiers.

An extension of the sketching technique may let the player use this strategy by simply drawing one path to the enemy headquarters, where in a certain division point the path divides into two parts: one goes through the swamp and the other avoids it. Then, all units (walking soldiers and tanks) follow the path, and in the division point, the army divides into two groups, one that can trespass the swamp and another that goes through mainland.

In order to allow this maneuver, we suggested a structure where the user sketches are stored in a tree. In this tree, each node represents a division point, start point or end point of the motion, and the links between nodes store one section of the path drawn by the player. As in the Section 4.1, each path section is stored as a list of 3D points.

When the player draws the first path, a node a is created for the 3D point where the path starts, and a node b where the path ends. A link storing all the points between a and b is created, with b being child of a. Then, the user draws a second path, starting nearly the point of division chosen by the player along the link. Considering a tree composed by the previously drawn paths, we search for the link l with the path section that contains the closest point to the start point of the newly drawn path. This link l is broken in two parts, l_1 and l_2, and a new end node p is created between both. Finally, we create a new end node c, a link between p and c, and attach the recently drawn path section to the tree.

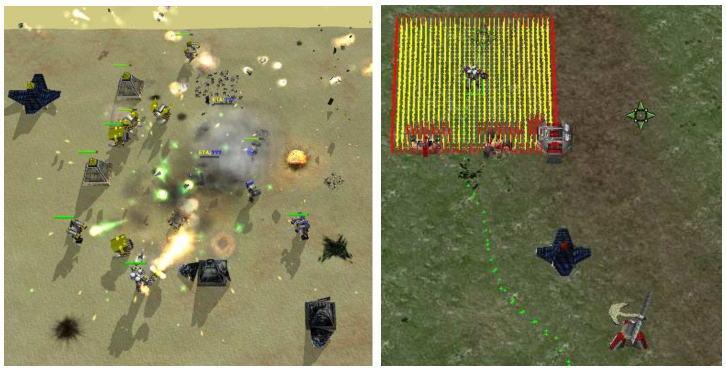
5 Improvements and Speedup using GPU

Solving the Laplace equation is the most expensive part of the BVP Path Planner algorithm. The iterative convergence process of an initial solution to an acceptable solution demands several iterations. So, if we want to improve the convergence speed, we must focus our attention in the relaxation process. We present an approach to improve the convergence on the local maps based on a GPU implementation.

The technique presented here is highly parallelizable, mainly the update of local maps and the computation of Laplace's equation. Each of these steps has several computations to be done, and it can be accomplished in parallel for each agent. We proposed an approach to implement it using nVidia CUDA®and will present it here assuming that the reader knows the CUDA architecture (a detailed explanation can be found in [6]).

Intuitively, each agent detects its observable obstacles in parallel, and each one has its own objective. So, the update of each local map can be also done in parallel. In our algorithm, each agent must seek for obstacles in its own view cone, setting the corresponding cells in the local map as “blocked”. Note that, for a given agent, all other agents are seen as obstacles too. We assume that, in the beginning of this step, all the space occupied by the local map is free of obstacles. Then, for each agent, we launch one thread for each obstacle that the agent can potentially see. In each thread, we check if the obstacle is inside the view cone and inside the local map. If it is, then the thread sets each one of the cells that contains part of the obstacle with a tag “blocked”.

This scheme of launching one thread for each obstacle of each agent fits very well in the nVidia CUDA architecture, where the processing must be split into blocks of threads. Assigning one block for each agent, each thread of the block



(a) (b)

Fig. 4. Two autonomous army fighting (a). The unit local map and the path produced by the BVP Path Planner, illustrated by green dots (b).

can look for one obstacle. Also, each thread can update the local map without synchronization, because all the threads will, if needed, update one cell from a "free" state to a "blocked" state.

After the update of the local maps with the obstacles position, we need to set the intermediate goal. For this, we simply need one thread per agent, each one updating the cells with a "goal" tag. This must be done sequentially to the previous step, to avoid race conditions and the need to synchronize all threads of each block of threads. When each local map has up-to-date information about what cells are occupied, free, or goals, we must relax it to get a smooth potential field. To do this, we assigned one local map to one block of threads in CUDA. In each block of threads, each thread is responsible for updating a value of potential to a single cell. Each thread stays in loop, with one synchronization point between the cells at the end of each iteration. Each thread updates the potential value of the cell using the Jacoby relaxation method.

Finally, to avoid unnecessary memory copies between the GPU and the main memory, we store each attribute of all agents in one single structure of contiguous memory. With this, some parameters that do not change so frequently (e.g. the local map sizes, and the current goal) can be sent only once to the GPU. When the new agent position must be computed, we fetch from the GPU only the current gradient descent on the agent positions.

With our GPU-based strategy we achieved a speed up to 56 times the previous CPU implementation. For a detailed evaluation of our results, refers to [6].

6 A RTS Game Implementation using the BVP Path Planner

In order to demonstrate the applicability of the proposed technique, we implemented the BVP Path Planner in the Spring®Engine for RTS games. Spring is an open source and multi-platform game engine available under the GPLv2 license to develop RTS games. We have chosen this engine since it is well known in the RTS community and there are several commercial games made with it and available for use. With our planner, we can populate a RTS game with hundred of agents at interactive frame rates [6]. Figure 4(a) shows a screenshot of the game where two army are fighting using the BVP Path Planner. Figure 4(b) shows one unit and its local map with 33×33 cells. Red dots represent blocked cells, while the yellow dots represent free cells. The path followed by the unit is illustrated by green dots. An executable demo using the BVP Path Planner implemented with the Spring Engine can be found at `http://www.inf.ufrgs.br/~lgfischer/mig2010`, as well as a video including examples that demonstrate all the techniques presented in this paper. All animations in the video were generated and captured in real time.

7 Conclusion

This paper presented several complementary approaches recently developed by us to produce natural steering behaviors for virtual characters and to allow interaction with the agent navigation. The core of these techniques is the BVP Path Planner [14], that generates potential fields through a differential equation whose gradient descent represents navigation routes. Resulting paths are smooth, free from local minima, and very suitable to be used in RTS games.

Our technique uses a global and a local path planner. The global path planner ensures a path free of local minima, while the local planner is used to control the steering behavior of each agent, handling dynamic obstacles. We demonstrated that our technique can produce natural paths with interesting and complex human-like behaviors to achieve a navigational task, when compared with real paths produced by humans. Dealing with groups of agents, we shown a strategy to handle the path-planning problem while keeping the agent formations. This strategy enables the user to sketch any desirable formation shape. The user can also sketch a path to be followed by agents replacing the global path planner.

We also described a parallel version of this algorithm using the GPU to solve the Laplace's Equation. Finally, to demonstrate the applicability of the proposed technique we implemented the BVP Path Planner in a RTS game engine and released an executable demo available on the Internet.

We are now developing a hierarchical version of the BVP Path Planner that spends less than 1% of the time needed to generate the potential field using our original planner in several environments. We are also exploring strategies to use this planner in very large environments. Finally, we are also working on the generation of benchmarks for our algorithms.

References

1. van den Berg, J., Patil, S., Sewall, J., Manocha, D., Lin, M.: Interactive navigation of multiple agents in crowded environments. In: Proc. of the symposium on Interactive 3D graphics and games. pp. 139–147. ACM (2008)
2. Burgess, R.G., Darken, C.J.: Realistic human path planning using fluid simulation. In: Proc. of Behavior Representation in Modeling and Simulation (BRIMS) (2004)
3. Choi, M.G., Lee, J., Shin, S.Y.: Planning biped locomotion using motion capture data and probabilistic roadmaps. ACM Trans. Graph. 22(2), 182–203 (2003)
4. Dapper, F., Prestes, E., Nedel, L.P.: Generating steering behaviors for virtual humanoids using bvp control. Proc. of CGI 1, 105–114 (2007)
5. Dietrich, C.A., Nedel, L.P., Comba, J.L.D.: A sketch-based interface to real-time strategy games based on a cellular automaton. In: Game programming gems 7. pp. 59–67. Charles River Media, Bostom (2008)
6. Fischer, L.G., Silveira, R., Nedel, L.: Gpu accelerated path-planning for multi-agents in virtual environments. SBGames 0, 101–110 (2009)
7. Funge, J.D.: Artificial Intelligence For Computer Games: An Introduction. A. K. Peters, Ltd., Natick, MA, USA (2004)
8. James J. Kuffner, J.: Goal-directed navigation for animated characters using real-time path planning and control. In: Int. Workshop on Modeling and Motion Capture Techniques for Virtual Environments. pp. 171–186. Springer-Verlag (1998)
9. Kallmann, M.: Shortest paths with arbitrary clearance from navigation meshes. In: Symposium on Computer Animation (SCA) (2010)
10. Kavraki, L., Svestka, P., Latombe, J.C., Overmars, M.: Probabilistic roadmaps for path planning in high-dimensional configuration space. IEEE Trans. on Robotics and Automation 12(4), 566–580 (1996)
11. Metoyer, R.A., Hodgins, J.K.: Reactive pedestrian path following from examples. Visual Comput. 20(10), 635–649 (2004)
12. Nieuwenhuisen, D., Kamphuis, A., Overmars, M.H.: High quality navigation in computer games. Sci. Comput. Program. 67(1), 91–104 (2007)
13. Pettre, J., Simeon, T., Laumond, J.: Planning human walk in virtual environments. In: Int. Conf. on Intelligent Robots and System. vol. 3, pp. 3048 – 3053 (2002)
14. Silveira, R., Dapper, F., Prestes, E., Nedel, L.: Natural steering behaviors for virtual pedestrians. Visual Comput. (2009)
15. Silveira, R., Prestes, E., Nedel, L.P.: Managing coherent groups. Comput. Animat. Virtual Worlds 19(3-4), 295–305 (2008)
16. Tecchia, F., Loscos, C., Conroy, R., Chrysanthou, Y.: Agent behaviour simulator (abs): A platform for urban behaviour development. In: Proc. Game Technology, 2001 (2001)
17. Treuille, A., Cooper, S., Popović, Z.: Continuum crowds. In: ACM SIGGRAPH. pp. 1160–1168. ACM Press, New York, NY, USA (2006)
18. Trevisan, M., Idiart, M.A.P., Prestes, E., Engel, P.M.: Exploratory navigation based on dynamic boundary value problems. J. Intell. Robot. Syst. 45(2), 101–114 (2006)

GPU Accelerated Path-planning for Multi-agents in Virtual Environments

Leonardo G. Fischer, Renato Silveira, Luciana Nedel
Institute of Informatics
Federal University of Rio Grande do Sul
Porto Alegre, Brazil
{*lgfischer,rsilveira,nedel*}@*inf.ufrgs.br*

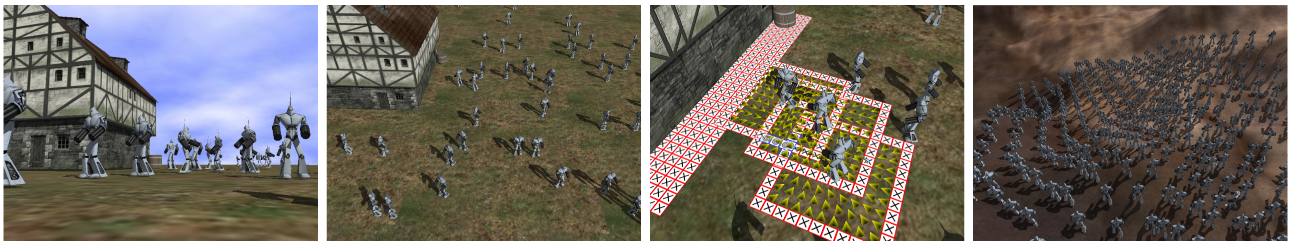


Figure 1. Virtual characters controlled by the BVP Planner in a virtual environment.

***Abstract*—Many games are populated by synthetic humanoid actors that act as autonomous agents. The animation of humanoids in real-time applications is yet a challenge if the problem involves attaining a precise location in a virtual world (path-planning), and moving realistically according to its own personality, intentions and mood (motion planning). In this paper we present a strategy to implement – using CUDA on GPU – a path planner that produces natural steering behaviors for virtual humans using a numerical solution for boundary value problems. The planner is based on the potential field formalism that allows synthetic actors to move negotiating space, avoiding collisions, and attaining goals, while producing very individual paths. The individuality of each character can be set by changing its inner field parameters leading to a broad range of possible behaviors without jeopardizing its performance. With our GPU-based strategy we achieve a speed up to 56 times the previous implementation, allowing its use in situations with a large number of autonomous characters, which is commonly found in games.**

***Keywords*-Path-planning; GPGPU; NVIDIA CUDA; Agent Simulation**

I. Introduction

Many types of games, specifically First Person Shooters (*FPS*) and Real Time Strategy (*RTS*) are populated by synthetic actors that should act as autonomous agents. Autonomous agents, also called *non-player characters*, are characters with the ability of playing a role into the environment with life-like and improvisational behavior. To behave in such way, the agents must act in the virtual world, perceive, react and remember their perceptions about this world, think about the effects of possible actions and finally, learn from their experience [1]. In this complex and suitable context, navigation plays an important role [2]. To move agents in a synthetic world, a semantic representation of the environment is needed, as well as the definition of the agent initial and target position (goal). Once these parameters were set, any path-planning algorithm can be used to find a trajectory to be followed.

However, in the real world, if we consider different persons (all in the same initial position) looking for achieving the same target position, each path followed will be unique. Even for the same task, the strategy used for each person to reach his/her goal depends on his/her physical constitution, personality, mood, reasoning, urgency, and so on. From this point of view, a high quality algorithm to move characters across virtual environments should generate expressive, natural and unexpected steering behaviors.

In contrast, the high performance required for real-time graphics applications compels developers to look for most efficient and less expensive methods that produce yet good and almost natural movements. To illustrate how performance is a crucial problem, it is known that to be playable, a game must run at least at a rate of 30-100 frames per second. This implies in 0.02 seconds per frame. Each frame (or step of an animation) includes the updating of the game status, handling user inputs, graphics processing, physics computations, strategic AI, path-planning, among others. Then, we can easily consider something as one millisecond per step for path-planning (with multi-core architectures, this restriction is relaxed).

Many researchers are working on methods to improve the quality of the steering behavior of synthetic agents with a minimal cost. One way to improve the performance is taking advantage of massively parallel architectures, as multi-core CPUs and GPUs (*Graphics Processing Unit*). In this work we propose a GPU implementation of the BVP Planner recently proposed by us [3]. The BVP Planner is a method based on the numeric solution of the boundary value problem (BVP) to control the movement of pedestrians allowing the individuality of each agent.

Our main contributions in this paper are:

- A parallel version of our previously technique [3], implemented on the GPU using nVIDIA CUDA (Compute Unified Device Architecture) [4]
- A strategy to reduce the number of memory transactions between CPU and GPU
- Several tests showing that the GPU implementation improves up to 56 times the CPU sequential version, allowing the real-time use of this technique even in scenarios with a large number of autonomous characters

Despite *humanoid*, *autonomous agent*, and *behavior* are terms used in many different contexts, in this paper we limit its use in order to match our goals. For the sake of simplicity, we consider *humanoids* as a kind of embodied *autonomous agent* with reactive behaviors (driven by stimulus), represented by a computational model, and capable of producing physical manifestations in a virtual world. The term *behavior* will be used mainly as a synonymous of *animation* or *steering behavior* and intend to refer the improvisational and personalized action of a *humanoid*.

The remaining of this paper is structured as follows. Section II reviews some related works on path-planning techniques applied to virtual agents simulation. Section III describes the fundamentals of the path-planning method proposed by us. In Section IV we detail the strategy used to handle the information about the environment and other agents. In Section V we present our strategy to implement this technique on GPU. Section VI shows our results, including several comparisons between the CPU and GPU version, and exposes considerations about performance. Finally, Section VII presents our conclusions and some ideas for future works.

II. Related Work

The path-planning problem has been deeply explored in game development. The generation of a path between two known configurations in a bi-dimensional world is a well-known problem in robotics, artificial intelligence, and computer graphics field. However, to find the path is not enough when we want to endow artificial characters with natural and realistic movement similar to the ones found and followed by real human beings. When it comes to a game with many autonomous characters, for instance, these characters must also present convincing behavior. It is very difficult to produce natural behavior by using a strategy focusing on the global control of characters. On the other hand, taking into account the individuality of each character can be a costly task. As a consequence, most of the approaches proposed in computer graphics literature do not take into account the individual behavior of each agent.

An example is the technique proposed by Kuffner [5]. Kuffner proposed a technique where the scenario is mapped onto a 2D mesh and the path is computed using a dynamic programming technique like Dijkstra. Then, the motion controller is used to animate the agent along the path planned. Kuffner argue that his technique is fast enough to be used in dynamic environments. Another example is the work developed by Metoyer and Hodgings [6]. They proposed a technique where the user defines the path that should be followed by each agent. During the motion along this path, it is smoothed and slightly changed to avoid collisions using force fields that act on the agent.

The development of randomized path-finding algorithms – specially the PRM (Probabilistic Roadmaps) [7] and RTT (Rapidly-exploring Random Tree) [8] – allowed the use of large and more complex configuration spaces to generate paths efficiently. Thus, the challenge becomes more the generation of realistic movements than finding a valid path. For instance, Choi et al. [9] use a library of captured movements associated to the PRM to generate realistic movements in a static environment, that is, live-captured motions are used insofar the agent tracks the path computed from a roadmap. Despite the fact the path is computed in a pre-processing phase, results are very realistic. Pettré et al. [10] improved this idea adding one more step in this process. This step consists of smoothing the path computed by the PRM using Bézier curves. Hereinafter, the already captured motions are associated to the agent position during the path execution. As in previous works, the motion is also performed on a 2D environment.

Differently, Burgess and Darken [11] proposed a method based on the *principle of least action* which describes the tendency of elements in nature to seek the minimal effort solution. Authors claim that a realistic path for a human is the one that requires the smallest amount of effort. The method produces human-like movements, through very realistic paths, using properties of fluid simulation.

Tecchia et al. [12] proposed a platform that aims to accelerate the development of behaviors for agents through local rules that control these behaviors. These rules are governed by four different control levels, where each one reflects a different aspect of the behavior of the agent. Results show that, for a fairly simple behavioral model, the system performance can achieve interactive time.

Pelechano et al. [13] described a new architecture to integrate a psychological model into a crowd simulation system in order to obtain believable emergent behaviors. The architecture achieves individualistic behaviors through the modeling of the agent knowledge, as well as the basic principles of communication between agents.

Treuille et al. [14] proposed a crowd simulator driven by dynamic potential fields which integrates both global navigation and local collision avoidance. Basically, this technique uses the crowd as a density field, and, for each group, constructs a unit cost field which is used to control people displacement. The method produces smooth behavior for a large amount of agents at interactive rates.

Recently, Reynolds [15] implemented a high performance multi-agent simulation and animation for the Playstation® 3. Basically, his technique uses a spatial partitioning that divides the simulation into disjoint jobs which are evaluated in an arbitrary order on any number of Playstation® 3 Synergistic Processor Units (SPUs). A fine-grain partitioning suits SPU memory size and provides automatic load balancing. This approach allows a scalable multi-processor implementation of a large and fast crowd simulation, achieving good frame rates with thousand of agents.

In 2008, Bleiweiss [16] implemented the Dijkstra and the A* algorithms using CUDA. Differently from our work, these algorithms are used in the path finding problem with pre-computed graphs. After several benchmarks, he observed that the Dijkstra implementation reached a speed up of 27 times compared to a C++ implementation without SSE instructions. The A* implementation reached a speed up of 24 times compared to the C++ implementation with SSE instructions.

Based on local control, van den Berg [17] proposed a technique that handles the navigation of multiple agents in the presence of dynamic obstacles. He uses an extended *velocity obstacles* concept to locally control the agents with few oscillation. Kapadia [18] presented a framework that enables agents to navigate in unknown environments based on *affordance fields* that compute all the possible ways an agent can interact with its environment.

As mentioned above, most of the approaches do not take into account the individual behavior of each agent, his internal state or mood. Our assumption is that realistic paths derive from human personal characteristics and internal state, thus varying from one person to another. As a consequence, we [3], [19] recently proposed a technique that generate individual paths. Our path is smooth and is dynamically generated while the agent walks. In the following sections, we will explain the concepts of our technique and our strategy to implement it on the GPU.

III. Path Planner based on Boundary Value Problems

Recently, we [3], [19] developed a technique that produces natural and individual behaviors for virtual humanoids. This technique is based on an extension of the Laplace's Equation that produces a family of potential field functions that do not have local minima. This family is generated through the numeric solution of a convenient partial differential equation with Dirichlet boundary conditions, i.e., a boundary value problem (BVP). Boundary conditions are central to the method indicating which regions in the environment are obstacles and which ones are targets. Our method uses the following equation

$$\nabla^2 p(\mathbf{r}) + \epsilon \mathbf{v}.\nabla p(\mathbf{r}) = 0 \qquad (1)$$

where $\mathbf{v}$ is a bias unity vector and ϵ is a scalar value.

The use of terms ϵ and $\mathbf{v}$ distort the potential field providing a preferred direction to be followed. This distortion allows the production of individual behaviors for humanoids illustrated through the path followed by each one during navigation tasks.

To generate realistic steering behaviors, we need to conveniently adjust both parameters ϵ and $\mathbf{v}$. The vector $\mathbf{v}$, called *behavior vector*, can be thought as an external force that pulls the agent to its direction always as possible whereas the parameter ϵ can be understood as the *strength* or *influence* of this vector in the agent behavior. The allowed values of parameters ϵ and $\mathbf{v}$ permit to generate an expressive amount of action sequences – *displacement sequences* – that virtual humanoids can use to reach a specific target position. Figure 2 shows three different paths followed by an agent using the Equation 1 and changing the parameters ϵ and $\mathbf{v}$.

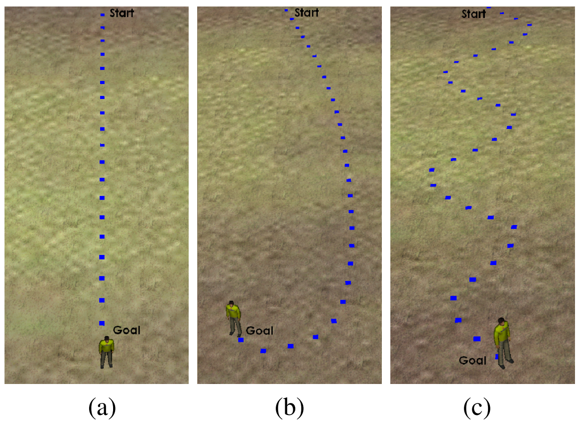


Figure 2. Different paths followed by an agent using Equation 1: (a) path produced by harmonic potential, i.e., with $\epsilon = 0$; (b) with $\epsilon = -1.0$ and $\mathbf{v} = (1, 0)$; (c) with $\epsilon = -1.0$ and $\mathbf{v} = (1, sin(0.6t))$

Two action sequences are not statically defined for a same pair ϵ and $\mathbf{v}$, i.e., the path generated vary according to the information gathered by the agent to allow it to dynamically react against unexpected events (e.g. dynamic obstacles). In other words, the configuration of the obstacles has an important role in the generation of the path.

Besides, this pair is not constrained to keep constant during the execution of tasks. They can vary insofar the agent displaces in the environment to obtain the desired behavior. Figure 2(c) shows a situation where the behavior vector varies according to a sin function. It is not natural for human beings to walk based on a sin function. However, the path based on a sin function illustrates the flexibility of Equation 1. Any function can be associated to $\mathbf{v}$ and ϵ to generate a behavior.

When $\epsilon = 0$, Equation 1 reduces to $\nabla^2\, p(\mathbf{r}) = 0$ which corresponds to Laplace's Equation. This equation is used as core of the path planner based on harmonic function developed by Connolly and Grupen [20] on Robotics context. This planner produces paths that minimize the hitting probability of the agent with obstacles, i.e., in an indoor environment the agent will tend to follows a path equidistant to the walls, as shown in Figure 2(a). This behavior is not always adequate to simulate humanoid motion since it looks very stereotyped because humans do not always walk equidistant to the walls. Hence the importance of using these parameters ϵ and $\mathbf{v}$.

The common approach to numerically solve a BVP is to consider that the solution space is discretized in a regular grid. Each cell (i, j) is associated to a squared region of the real environment and stores a potential value $p^t_{i,j}$ at instant t. Each cell is distant from each other 1 unit. The Dirichlet boundary conditions previously associate a specific potential value to some cells, before the relaxation process is performed. That is, cells associated to obstacles in the real environment store a potential value equal to 1 (*high potential*) whereas cells containing the target store a potential value equal to 0 (*low potential*). The high potential value prevents the agent from running into obstacles whereas the low potential value generates an attraction basin that pulls the agent. The potentials of the other cells are computed using the Gauss-Seidel relaxation method, as discussed in [21]. By considering the Equation 1, the potentials of the free space cells are updated through the following equation

$$p_c = \frac{p_b + p_t + p_r + p_l}{4} + \frac{\epsilon((p_r - p_l)v_x + (p_b - p_t)v_y)}{8} \qquad (2)$$

where $p_c = p^{t+1}_{i,j}$, $p_b = p^t_{i,j+1}$, $p_t = p^{t+1}_{i,j-1}$, $p_r = p^t_{i+1,j}$, $p_l = p^{t+1}_{i-1,j}$ and $\mathbf{v} = (v_x, v_y)$. Figure 3 shows a representation of these cells.

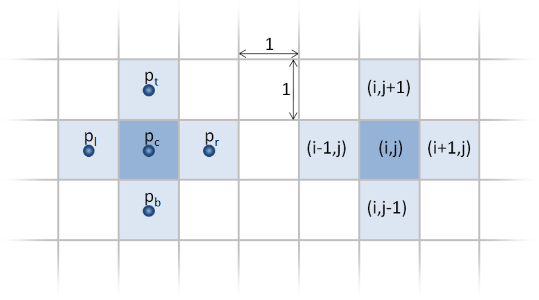


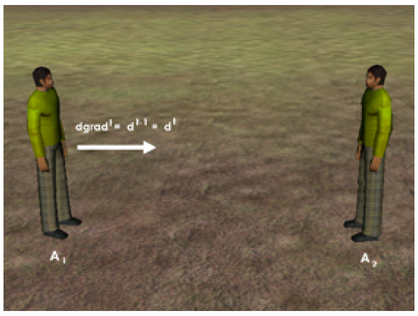
Figure 3. Representation of p_c, p_b, p_t, p_r and p_l on the grid.

The parameter $\mathbf{v}$ must be a unit vector and ϵ must be in the interval $(-2, 2)$. Values out of this range generate oscillatory and unstable paths that do not guarantee that the agent will reach the target or will avoid obstacles. This happens because the boundary conditions – that assert the agent is repelled by obstacles and attracted by targets – are violated.

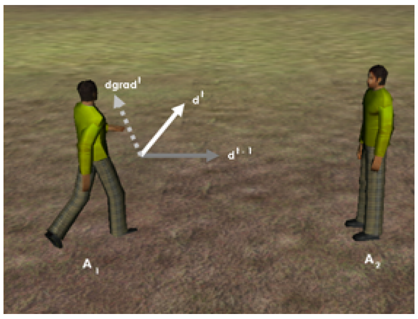
After the potential computation, the agent moves following the direction of the gradient descent of this potential at its current position (i, j),

$$(\nabla \mathbf{p})_{(i,j)} = \left(\frac{p_{i+1,j} - p_{i-1,j}}{2}, \frac{p_{i,j+1} - p_{i,j-1}}{2} \right)$$

This process is an intuitive way to control the agent motion. However, it can easily fail in producing realistic steering behaviors, as observed in real world. One of the reasons is that the agent changes its direction based solely on the gradient descent of its position. For instance, if the field of view of the agent is small, its reaction time will be very short to treat dynamic obstacles[1]. Then, these obstacles will produce a strong repel force that will change the agent direction abruptly. As we can see in Figure 4, if the agent uses only the gradient descent (*dgrad*) it will change its direction in nearly $\pi/2$.



(a)



(b)

Figure 4. Defining agent motion. (a) Situation before the agent A_2 enters in the field of view of A_1. (b) If the agent A_1 follows the direction defined by the gradient descent (*dgrad*), it will changes its direction in nearly $\pi/2$, what is undesirable. However, if the agent uses the vector d, it will achieve a smooth curve, what is more natural and realistic.

We handle this problem by adjusting the current agent position by

$$\Delta\, \mathbf{d} = v(\cos(\varphi^t), \sin(\varphi^t)) \qquad (3)$$

[1]We consider that dynamic obstacles (as other agents) are mapped in the environment only when they are inside the field of view of the agent, which almost corresponds to reality.

where υ defines the maximum agent speed and φ^t is

$$\varphi^t = \eta\, \varphi^{t-1} + (1-\eta)\, \zeta^t \tag{4}$$

where $\eta \in [0,1)$ and ζ is the orientation of the gradient descent at current agent position.

When $\eta = 0$, the agent adjusts its orientation using only information about the gradient descent. If $\eta = 0.5$, the previous agent direction (φ^{t-1}) and the gradient descent direction influence equally the computation of the new agent direction. Figure 4(b) shows the vector $\mathbf{d}^t$ with orientation φ^t computed with $\eta = 0.5$. The parameter η can be viewed as an inertial factor that tends to keep the agent direction constant insofar $\eta \to 1$. When $\eta \to 1$, the agent reacts slowly to unexpected events, increasing its hitting probability with obstacles. η is a flexible parameter that the user is able to control. However, a learning strategy could be used to specify what is the best η to a specific situation.

Despite Equation 3 produces good results and smooth paths in environments with few obstacles, when the environment is cluttered with obstacles, the agent behavior is not realistic and collisions can happen. To solve this problem, a speed control was incorporated into this equation,

$$\Delta\, \mathbf{d} = \upsilon\, (\cos(\varphi^t), \sin(\varphi^t))\; \Psi(|\varphi^{t-1} - \zeta^t|) \tag{5}$$

where function $\Psi : \mathbf{R} \to \mathbf{R}$ is

$$\Psi(x) = \begin{cases} 0 & \text{if } x > \pi/2 \\ \cos(x) & \text{, otherwise} \end{cases} .$$

If $|\varphi^{t-1} - \zeta^t|$ is higher than $\pi/2$, then there is a high hitting probability and this function returns the value 0, making the agent stops. Otherwise, the agent speed will change proportionally to the collision risk. In regions cluttered with obstacles, agents will tend to move slowly. If a given agent is about to cross the path of another, one of them will stop and wait until the other get through. Furthermore, speed control allows the simulation of agents' mood through the variation of the speed magnitude, that is, it is possible to simulate a tired agent making it move slower and an agent that is anxious about its work making it move faster.

IV. Implementation Strategy

As previously explained, our motion planning method requires the discretization of the environment into a regular grid. In this section we present the strategy that was used in our previous work [3], [19] to implement it by using global environment maps (one for each target) and local maps (one for each agent), as well as the mechanisms used to control each agent steering behavior.

A. Environment Global Map

The entire environment is represented by a set of homogeneous meshes, $\{\mathcal{M}_k\}$, in which each mesh $\mathcal{M}_k$ has $L_x \times L_y$ cells, denoted by $\{\mathcal{C}^k_{i,j}\}$. Each cell $\mathcal{C}^k_{i,j}$ corresponds to a squared region centered in environment coordinates $r = (r_i, r_j)$ and stores a particular potential value $\mathcal{P}^k_{i,j}$. The potential associated to the mesh $\mathcal{M}_k$ is computed by the harmonic path planner, through the Equation 2, and then used by agents to reach the target $\mathcal{O}_k$.

In order to delimit the navigation space, we consider that the environment is surrounded by static obstacles. Global maps are built before simulation starts, in a pre-processing phase.

B. Agent Local Map

Each agent a_k has one map m_k that stores the current local information about the environment obtained by its own sensors. This map is centered in the current agent position and represents a small fraction of the global map, usually about 10% of the total area covered by the global map.

The map m_k has $l^k_x \times l^k_y$ cells, denoted by $\{c^k_{i,j}\}$ and divided in three regions: the update zone (*u-zone*); the free zone (*f-zone*) and the border zone (*b-zone*), as shown in Figure 5. Each cell corresponds to a squared region centered in environment coordinates $r = (r_i, r_j)$ and stores a particular potential value $p^k_{i,j}$.

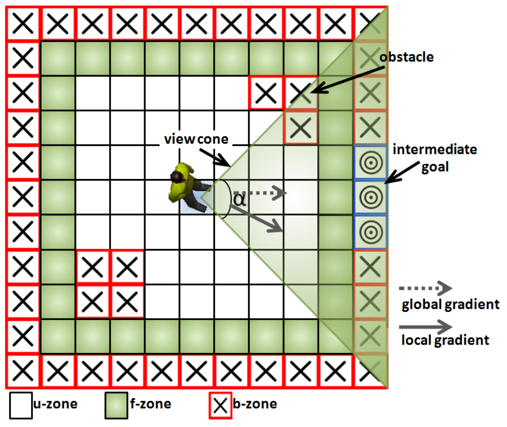


Figure 5. Agent Local Map. The update (u-zone), free (f-zone) and border zones (b-zone) are shown. Blue and red cells correspond to the intermediate goal and obstacles, respectively.

The area associated to each agent map cell is smaller than the area associated to the global map cell. The main reason is that the agent map is used to produce refined motion, while the global map is used only to assist the long-term agent navigation. Hence, the smaller the size of the cell on the local map, the better the quality of motion.

C. Updating Local Maps from Global Maps

For each agent a_k, a goal $\mathcal{O}_{goal(k)}$[2], a particular vector $\mathbf{v}_k$ that controls its behavior, and a ϵ_k should be stated. The same goal, $\mathbf{v}$, and ϵ can be designated to several agents. If $\mathbf{v}_k$ or ϵ_k is dynamic, then the function that controls it must also be specified.

[2]Function *goal()* maps the agent number k into its current target number

To navigate into the environment, an agent a_k uses its sensors to perceive the world and to update its local map with information about obstacles and other agents. The agent sensor sets a view cone with aperture α.

Figure 6 exemplifies a particular instance of the agent local map where we can see the obstacles mapped from the global map. The *u-zone* cells $c_{i,j}^{k}$ which are inside the view cone and correspond to obstacles or other agents have their potential value set to 1. In Figure 7, as there is an agent in the *u-zone* of the agent local map, inside of his view cone, it is mapped as an obstacle into his local map. This procedure assures that dynamic or static obstacles behind the agent (out of his view cone) do not interfere in his future motion.

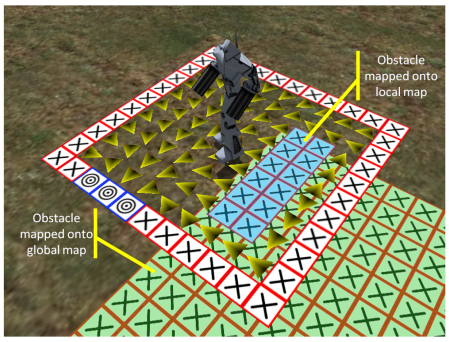

Obstacle mapped onto local map
Obstacle mapped onto global map

Figure 6. Global map mapped onto the agent local map.

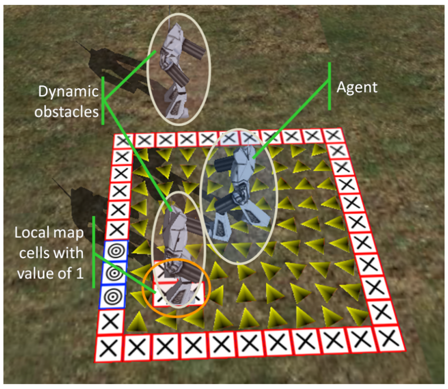


Figure 7. The cells which are inside the agent's view cone and correspond to obstacles or other agents have their potential value set to 1.

For each agent a_k, the global descent gradient on the cell in the global map $\mathcal{M}_{goal(k)}$ that contains his current position is calculated. The gradient direction is used to generate an intermediate goal in the border of the local map, setting the potential values of a couple of *b-zone* cells to 0, while the other *b-zone* cells are considered as obstacles, with their potential values set to 1. In Figure 7, the agent calculates his global gradient in order to project an intermediate goal in its own local map. As the agent local map is delimited by obstacles, the agent is pulled towards the intermediate goal using the direction of his local gradient. The intermediate goal helps the agent a_k to reach its target $\mathcal{O}_{goal(k)}$ while allowing it to produce a particular motion.

In some cases, the target $\mathcal{O}_{goal(k)}$ is inside both view cone and *u-zone*, and consequently local map cells associated are set to 0. The intermediate goal is always projected, even if the target is mapped onto the *u-zone*. Otherwise the agent can easily get trapped because it would be taking into consideration only the local information about the environment, in a same way as traditional potential fields [22].

F-zone cells are always considered free of obstacles, even when there are obstacles inside. The absence of this zone may close the connection between the current agent cell and the intermediate goal due to the mapping of obstacles in front of the intermediate goal. When this occurs, the agent gets lost because there is no information coming from the intermediate goal to produce a path to reach it. *F-zone* cells handle the situation always allowing the propagation of the goal's information to the cells associated to the agent position.

After the sensing and mapping steps, the agent k updates the potential value of its map cells using Equation 2 with its pair $\mathbf{v}^k$ and ϵ^k. Hereinafter, it updates its position according to Equation 5 using the gradient descent computed from the potential field stored on its local map in the position $p_x = \lceil l_x^k/2 \rceil$ and $p_y = \lceil l_y^k/2 \rceil$.

V. Implementation on GPU

In the real world, people walking inside a room react to what they perceive from the environment based on their own personality, mood and reasoning, i.e., they think in parallel. So, a technique that handles several agents should be parallelized in the same way.

According to Section III, during the update phase of our technique, each agent must update its local map with the environment obstacles which are inside this region. Note that, in this step, we consider that for a given agent a_i, each other agent $a_j, i \neq j$, is also an obstacle. Then, each cell in the agent local map inside his view cone is updated as an obstacle, with the potential value equal to 1. After the update of these cells, we update the cells which correspond to the agent goal, with the potential value of 0.

Note that each one of these updates can be made in parallel between the agents. The only dependency here is that obstacle cells must be updated before goal cells. It must be done sequentially, otherwise, if an agent has a goal very close to an obstacle, both obstacle and goal may be mapped to the same cell. In this case, if goal cells are updated before obstacle cells, the agent will become lost, without a goal to achieve. All other cells are updated as free cells.

Afterwards, the Equation 2 is evaluated for each agent local map. Since it is difficult and needs to be evaluated independently for each agent, it is a good candidate for a parallel implementation. The Gauss-Seidel relaxation method (previously used in Equation 2) is not suitable for a parallel implementation because it uses values from the current and previous iterations. In a sequential approach, it is very simple to implement and fast to execute, but a parallel implementation will require some kind of synchronization, which may cause degradation in performance. A better approach for a parallel implementation is to use values only from the previous iteration. This is exactly what the Jacobi method does. The update rule is described below.

$$p_c = \frac{p_b + p_t + p_r + p_l}{4} + \frac{\epsilon((p_r - p_l)v_x + (p_b - p_t)v_y)}{8} \quad (6)$$

where $p_c = p^t_{i,j}$, $p_b = p^t_{i,j+1}$, $p_t = p^t_{i,j-1}$, $p_r = p^t_{i+1,j}$, $p_l = p^t_{i-1,j}$ and $\mathbf{v} = (v_x, v_y)$.

We implemented the parallel version of our technique using the nVIDIA® Cuda [4] language, which allows us to use the graphics processor without using shading languages. In the context of CUDA, the CPU, here called *Host*, controls the graphics processor, called *Device*. It sends data, calls the *Device* to execute some functions, and then copies back its results.

Each graphics processor of a nVIDIA graphics card is divided into several multiprocessors. Cuda divides the processing in blocks, where each block is divided in several threads. Each block of threads is mapped to one multiprocessor of the graphics processor. When the *Host* calls the *Device* to execute a function, it needs to inform how the work will be divided in blocks and threads. Maximum performance is achieved when we maximize the use of blocks and threads for a given graphics processor.

Each of the multiprocessors is a group of simple processors that share a set of registers and some memory (the *shared memory* space). The shared memory size is very small (16KB on graphics cards up to *Compute Capability 1.3*), but it is as fast as the registers. The communication between two multiprocessors must be done through the Device Memory, which is very slow if compared to the shared memory. There is also the *Constant Cache* and *Texture Cache memory*, which has better access times than the Device memory, but it is read-only for the *Device*.

Before the execution of the code in the *Device*, the *Host* must send the data to its Device Memory to be processed later. The memory copy from the Host Memory to the Device memory is a slow process, and should be minimized. Besides, the nVIDIA Cuda Programming Guide [4] says that one single call to the memory copy function with a lot of data is much more efficient than several calls to the same function with a few bytes. We can improve the performance of our application making good use of these restrictions of Cuda.

As previously mentioned, each agent a_k has several attributes: the scalar ϵ_k, the vector $\mathbf{v}_k$, and its current objective $\mathcal{O}_{goal(k)}$. The local map also has some attributes, like its width l^k_x and height l^k_y. All these attributes must be sent at least once to the *Device*. The agent goal and the local map position in the world, for instance, will be frequently updated. To avoid several memory transactions between the *Host* and the *Device*, we store all these attributes in contiguous memory areas, and treat it like an array. At the position k we store an attribute of the agent a_k. Proceeding this way, we avoid several unnecessary copies, improving the overall performance.

Figure 8 shows our data structure for a set of 3 agents. The array m with all local map cells is illustrated with its cell's index. Each position k of the array **s** contains an index to the first position in the array **m** in which the agent a_k local map information is stored. Each position k of the array **l**, **O**, ϵ, **v** contains the information of the local map dimension and goal, as well as the behavioral parameters ϵ and **v** of the agent a_k, respectively.

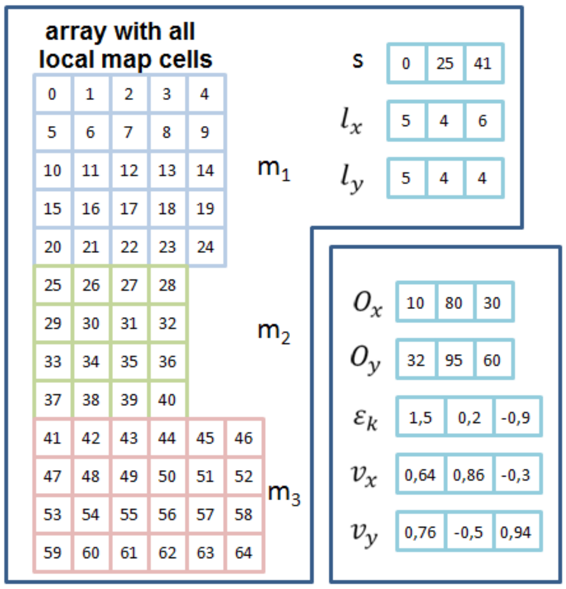


Figure 8. Data structure used on GPU.

There are situations in which the size of the agent local map must be changed. Any update on the size of an agent local map will require the modification of the array m, which implies in the entire data structure reconstruction. In these cases, the *Host* must reallocate the entire array in the Host Memory, and send it again to the Device Memory. These attributes should not only be copied once to the Device memory, but they should be sent to the Constant Memory or Texture Memory.

As the *Environment Global Map* is composed only of static obstacles, it can be copied to the Device Memory only once. Then, the update step can be done in the following way. First, each local map is mapped into a block of threads, in which each thread updates one cell of the local map. The thread will find the local map cell corresponding to the Environment Global Map, and will copy the information from the global map cell to the local map cell. This is done only to the cells in the *f-zone*. Figure 6 illustrates this situation.

Afterwards, each local map is mapped to a block of threads, and each thread is associated with a dynamic obstacle. This thread checks whether the obstacle appears inside the view cone. If yes, the local map cells occupied by the obstacle update its potential value to 1. Next, each cell in the *b-zone* is mapped as an obstacle, also updating its potential to 1, except for the ones that are goal cells. The remaining cells are updated as free cells. Then, the Equation 6 can be evaluated, starting one thread to each local map cell. A synchronization must be made between the iterations in order to guarantee that all cells are up to date to the next iteration.

The convergence of the Equation 6 is achieved through several *reads* and *writes* at the Device Memory during several iterations. In order to avoid the high latency of the Device Memory, this must be made in the shared memory of the multiprocessor. An implementation of the Jacobi method will require two copies of the potential map, where at each iteration the values are *read* from one of them and *written* to the other. However, the shared memory size is very limited. Then, we decided to use a combination of the Jacobi method with the Gauss-Seidel. In our implementation, only one copy of the potential map is stored in the shared memory. At each iteration t, a cell $c^k_{i,j}$ may be updated with the potential of the neighborhood cells at the iteration $t-1$ or t. We do not specify whether will be used values from iteration $t-1$ or t. It will depend on how the information will be arranged in the shaders, i.e., the synchronization between cells update is not needed.

VI. Results

In order to verify that our parallel implementation can be executed faster than the sequential one, a couple of tests were accomplished. All the tests were executed in an Intel® Core 2 6300 1.86GHz, with 2Gb of RAM memory, a nVIDIA GeForce 9800 GX2 graphics card (the graphics processor has 600 MHz of clock) and Microsoft Windows XP SP3 operating system. We measured how many times per second the algorithm can be executed, and what is the impact of the memory copy between the *Host* and the *Device*, using three different sizes for the local maps.

The tests were executed in the following way. Initially, three sizes of local maps where chosen: 11×11, 16×16 and 21×21. We chose these sizes because previous tests [19] showed that they generate animations with very good quality, being the most interesting for tests. Then, several scenarios were executed using the parallel and sequential versions of the algorithm, changing the number of agents in the scene. For each test, we recorded the frequency at which the algorithm can be executed, and the percentage of time spent in memory copies between the *Host* and the *Device*.

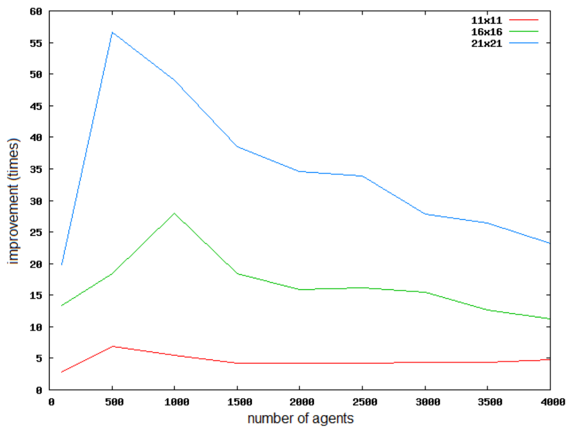


Figure 9. Speed up achieved using the parallel implementation over the sequential version, with three different sizes of local maps.

The graphic in Figure 9 shows the speed up achieved using the parallel implementation over the sequential version of the technique. As we can see, in all tests executed the parallel version was above twice faster than the sequential one (exactly the lowest point in the graphic is at 2.85 times). Besides that, the highest point in the graphic occurs at the point 56.60, meaning that in an optimal configuration the parallel version was 56 times faster than the sequential version.

Using bigger local maps means that more threads are needed for each local map in the several steps of the technique. The fact that the multiprocessor offers several running threads at the same time implies in a better use of the resources and in good improvements in performance.

On the other hand, for several reasons, with smaller local maps the speed up is not so high. On the side of the parallel version, a small local map does not make a good use of the resources of each multiprocessor. And on the side of the sequential version, a small local map may fit better in the processor cache. Moreover, the processor clock is three times higher than the graphics processor clock. If we combine all these factors in the same test, the speed up in the parallel version is minimized.

In addition, according to the nVIDIA Cuda Programming Guide [4], the graphics processor cannot handle all the data in a parallel way. The division of the work in blocks of threads lets the graphics processor scheduler run some blocks of thread while others wait for execution. Because of this, the computation of 256 local maps in a parallel way does not give a speed up of 256 times.

To explain what is the cause of the graphics peaks, the nVIDIA Cuda Programming Guide says that each algorithm implemented with Cuda has an optimal point, in which the amount of blocks and threads uses the most possible number of resources available in the graphics processor simultaneously. In our technique, this point is the one with

500 agents in the scene, each one with a local map of a size of 21×21.

VII. Conclusion

This paper presented a strategy to implement on GPU a BVP Planner [3] that produces natural steering behaviors for virtual humans, using a path-planning algorithm based on the numerical solution of boundary value problems.

The guiding potential of Equation 1 is free of local minima, what constitutes a great advantage when compared to the traditional potential fields method. Furthermore, the method proposed is formally complete [20] and generates smooth and safe paths that can be directly used in mobile robots or autonomous characters in games. The local information gathered by agent sensors allows treating dynamic obstacles, such as other agents navigating in the environment.

We implemented a parallel version of this algorithm using the nVIDIA® Cuda [4] language, which allows us to use the graphics processor avoiding the use of shading languages. The parallelism was explored, reducing the amount of memory transactions between CPU and GPU.

Our result shown that the GPU implementation improves up to 56 times the sequential CPU version, allowing the real-time use of this technique even in scenarios with a huge number of autonomous characters, which is a common situation often found in games.

As future work, we suggest the exploration of ADI Method [23], obtaining a faster convergence of the relaxation process. The ADI Method is suitable to be used on parallel architectures and to explore the use of other shading languages. It would be interesting to compare the possible improvements in performance using other languages.

We have also proposed an extension of this technique to manage the movement of groups of agents in dynamic environments [24]. We intend to implement a parallel version of this extension and release the project over an open source license.

Acknowledgements

The authors would like to thank Edson Prestes and Fabio Dapper for their valuable work on the CPU version of the path-planning algorithm, and Francele Carvalho Rodrigues for helping in the orthographic and grammatical revision of the text. This work was partially supported by grants from CNPq to Leonardo Fischer, Renato Silveira and Luciana Nedel.

References

[1] J. D. Funge, *Artificial Intelligence For Computer Games: An Introduction*. Natick, MA, USA: A. K. Peters, Ltd., 2004.

[2] D. Nieuwenhuisen, A. Kamphuis, and M. H. Overmars, "High quality navigation in computer games," *Sci. Comput. Program.*, vol. 67, no. 1, pp. 91–104, 2007.

[3] F. Dapper, E. Prestes, and L. P. Nedel, "Generating steering behaviors for virtual humanoids using bvp control," *Proc. of CGI*, 2007.

[4] NVIDIA., "Nvidia cuda," *http://www.nvidia.com/cuda, last acces at 07/2009*, 2009.

[5] J. James J. Kuffner, "Goal-directed navigation for animated characters using real-time path planning and control," in *International Workshop on Modelling and Motion Capture Techniques for Virtual Environments*. London, UK: Springer-Verlag, 1998, pp. 171–186.

[6] R. A. Metoyer and J. K. Hodgins, "Reactive pedestrian path following from examples," *The Visual Computer*, vol. 20, no. 10, pp. 635–649, 2004.

[7] L. Kavraki, P. Svestka, J. C. Latombe, and M. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration space," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, 1996.

[8] S. LaValle, "Rapidly-exploring random trees: A new tool for path planning," Computer Science Dept., Iowa State University, Tech. Rep. 98-11, 1998.

[9] M. G. Choi, J. Lee, and S. Y. Shin, "Planning biped locomotion using motion capture data and probabilistic roadmaps," *ACM Trans. Graph.*, vol. 22, no. 2, pp. 182–203, 2003.

[10] J. Pettre, T. Simeon, and J. Laumond, "Planning human walk in virtual environments," in *IEEE/RSJ International Conference on Intelligent Robots and System*, vol. 3, 2002, pp. 3048 – 3053.

[11] R. G. Burgess and C. J. Darken, "Realistic human path planning using fluid simulation," in *Proceedings of Behavior Representation in Modeling and Simulation (BRIMS)*, 2004.

[12] F. Tecchia, C. Loscos, R. Conroy, and Y. Chrysanthou, "Agent behaviour simulator (abs): A platform for urban behaviour development," 2001. [Online]. Available: citeseer.ist.psu.edu/tecchia01agent.html

[13] N. Pelechano, K. Obrien, B. Silverman, and N. Badler, "Crowd simulation incorporating agent psychological models, roles and communication," in *1st Int'l Workshop on Crowd Simulation*, 2005, pp. 21–30.

[14] A. Treuille, S. Cooper, and Z. Popović, "Continuum crowds," in *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*. New York, NY, USA: ACM Press, 2006, pp. 1160–1168.

[15] C. Reynolds, "Big fast crowds on ps3," in *sandbox '06: Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames*. New York, NY, USA: ACM Press, 2006, pp. 113–121.

[16] A. Bleiweiss, "Gpu accelerated pathfinding," in *GH '08: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2008, pp. 65–74.

[17] J. van den Berg, S. Patil, J. Sewall, D. Manocha, and M. Lin, "Interactive navigation of multiple agents in crowded environments," in *I3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*. New York, NY, USA: ACM, 2008, pp. 139–147.

[18] M. Kapadia, S. Singh, W. Hewlett, and P. Faloutsos, "Egocentric affordance fields in pedestrian steering," in *I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games*. New York, NY, USA: ACM, 2009, pp. 215–223.

[19] F. Dapper, E. Prestes, M. A. P. Idiart, and L. P. Nedel, "Simulating pedestrian behavior with potential fields," in *Advances in Computer Graphics*, ser. Lecture Notes in Computer Science, vol. 4035. Springer Verlag, 2006, pp. 324–335.

[20] C. Connolly and R. Grupen, "On the applications of harmonic functions to robotics," *International Journal of Robotic Systems*, vol. 10, pp. 931–946, 1993.

[21] E. Prestes, P. M. Engel, M. Trevisan, and M. A. Idiart, "Exploration method using harmonic functions," *Robotics and Autonomous Systems*, vol. 40, no. 1, pp. 25–42, 2002.

[22] O. Khatib, "Commande dynamique dans l'espace opérational des robots manipulaters en présence d'obstacles," Ph.D. dissertation, École Nationale Supérieure de l'Aéronatique et de l'Espace, France, 1980.

[23] R. J. H. H. Peaceman D. W., "The numerical solution of parabolic and elliptic differential equations," *Journal of the Society for Industrial and Applied Mathematics*, vol. 3, pp. 28–41, 1995.

[24] R. Silveira, E. Prestes, and L. P. Nedel, "Managing coherent groups," *Comput. Animat. Virtual Worlds*, vol. 19, no. 3-4, pp. 295–305, 2008.