Physics-Based Robot Motion Planning in Dynamic Multi-Body Environments

Stefan Zickler

CMU-CS-10-115

May 10th, 2010

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA

Thesis Committee:
Manuela M Veloso, Chair
Jessica K. Hodgins
James J. Kuffner Jr.
John E. Laird, University of Michigan

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by Intelligent Automation, Incorporated under grant number 654-1 (prime sponsor DARPA under grant number FA8650-08-C-7812); University of Southern California under grant number 138803 (prime sponsor Office of Naval Research under grant number N00014-09-1031); Boeing under grant number A003221; Department of Interior under grant number NBCH-1040007; National Science Foundation under grant number IIS-0637069; and L3 Communication Integrated Systems, LD., under grant number 4500244745. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Robot Motion Planning, Physics-Based Planning, Uncertainty, Replanning, Robot Soccer, Robot Minigolf

Abstract

Traditional motion planning focuses on the problem of safely navigating a robot through an obstacle-ridden environment. In this thesis, we address the question of how to perform robot motion planning in complex domains, with goals that go beyond collision-free navigation. Specifically, we are interested in problems that impose challenging constraints on the intermediate states of a plan, and problems that require the purposeful manipulation of non-actuated bodies, in environments that contain multiple, physically interacting bodies with varying degrees of controllability and predictability. Examples of such domains include physical games, such as robot soccer, where the controlled robot has to deliver the ball into the opponent's goal. For these domains, navigation only constitutes a small part of the overall planning problem. Additional planning challenges include accurately modeling and exploiting the dynamic interactions with other non-actuated bodies (e.g., dribbling a ball), and the problem of predicting and avoiding foreign-controlled bodies (e.g., opponent robots).

To plan in such domains, this thesis introduces physics-based planning methods, relying on rich models that aim to reflect the detailed dynamics of the real physical world. We introduce non-deterministic Skills and Tactics as an intelligent action sampling model for effectively reducing the size of the searchable action space. We contribute two efficient Tactics-driven planning algorithms, BK-RRT and BK-BGT, and we evaluate their performance across several challenging domains. We contribute a physics model parameter optimization method for increasing the planner's physical prediction accuracy, resulting in significantly improved real-world execution success rates. Additionally, we contribute Variable Level-Of-Detail (VLOD) planning, a method for reducing overall planning time in uncertain multi-body execution environments.

Besides relying on an extensive simulated testbed, we apply and evaluate our planning approaches in two challenging real-world robot domains. We contribute the robot minigolf domain, where a robot uses physics-based planning methods to solve freely configurable minigolf-like courses, e.g., by purposefully bouncing a ball off from obstacles. We furthermore contribute a robot soccer attacker behavior that uses physics-based planning to out-dribble opponents, which has been successfully tested as part of the "CMDragons" robot soccer Small Size League team at the RoboCup world cup in 2009.

Acknowledgments

I would like to thank the many people who have supported me over the past years as a graduate student and who have helped to facilitate this work.

First and foremost, I would like to thank my advisor, Manuela Veloso. Manuela has been a constant source of guidance and encouragement throughout my entire graduate career. I am grateful for all of her support and inspiration which has ultimately made this thesis possible. I would also like to thank the rest of my thesis committee: Jessica Hodgins, James Kuffner, and John Laird, for sharing their time and expertise, and for providing constructive advice on my work.

I would like to thank all of my collaborators, friends, and lab-mates, especially Joydeep Biswas, Stephanie Rosenthal, Sonia Chernova, Colin McMillen, Douglas Vail, and James Bruce, for sharing the ups and downs of the PhD program with me, and for providing me with invaluable discussions and ideas.

A big thank you goes to all of my CMDragons robot soccer team-mates and collaborators, in particular James Bruce, Mike Licitra, Joydeep Biswas, Philip Wasserman, and Gabriel Levi, who have collectively contributed to the robot platform that enabled the experiments of this thesis, and who have allowed the team to succeed at the various RoboCup competitions. Special credit goes to James Bruce for having developed the majority of the CMDragons software framework which acts as the team's solid foundation to this day. I would also especially like to thank Mike Licitra, for being a friend and colleague since our undergraduate days, and for designing and building the most amazing soccer-playing robots that consistently withstand all of our abuse.

Beyond CMU, I would like to thank all of the members of the RoboCup Small Size League who have contributed to the development of the SSL-Vision open-source vision system, in particular (but not limited to), Tim Laue, Oliver Birbach, and Joydeep Biswas.

Last, but not least, I want to express my deepest thanks to my family. Without their love and support, this thesis would not have happened. I want to thank my parents for having always supported my endeavors, wherever they have led me. Finally, I want to thank my wife Shannon for having been my constant companion along the way. She has always believed in me and has been my source of strength and sanity throughout the entire time. Words cannot capture my gratitude.

Things are only impossible until they're not.
Jean-Luc Picard (Star Trek: The Next Generation)

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Motion planning for mobile robots acting in the physical world is a challenging task. Traditionally, motion planning focuses on the problem of safely navigating a robot to a specific destination through an obstacle-ridden environment. More specifically, a motion planner's objective is to find a sequence of control actions that, when applied, lead the robot from its initial state to some goal state. In this thesis, we address the question of how to perform motion planning in complex environments that go beyond traditional navigation planning. Concretely, we are interested in domains with additional goals, such as the purposeful manipulation of non-actuated bodies, and domains that impose additional constraints on the intermediate states of a plan. Furthermore, we focus on planning in dynamic environments that contain multiple physically interacting bodies with varying degrees of controllability and predictability.

One example domain, that also happens to be a significant source of motivation for this thesis, is robot soccer. The challenges in robot soccer are immense. Besides being able to navigate, a soccer-playing robot needs to be able to purposefully manipulate the ball (e.g., dribble) with the strategic objectives to keep it away from opponents and to eventually pass it to a teammate or deliver it into the opponent's goal box. For a motion planner, this is a challenging task, because the non-actuated ball can only be indirectly controlled through the means of targeted collisions exerted by the controllable robot, thus requiring predictive knowledge about the physical interactions between the domain's bodies. This planning task is further complicated by the fact that the motions of adversarial robots are uncertain to predict, sensor input has noticeable latency, and a robot's execution is never completely accurate.

Another challenging example domain which we introduce in the thesis, is robot minigolf, where a robot is required to purposefully bounce the ball off obstacles to achieve a hole-

in-one. Because the robot can only exert control upon the ball during putting, a planner must be able to make accurate long term predictions about the ball and its interactions with other bodies, such as the floor or obstacles, in order to find an effective action for the robot to actuate the ball.

Moving even beyond robotics, another example is the automated control of many simulated physically interacting bodies in computer animation. Frequently, such animation problems not only require that the bodies reach a particular goal state, but also have additional constraints on the intermediate states of the solution sequence, such as the requirement that the bodies expose some specific visual behavior. A motion planner needs to include such constraints when selecting actions for the animated bodies.

This thesis addresses the problem of planning in such complex environments by introducing and investigating *physics-based* motion planning. We contribute a formal planning model that uses rigid body dynamics to represent the physical world. In this model, state transitions are performed by invoking a black-box physics engine which is able to accurately simulate the complex physical interactions between multiple mass-based bodies, thus allowing the representation of many interesting and novel motion planning problems.

To make search in the physics-based control space computationally feasible, this thesis contributes non-deterministic *Skills* and *Tactics* as a method to guide the planning algorithm and to reduce the size of the searchable action space. These Skills and Tactics make it possible to encode a variable amount of domain-dependent knowledge into the planning stage, making search possible even for problems with challenging goals, such as the purposeful manipulation of non-actuated bodies in the presence of other foreign-controlled bodies.

Additionally, Skills and Tactics also address the problem of planning in domains that have constraints on the intermediate states of the solution sequence (e.g., animation domains). Skills and Tactics can be used as a behavior model which constrains the set of selectable actions to sequences that are aimed to meet the intermediate state and goal requirements of the problem.

The thesis then introduces two concrete planning algorithms that rely on the non-deterministic Skills and Tactics for their action selection, but differ in the way they construct the search tree: *Behavioral Kinodynamic Rapidly-Exploring Random Trees (BK-RRT)* and *Behavioral Kinodynamic Balanced Growth Trees (BK-BGT)*.

The thesis also contributes several approaches to address the problem of planning and execution under uncertainty. We contribute a method to *automatically optimize* the parameters of the planner's physical domain model to best match the real world and to ultimately yield better planning and execution accuracy. Furthermore, to plan in rapidly changing environments with unpredictable foreign-controlled bodies, we present an *anytime-version*

of our BK-BGT algorithm that delivers partial solutions and thus can be used for frequent replanning. Finally, we contribute *Variable Level-Of-Detail (VLOD)* planning as a novel method for reducing overall computational time when planning in environments with poorly predictable bodies.

1.1 Approach

This thesis seeks to answer the following questions:

- **How can a motion planning algorithm efficiently find solutions for challenging control problems in physically complex environments?**
- **How can the above approach be applied on real-world robots to accurately solve motion planning problems in the presence of uncertainty?**

1.1.1 Efficient Planning in Physically Complex Domains

- **How can we model and plan in physically complex environments?**

  To successfully plan in physically complex environments, we need a model that is sufficiently detailed to accurately predict the dynamics of the real world. In Chapter 2, we introduce a formal planning model that utilizes *Rigid Body Dynamics* to provide a computationally feasible approximation of basic Newtonian physics, allowing the simulation of the physical interactions between multiple mass-based non-deformable bodies. In this model, a planning state is represented as the internal state of the entire rigid body system, whereas actions are represented as continuous forces and torques that can be applied to selected bodies. A state transition is performed by applying the action's forces and torques to a particular state and forward simulating the rigid body system by some timestep. A physics-based motion planning algorithm then searches over different possible sequences of actions to find an acceptable goal state.

  Using this physics-based planning model, it is possible to represent a rich variety of interesting and novel motion planning problems. Chapter 3 presents several example domains for which physics-based planning methods are well-suited. While the main focus of this thesis is robot motion planning, we furthermore demonstrate that our planning model is also applicable to solve difficult control problems encountered in the world of computer animation. Understanding the unique potential and challenges of the physics-based planning model is a key objective of this thesis.

- **How can we make physics-based planning efficient?**

  Given our physics-based motion planning model, we focus on how to search for solutions efficiently. Because we are interested in real-world applications, we must discover planning methods that are able to find solutions in a reasonable amount of time. Searching the space exhaustively is infeasible, due to the fact that actions consist of continuous forces and torques that can be chained in arbitrarily long sequences.

  In order to search the continuous space, we focus on *sampling-based* planning methods which explore the search space by non-deterministically selecting actions. A key aspect to efficient planning lies in *how* these actions are sampled. Especially in physics-based planning problems that require the robot to manipulate other rigid bodies, pure traditional navigation planning algorithms (e.g., Rapidly-Exploring Random Trees (RRT)) are not well suited, because they lack an informed method to reduce the set of searchable actions, thus making efficient planning difficult.

  In Chapter 4 of this thesis, we introduce a new method for constraining the size of the applicable action space. We present a behavioral model consisting of sampling-based *Skills* and *Tactics*, limiting the motion planner's search to a smaller set of actions, allowing a more informed search toward the goal state. In Chapter 5 we contribute two concrete planning algorithms that implement this action sampling model: *Behavioral Kinodynamic Rapidly-Exploring Random Trees (BK-RRT)* and *Behavioral Kinodynamic Balanced Growth Trees (BK-BGT).*

1.1.2 Physics-Based Planning under Uncertainty

A major focal point of this thesis is to investigate how physics-based motion planning can be applied in real-world robot environments, characterized by the presence of uncertainty. Robot sensing is typically imperfect and accompanied by latency. Planning itself introduces additional latency, thus effectively out-dating the planner's original assumptions about the world's initial state by the time that the robot begins to finally execute the planner's solution. Even when using rich physics-based models, the planner's predictions are bound to never be completely accurate. Furthermore, a domain may contain bodies that are controlled by a foreign entity, thus making accurate predictions extremely unlikely.

This thesis addresses the problem of uncertainty through the following questions:

- **How can we increase model accuracy?**

  A significant source of uncertainty is model inaccuracy. To accurately predict real-world physics, the planner's domain model needs to be carefully configured through

various parameters. Typically, however, it is not inherently clear what the optimal parameter values are. Resorting to best-guessed values can lead to false predictions during planning and ultimately result in failure during real-world execution. Chapter 8 introduces an automated physics-based model parameter optimization technique that is able to improve the physics-based planner's real-world execution accuracy by finding the set of parameter values that minimizes the error between the simulated model and the real-world.

- **How can we plan in real-world dynamic environments?**

  Motion planning can become very challenging when the robot's surroundings keep changing during the course of planning and execution.

  In some instances, changes in the domain can be relatively accurately predicted during the planning stage by using sensing and motion extrapolation techniques. However, even with very accurate sensing and prediction models, execution will only succeed if the robot is able to accurately perform and time the planner's solution. Chapter 7 discusses all of these prediction and execution challenges in detail and demonstrates how our physics-based motion planner can be integrated into a dynamic real-world robot system.

  In some cases, however, long-term prediction of the domain's dynamics becomes impossible. An extreme example are domains that contain other unpredictable, or even adversarial bodies that are controlled by an external entity. Any predictions or extrapolations made in such domains will rapidly diverge from reality, the further we predict into the future. In such environments, the only feasible way to deal with the domain's uncertainty is *replanning*. When using replanning, the robot only executes a portion of the planner's generated solution after which it re-observes the state of the world and then repeats the planning process using the newly observed world state as its new initial state. This process is repeated in a continuous *replanning loop*. Chapter 7 investigates the trade-offs and presents an integration of rapid replanning into a dynamic physics-based domain.

- **How can we replan more efficiently in the presence of poorly predictable bodies?**

  A major problem of replanning is its computational cost. At each replanning iteration, the planner performs an intensive search for a complete solution that will bring the robot from its current state all the way to the final goal state. During this search, our physics-based motion planner will ensure that the resulting plan is dynamically sound and collision-free by employing its computationally expensive models to predict the robot's motions and its interactions with the predicted environment. Such an elaborate search might seem unnecessary, given the fact that the robot will only execute a small portion of the resulting plan, before discarding it and re-invoking

the planner to start from scratch in the next replanning iteration. However, simply limiting the planner's search depth, an approach also known as *finite-horizon planning*, is dangerous because it can lead to a robot becoming stuck in a local search minima as there are no guarantees that a partial plan will actually lead to the final goal state.

In Chapter 9 of this thesis, we introduce *Variable Level-Of-Detail (VLOD)* planning, a novel approach to reduce the computational overhead of replanning in poorly predictable multi-body environments. VLOD planning is able to focus its search on obtaining accurate short-term results, while considering the far-future with a different level of detail, selectively ignoring the physical interactions with poorly predictable dynamic objects (e.g., other mobile bodies that are controlled by external entities). Unlike finite-horizon planning, which limits the maximum search depth, VLOD planning deals with local minima and generates full plans to the goal, while requiring much less computation than traditional planning.

1.1.3 Evaluation

We use several simulated and real-world domains to experimentally evaluate the algorithms presented in this thesis. Chapter 3 provides a detailed overview of the domains.

For the real-world domains, we use the *CMDragons* multi-robot system as our experimental platform. The CMDragons system consists of multiple small homogeneous robots that are wirelessly controlled by an offboard computer featuring an overhead global vision system for robot localization. Using the CMDragons system, we contribute two real-world experimental domains:

- **Robot Soccer**
  The robot soccer domain requires a robot to autonomously play a game of soccer against a robot opponent. Robot soccer is a challenging planning environment because the game is very fast-paced and adversarial. Succeeding in robot soccer requires both low-level ball manipulation capabilities, as well as higher level tactics.
- **Robot Minigolf**
  The robot minigolf domain requires a CMDragons robot to putt a golf ball into a hole by purposefully bouncing it from wooden bars. The initial positions of the hole, bars, and golf ball can be freely configured to build novel and challenging course configurations. Planning in the minigolf domain can be extremely challenging because it requires an accurate model of the domain's dynamics. To further increase the challenge, minigolf courses can also contain moving obstacles, thus requiring extremely accurate sensing, planning, and execution timing from the robot.

1.2 Contributions

This thesis makes the following contributions:

- An introduction to *physics-based* motion planning in dynamic multi-body environments. We introduce a *formal planning model* based on rigid body dynamics and specify its unique planning characteristics and challenges.
- Non-deterministic *Skills* and *Tactics* as an efficient action sampling model.
- Two efficient Tactics-based planning algorithms: *Behavioral Kinodynamic Rapidly-Exploring Random Trees (BK-RRT)* and *Behavioral Kinodynamic Balanced Growth Trees (BK-BGT).*
- *Variable Level-Of-Detail* planning, a method for reducing overall planning time in uncertain multi-body execution environments that require replanning.
- A method for increasing physics-based planning accuracy through *automated physics model parameter optimization.*
- An *anytime* planning version of BK-BGT and its integration into a *robot soccer attacker behavior* which has been successfully tested at the RoboCup Small Size League World Cup.
- A *minigolf-playing robot* that is able to autonomously solve complex novel course configurations using our physics-based planning approach.

1.3 Document Outline

- **Chapter 1 – Introduction**
  We provide an introduction to this thesis. We outline the core questions that this thesis seeks to answer and we summarize our major contributions.
- **Chapter 2 – Physics-Based Planning Model and Basic Algorithm**
  We present a formalization of the physics-based motion planning problem. We introduce a taxonomy of body types that can be encountered in physics-based domains and we highlight some of the key challenges when planning in complex multi-body environments. Finally, we present a basic skeleton planning algorithm that the remaining algorithms in this thesis are based on.

- **Chapter 3 – Applications and Domains**
  We present possible applications for our work and provide an overview of the various domains used for evaluation throughout this thesis, including simulated and real-world implementations.

- **Chapter 4 – Non-Deterministic Skills and Tactics**
  We introduce a novel version of *Skills* and *Tactics* as a method for efficiently reducing the size of the physics-based search space by constraining the set of applicable actions.

- **Chapter 5 – Efficient Planning via Skills and Tactics**
  We introduce two concrete Skills and Tactics-driven planning algorithms: *BK-RRT* and *BK-BGT*.

- **Chapter 6 – Empirical Evaluation in Simulated Domains**
  We evaluate the performance of the BK-RRT and BK-BGT algorithms in several simulated domains.

- **Chapter 7 – Planning in Real-World Dynamic Environments**
  We demonstrate how our planning approaches can be applied to real-time robot systems with domains containing foreign-controlled bodies. We discuss and evaluate various integration challenges. As specific examples, we present our planner's integration into the robot minigolf and robot soccer domains.

- **Chapter 8 – Parameter Optimization of Physics-Based Models**
  We present an automated physics model parameter optimization approach. We experimentally evaluate the approach in the robot minigolf domain.

- **Chapter 9 – Variable Level-Of-Detail Planning**
  We introduce and evaluate *Variable Level-Of-Detail (VLOD)* planning as a method for reducing overall planning time in uncertain multi-body execution environments.

- **Chapter 10 – Related Work**
  We discuss previous work that relates to this thesis.

- **Chapter 11 – Conclusion and Future Work**
  We conclude this thesis with a summary of its contributions and with a discussion of possible directions for future work.

- **Appendix A – The CMDragons Multi-Robot System**
  We present the *CMDragons* multi-robot system that is used for the robot soccer and robot minigolf domains. We explain the robot hardware and the system's overall behavioral control architecture.

- **Appendix B – The SSL-Vision Shared Vision System**
  We present *SSL-Vision*, the open source vision system that provides sensing for the CMDragons multi-robot system.

- **Appendix C – Example Skills**
  We show concrete examples of multiple selected Skills.

- **Appendix D – List of Notation**
  We provide an index of frequently used notation in this thesis.

- **Appendix E – List of Videos**
  We provide a list of supplemental videos for this thesis, available to download and view at `http://szickler.net/thesis/`.

Chapter 2

Physics-Based Planning Model and Basic Algorithm

This chapter formally introduces the physics-based motion planning model, including its states, actions, and state transitions. Furthermore, we discuss some of the unique challenges encountered in many physics-based planning problems. Finally, we present a baseline planning algorithm for finding solutions in the physics-based motion search space.

2.1 Definitions

We first define the general planning problem. Let X be the state space, let $x_{\text{init}} \in X$ be the initial state, let $X_{\text{goal}} \subset X$ be a set of goal states, and let A be a set of possible actions. Furthermore, let e be a state transition function that can apply an action $a \in A$ to a particular state $x \in X$ and compute a succeeding state $x' \in X$, that is: $e : \langle x, a \rangle \rightarrow x'$.

A planner's objective is to find a sequence of actions $a_1, \ldots, a_n$, which, when applied in order and starting from x_{init}, ends in a goal state $x_{\text{goal}} \in X_{\text{goal}}$. Additional constraints can be imposed on all the intermediate states of the action sequence by defining only a subset of the state space to be valid ($X_{\text{valid}} \subseteq X$) and requiring that all states of the solution sequence $x_{\text{init}}, x_1, x_2, \ldots, x_{\text{goal}}$ are elements of X_{valid}.

Motion planning describes planning models where the state space X represents some set of physical configurations and the actions in A correspond to motions applicable to that space. Chapter 10 contains a detailed discussion of many traditional motion planning models and methods.

We introduce a *physics-based planner* as a special type of motion planner which uses domain models that aim to reflect the inherent physical properties of the real world. The *Rigid Body Dynamics* model [5, 6] provides a computationally feasible approximation of basic Newtonian physics, and allows the simulation of the physical interactions between multiple mass-based non-deformable bodies. The term *Dynamics* implies that rigid body simulators are second order systems, able to simulate physical properties over time, such as momentum and force-based inter-body collisions. Physics-based planning can be seen as an extension to *kinodynamic planning* [30], adding the simulation of rigid body interactions to traditional second order navigation planning [93].

2.1.1 Rigid Bodies and Parameters

Our physics-based planner models the physical world as a rigid body system, composed of n rigid bodies $r_1 \dots r_n$.

Definition 2.1.1 (Rigid Body)**.** A rigid body r is defined by two disjoint subsets of parameters $r = \langle \hat{r}, \bar{r} \rangle$ where

- $\hat{r}$: the body's mutable state parameters,
- $\bar{r}$: the body's immutable parameters.

Mutable Body Parameters

Definition 2.1.2 (Mutable Body Parameters)**.** The body's mutable parameters, $\hat{r}$, are the parameters of the body that are allowed to be manipulated during the course of planning through the application of actions. Let $\hat{r}$ be a tuple, containing the second order state parameters of the rigid body, namely its position, orientation, and their derivatives:

$$\hat{r} = \langle \alpha, \beta, \gamma, \omega \rangle$$

where

- α : position (3D vector),
- β : orientation (3×3 rotation matrix),
- γ : linear velocity (3D vector),
- ω : angular velocity (3D vector).

Immutable Body Parameters

Definition 2.1.3 (Immutable Body Parameters)**.** The body's immutable parameters, $\bar{r}$, are the parameters that describe the fixed nature of each body and that remain constant during the course of planning. More concretely, $\bar{r}$ is a tuple $\bar{r} = \langle \phi_{\text{Shape}}, \phi_{\text{Mass}}, \phi_{\text{MassC}}, \phi_{\text{FricS}}, \phi_{\text{FricD}}, \phi_{\text{Rest}}, \phi_{\text{DampL}}, \phi_{\text{DampA}} \rangle$, describing all of the rigid body's inherent physical parameters, where:

- ϕ_{Shape} : shape description of the body (3D mesh or other 3D primitive),
- ϕ_{Mass} : mass of the body,
- ϕ_{MassC} : center of mass in body's coordinate frame (3D vector),
- ϕ_{FricS} : static friction coefficient,
- ϕ_{FricD} : dynamic friction coefficient,
- ϕ_{Rest} : restitution coefficient,
- ϕ_{DampL} : linear damping coefficient,
- ϕ_{DampA} : angular damping coefficient.

It should be noted that the above list represents a non-exhaustive set of physical body parameters that are typically supported by modern rigid body simulation techniques. Many modern rigid body simulators support additional body configuration parameters that could be added to $\bar{r}$ if needed. For example, several simulators support the application of multiple materials, and therefore multiple friction and restitution coefficients, for different locations on a single rigid body shape.

2.1.2 States, Actions, and Domain Model

Definition 2.1.4 (State and State Space)**.** The physics-based planning state space X is defined by the mutable states of all n rigid bodies in the domain and time t. That is, a state $x \in X$ is defined as the tuple

$$x = \langle t,\ \hat{r}_1, \ldots, \hat{r}_n \rangle.$$

Definition 2.1.5 (Action and Action Space)**.** The action space A is the set of the applicable controls that the physics-based planner can search over. An action $a \in A$ is defined as a vector of sub-actions $a = \langle \hat{a}_1, \ldots, \hat{a}_n \rangle$, where $\hat{a}_i$ represents a pair $\hat{a}_i = \langle force, torque \rangle$, where *force* and *torque* are each 3D vectors, applicable to a corresponding rigid body r_i.

Definition 2.1.6 (Domain)**.** A physics-based planning domain d is defined as the tuple $d = \langle G, \bar{r}_1 \ldots \bar{r}_n, A \rangle$ where

- G : global gravity force vector,
- $\bar{r}_1 \ldots \bar{r}_n$: immutable parameters of all n rigid bodies,
- A : the set of applicable actions.

Similarly to the tuple of physical parameters $\bar{r}$, the domain description tuple d is not necessarily exhaustive and could be extended to include additional configuration parameters for certain features of a particular rigid body simulation technique. One example of such an additional feature would be the modeling of joints. Similar to their anatomical equivalent connecting two bones, a joint can link multiple rigid bodies together by enforcing a certain set of kinematic motion contraints between them. Examples include spherical joints, revolute joints, or prismatic joints. Because the definition of the joint (i.e., its type and motion limits) do normally not change throughout the life of the rigid body system, we could encode them as part of the domain description, that is: $d = \langle G, \bar{r}_1 \ldots \bar{r}_n, A, joint_1, joint_2, \ldots \rangle$.

2.1.3 State Transitions

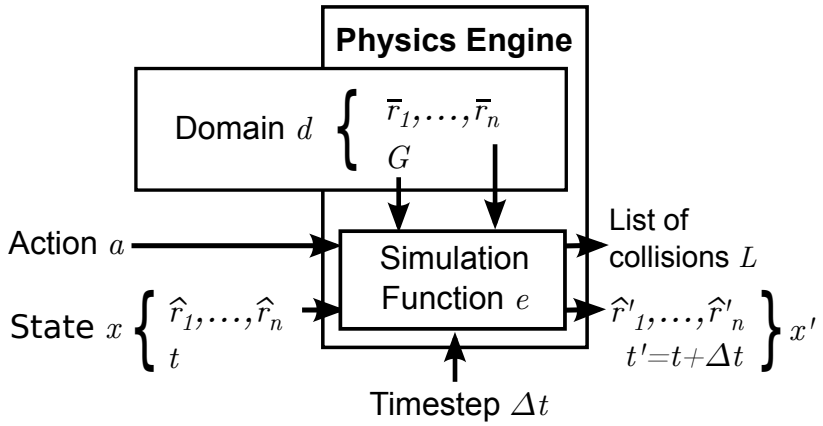


Figure 2.1: A physics engine computes state transitions.

A physics-based planner searches for solutions by reasoning about the states resulting from the application of possible actions. The state transition computations are performed by simulation of the rigid body dynamics. The task of computationally simulating rigid body dynamics is well-understood [6]. There are several robust third-party rigid body simulation frameworks freely available, such as the Open Dynamics Engine (ODE) [81], and NVIDIA PhysX [24]. Frequently referred to as *physics engines*, our planning approach employs one of these simulators as a "black box" to simulate state transitions in the physics space (see Figure 2.1).

Definition 2.1.7 (State Transition Function)**.** We define the physics state transition function e:

$$e : \langle \hat{r}_1, \ldots, \hat{r}_n, a, G, \bar{r}_1, \ldots, \bar{r}_n, \Delta t \rangle \rightarrow \langle \hat{r}'_1, \ldots, \hat{r}'_n, L \rangle.$$

The transition function e is part of the physics engine. Given a current state of the world x, the planner supplies the contained rigid body states $\hat{r}_1, \ldots, \hat{r}_n$ and a control action vector a to the the simulation function e. Using the immutable state parameters $\bar{r}_1, \ldots, \bar{r}_n$ and the global gravity vector G (both contained in the domain description d), the physics engine's e function then simulates the rigid body dynamics forward in time by a fixed timestep Δt, delivering new states for all rigid bodies $\hat{r}'_1, \ldots, \hat{r}'_n$. These rigid body states are then stored in the new resulting planning state x' along with its new time index $t' = t + \Delta t$. Additionally, e also returns a list of collisions $L = \langle l_1, l_2, \ldots \rangle$ that occurred during forward simulation. Each item $l \in L$ is an unordered pair $l = \langle \lambda_1, \lambda_2 \rangle$, consisting of the indices of the two rigid bodies $r_{\lambda_1}, r_{\lambda_2}$ involved in the collision.

2.1.4 Intermediate Constraints and Planning Problem Definition

Generally, the goal of a planner is to find a sequence of actions, that, when applied in sequence by using the transition function e, leads from some initial state x_{init} to a valid goal state $x_{\text{goal}} \in X_{\text{goal}}$. However, some problems might have additional requirements on the *intermediate* states of the solution sequence. For example, in a navigation domain, it is typically desirable that collisions never occur within a solution. To enforce such intermediate state constraints, we introduce the `Validate` function that checks the state transition function's output (namely the resulting state x' and the list of collisions L) to determine whether x' is a *valid* state:

Definition 2.1.8 (Validation Function)**.**

$$\texttt{Validate}(x', L) = \begin{cases} \textit{true} & \textbf{if } x' \text{ is valid,} \\ \textit{false} & \textbf{if } x' \text{ is invalid.} \end{cases}$$

The actual internal mapping of the `Validate` function (i.e. which specific inputs should generate a *false* as output) is problem-dependent, and needs to be provided, similar to the actual set of goal states. For problems without intermediate state constraints, the function can simply be configured to always return *true*, thus making any newly generated state x' a valid state.

Definition 2.1.9 (Problem)**.** Given a particular domain, it is possible to represent many planning *problems*. A particular planning problem is defined by its domain d, its initial state x_{init}, the set of valid goal states X_{goal}, and any constraints on the intermediate states, defined by the validation function `Validate`.

2.2 Rigid Body Simulations

Our state transition model effectively treats the physics engine as a "black box," separating our planning approaches from the question of *how* the engine's internal rigid body dynamics simulations are actually implemented. Nevertheless, it is important to discuss and understand several key aspects of the engine's internal simulation techniques because their configuration can have significant implications on the quality and computational performance of the engine's simulations.

2.2.1 Level of Time Discretization

In Section 2.1.3, we have shown how to invoke the forward simulation function e by passing it a set of rigid body states, their actions, a domain description, and a timestep Δt. To deliver the resulting body states and collision list that e returns, the physics engine needs to accurately compute and resolve the physical motions and interactions of the rigid bodies. Current physics engines rely on *discrete* timestep approaches that repeatedly apply integration techniques to forward-predict the state of bodies for some fixed *internal* timestep $\bar{t}$, attempting to approximate the true continuous motions of real rigid bodies. Note, that we differentiate this internal timestep $\bar{t}$ from the timestep Δt that is passed into e. The value of $\bar{t}$ is some unit fraction of Δt, that is $\bar{t} = \Delta t / \mathit{steps}$ where *steps* is a positive integer, defining the number of internal integration steps to be performed per invocation of e.

The value of $\bar{t}$ plays an important role in controlling the accuracy and performance of the rigid body simulations. Generally speaking, a smaller value of $\bar{t}$ delivers more accurate predictions of rigid body interactions, but also incurs greater computational cost. Similarly, selecting a large value of $\bar{t}$ (at most $\bar{t} = \Delta t$), is likely to result in computationally faster,

but less accurate simulations. The primary cause for this behavior is that between each internal timestep, the physics engine will perform *collision checks* to detect whether any of the rigid bodies are currently penetrating one another. If a collision is detected, the engine will attempt to resolve it, by computing the resulting forces of the collision and by adjusting the positions of the bodies involved. The more frequently these checks are performed and resolved, the more accurately does the resulting motion sequence approximate the true rigid body collision. However, each collision check and resolution comes at a computational cost.

Choosing a larger value of $\bar{t}$ can not only lead to less accurate collisions, but can in fact also lead to completely missed collisions. This error is particularly likely to arise, if the domain contains fast moving bodies. Figure 2.2 shows an example of how choosing different values of $\bar{t}$ can affect collision detection. Each subfigure shows a ball body approaching a rectangular body with the same initial conditions. In subfigure (a), the timestep is small enough so that the engine correctly detects the collision between the two bodies, allowing collision resolution to take place and forcing the ball body to its correct post-collision path. In subfigure (b), however, a larger value of $\bar{t}$ yields much longer distances travelled by the ball between individual collision checks, thus never actually detecting the collision between the two bodies, delivering a physically incorrect result.

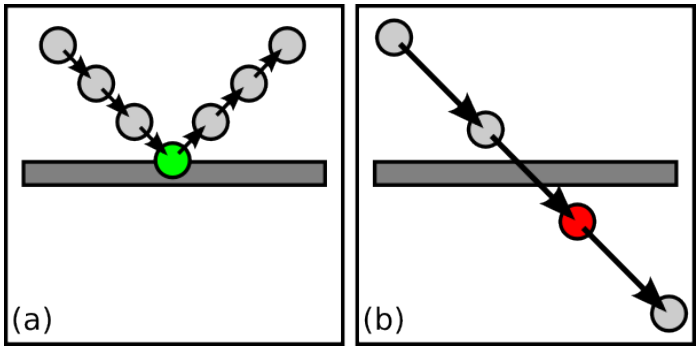


Figure 2.2: An example of successful (a) and unsuccessful (b) collision detection.

Several modern physics engines (e.g., [24]) provide *continuous collision detection* (CCD) methods, that attempt to detect collisions even when occurring completely between two succeeding timesteps. Although CCD does avoid the problem of missed collisions, it is a computationally intensive process, especially when body shapes are complex. Furthermore, CCD typically assumes a linear interpolation between body positions, which can still lead to inaccurate collision detections.

In this thesis, we do not use CCD in any of our experiments, but instead select a $\Delta t = 1/60s$, and $steps = 4$, thus letting $\bar{t} = 1/240s$ for all experiments, unless noted otherwise. These values are sufficiently small to allow accurate collision detection and resolution in

our domains, while also being computationally feasible.

2.2.2 Determinism

Besides the level of time discretization, another interesting aspect to discuss is determinism. We call the simulation function e deterministic, if the mapping from its input to its output does not change. In other words, if we were to call e multiple times with identical input states and actions, then its generated output states and collisions should be exactly the same, every single time. Such determinism is typically a desirable quality because it ensures consistency during simulations and scientific experiments.

Although one would expect a physics engine's simulation routine to behave deterministically, this is indeed *not* the case for many modern engines, including the engine used throughout our experiments (NVIDIA PhysX Version 2.8.1). The precise reasons for this behavior are not disclosed by the creators of the engine, but it is fair to assume that they consider small amounts of non-determinism acceptable in order to increase computational performance. Example causes for this non-determinism could be algorithms that use un-initialized memory blocks containing random data, or algorithms that use pseudo-random-number-generators in order to perform certain optimizations. Despite the fact that we were able to detect occasional non-determinism in our engine of choice, its effects were generally negligible, especially when compared to the impact of the controlled variables in our experiments. Interestingly, full determinism was recently announced as a new feature in a future version of the engine employed in our experiments.

2.3 Body Types

The difficulty of physics-based planning problems is clearly related to the types of bodies present in the domain. We introduce a hierarchy to classify the types of bodies in our domain model (see Figure 2.3).

Every body is by definition a *rigid body*. There are *static* rigid bodies that do not move, even when a collision occurrs, which are often used to model the ground plane and all non-movable bodies, such as walls and heavy objects. All other bodies are *manipulatable*, meaning that they react to collision forces exerted upon them. Among these, the planner can directly control the *actively controlled* bodies, i.e., the planner has available actions directly applicable to these bodies.

There are two other groups of bodies that are manipulatable, but not actively controlled,

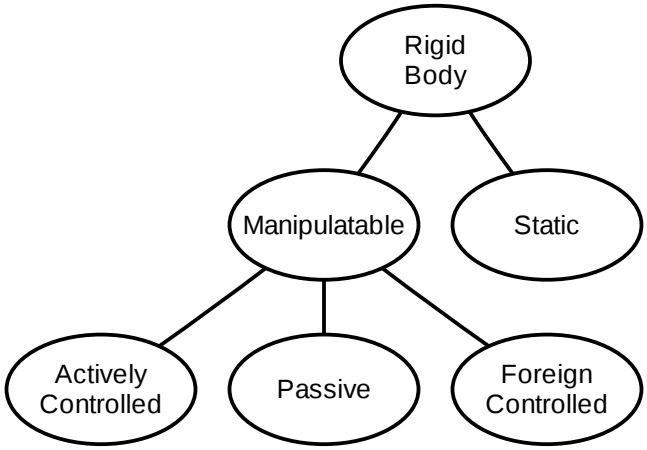


Figure 2.3: Rigid body classes

namely *passive* and *foreign-controlled* bodies. *Passive* bodies can only be actuated by external influences and interactions, such as being carried or pushed. *Foreign-controlled* bodies are actively actuated, but by external control to our planner.

The predictability of the actions taken by foreign controlled bodies can vary. An example of predictable foreign-controlled bodies are objects that are continuously actuated with a clearly observable and linearly extrapolatable motion, such as an escalator or a windmill. An example of a much less predictable body would be a robot executing a "random walk", repeatedly changing its direction of motion in an unpredictable fashion. Similarly, the actions of adversarial robots (e.g., the opponent robots in robot soccer) are typically not fully predictable. However, as we show in the following chapters, it can still be helpful to create models that will assume an adversary's *likely* action to be taken. Furthermore, in most domains, the possible motions of foreign-controlled bodies can be bounded, due to physical limits on the body's maximum acceleration, given some knowledge about its mass and types of actuators. Such bounds effectively constrain the possible space that the foreign-controlled body can reach within a particular amount of time, regardless of what action it chooses to execute.

Definition 2.3.1 (Rigid Body Classes)**.** The classification hierarchy in Figure 2.3 defines corresponding sets of rigid bodies which we refer to as B_{Rigid}, B_{Manip}, B_{Static}, $B_{\text{Controlled}}$, B_{Passive}, B_{Foreign}, with the following relationships:

- $B_{\text{Rigid}} = \{r_1, \ldots, r_n\}$,
- $B_{\text{Manip}} \cup B_{\text{Static}} = B_{\text{Rigid}}$ where $B_{\text{Manip}}, B_{\text{Static}}$ are disjoint sets,
- $B_{\text{Controlled}} \cup B_{\text{Passive}} \cup B_{\text{Foreign}} = B_{\text{Manip}}$ where $B_{\text{Controlled}}, B_{\text{Passive}}, B_{\text{Foreign}}$ are all mutually disjoint sets.

2.4 Problem Classes and Challenges

Given our formal planning model definitions and classification of body types, we now discuss the different possible types of planning problems and their unique challenges. First, we identify two main problem classes:

- $P_{\mathbf{Nav}}$ **- Navigation Problems**
  Problems in this class require that the *actively controlled* body reaches a particular target state. In other words, the set of goal states is defined by some constraints on the *actively controlled* body's state (e.g., its position, orientation, velocity, or some combination thereof).

- $P_{\mathbf{Man}}$ **- Manipulation Problems**
  Problems in this class require that a *passive* body reaches a particular target state, meaning that the set of goal states is defined by some constraints on the *passive* body's state.

The two problem classes are not mutually exclusive. It is possible to construct a set of goal states that requires both *actively controlled* and *passive* bodies to reach a particular state, thus creating a planning problem that is a member of both the classes P_{Nav} and P_{Man}.

Robot navigation is a prime example of a P_{Nav} problem. Given an initial position, the planner's goal is to find a sequence of actions that will lead the actively controlled robot to some goal position. As we discuss in Chapter 10, this is a well-explored problem that can often be solved sufficiently well with existing planning techniques.

Problems in the class P_{Man} however, can be significantly more challenging than problems of class P_{Nav}. The core reason for this is the lack of direct controllability over the passive body. In a typical problem of class P_{Man}, the planner has to find actions for the actively controlled body to *manipulate* the passive body into its goal state. The task of manipulating a passive body can be challenging for several reasons:

- **Lack of Simple Goal Heuristics**
  The searchable space of actions in our planning model is extremely large, for problems of either class P_{Man} or P_{Nav}. At any given point in time, the planner can choose from a continuous set of forces and torques to be applied on the actively controlled body. Navigation planning problems of class P_{Nav} can overcome searching this vast space through the use of relatively simple heuristics. Given a particular state, it is possible to heuristically bias the search toward the goal state by selecting actions which are *likely* to lead the robot towards its goal location. Computing such heuristics can be

as simple as taking the Euclidean distance between the controlled body and the goal state to express the relative progress toward achieving the goal. Chapter 10 discusses several traditional planning methods that heavily rely on such heuristics.

In problems of P_{Man} however, it is not always clear how to choose the action which is likely to bring the search closer to the goal state. Remember, that in a manipulation problem, the goal is defined in terms of parameters of the passive body which is being manipulated. However, the planner needs to choose between actions for the *actively controlled* body. Therefore, the planner would require a heuristic that is able to measure how far the actively controlled body is in achieving the overall domain goal. Defining such a heuristic can be difficult, as the active body's state itself is *unrelated* to the domain goal. To be useful, the heuristic would need some kind of reasoning as to which controlled body action is likely to lead to a successful manipulation of the passive body. Such reasoning, however, is difficult to define, because it strongly depends on the actual physical collision dynamics of the bodies involved.

- **Passive Body Dynamics**
  A related challenge for problems in class P_{Man} is that the planner has to accurately predict and deal with the complex dynamics of the passive body that it is trying to manipulate. Figure 2.4 shows an example illustrating this problem. Here, the goal is to manipulate the ball efficiently around the 90 degree corner in the corridor. In order to do so, the controlled robot needs to anticipate the ball's momentum after it has been set into motion. In order to change the ball's direction efficiently, it requires a sophisticated control strategy, moving the robot ahead of the ball, re-aligning to counter-act the ball's anticipated approach vector, and then re-accelerating it into the final desired direction.

- **Bifurcations of the Space**
  A third reason for the increased difficulty of class P_{Man} is that the effects of rigid body collisions add discontinuities to the search space by introducing bifurcations. For example, in the game of pool, a cue ball could either hit a target ball, thus giving it a final bump into its pocket, or it could miss it, not having any effects on the target ball at all. Thus, even a small error in the goal-heuristic can often decide between complete success or failure. In traditional navigation problems, on the other hand, the relationship between the space of actions and their resulting motions is typically more continuous and therefore less fragile to non-optimal action selection. For example, in many navigation problems, a non-optimal goal-trajectory can still lead to an acceptable solution. This discontinuity in the search space is worsened in manipulation problems where entire chains of accurate collisions are required to reach the goal state.

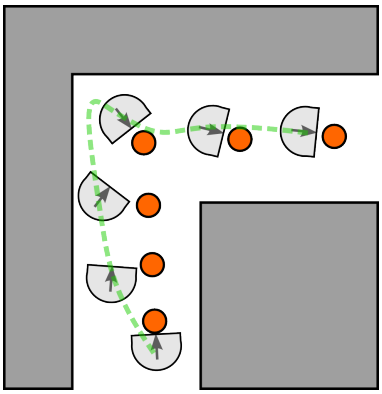


Figure 2.4: Deliberately manipulating a passive body solely through the means of collisions can require elaborate maneuvers by the controlled body to counter-act undesired effects of the passive body's dynamics.

The difficulty of planning problems in our model does not only depend on whether the problem belongs to P_{Man} or P_{Nav}. Problems of either class can become significantly more challenging for several other reasons, in particular including the following:

- **Intermediate State Constraints**
  In some domains, the planner is not only required to find a solution sequence of actions that leads to a goal state, but it is also required to ensure that the intermediate states of this solution sequence meet some particular user-defined constraints. In our planning model, The `Validate` function (see Definition 2.1.8) is used to encode such intermediate constraints. For example, in robot navigation domains, a typical constraint would be that the controlled robot does not collide with any other body in any of the states of the entire solution sequence. In computer animation domains, such intermediate constraints are often more subtle, requiring that the body's *behavior* during the solution sequence fullfils some particular requirements imposed by an animator. For example, if we were to use physics-based motion planning to compute the actions to be taken for an animated ice skating character, it might be desirable that the character performs a particular series of tricks on its path from its initial state to its goal state. Imposing such additional constraints can make a search for solutions challenging, as the set of valid solutions is reduced in size while the searchable action space remains the same. In Chapter 4, we present models that attempt to solve this problem by reducing the size of the searchable action space.

- **Unpredictable and Adversarial Bodies**
  Both navigation and manipulation problems can become significantly more challenging when poorly predictable foreign-controlled bodies are present in the domain. Adversarial domains are an extreme example, where foreign-controlled bodies are

not only poorly predictable, but are indeed actively attempting to stop the planner from succeeding.

2.5 A Basic Physics-Based Planning Algorithm

Given our formal planning model, we now outline a basic planning algorithm that searches for valid solutions in the physics-based planning space.

Algorithm 1 shows our basic planner. Note, that we use the dot symbol to indicate access to a member of a tuple (e.g., $x.t$ refers to the t item stored in x). The algorithm is based on a purely forward planning tree-based search. We initialize the search with a tree containing only the initial state $x_{init} \in X$ as its root. We then enter the main planning loop, which runs for a predefined maximum number of search iterations z, if no solution is found earlier. On each iteration, the algorithm calls a function `SelectSourceNode` to select an existing node x from the `tree`. How this node is best selected is an interesting question that will be addressed in Chapter 5, but for now, let us assume that the planner is uninformed and randomly selects an existing node from `tree`. Next, the planner calls the function `GenerateAction` to generate an action a from the action space A. How this action is generated can strongly influence the overall behavior and performance of our planner and will be discussed in detail in Chapter 4. For simplicity, let us again assume a purely uninformed approach, selecting a random action from the space of possible actions A.

After selecting the source node x and generating an action a, the planner can now invoke the physics transition function e that will forward-simulate the rigid body dynamics and generate the new rigid body states $\hat{r}_1, \ldots, \hat{r}_n$ as well as a list of collisions L that occurred during the forward simulation. The new rigid body states and the new timestamp t' are then combined to form the resulting state x'. The `Validate` function then checks whether the resulting state is a valid state by making sure that it did not validate any user-defined constraints and that no undesired collisions occurred in L. If accepted, the algorithm adds x' to the search tree as a child of its source node x and furthermore stores the action within the branch.

The complete loop of node selection and tree expansion is repeated until the algorithm either reaches a goal state (i.e., $x' \in X_{\text{goal}}$), or until it reaches the maximum allowed number of iterations z, at which point the search returns failure. Once a goal state is reached, the algorithm simply traces back the chain of states and actions from the goal state to the root of the tree and returns it in reverse order as the final solution sequence in the form $\langle x_{\text{init}}, a_1, x_2, a_2, x_3, \ldots, x_{\text{goal}} \rangle$.

Algorithm 1: BasicPlanner

Input: Domain: d, initial state: x_{init}, set of goal states: X_{goal}, timestep: Δt, validation function: `Validate`, max iterations: z.

```
tree ← NewEmptyTree();
tree.AddNode(x_init);
for iter ← 1 to z do
    x ← SelectSourceNode(tree);
    a ← GenerateAction(x, d.A);
    ⟨r̂'_1, ..., r̂'_n, L⟩ ← e(x.⟨r̂_1, ..., r̂_n⟩, a, d.G, d.⟨r̄_1, ..., r̄_n⟩, Δt);
    t' ← x.t + Δt;
    x' ← ⟨t', r̂'_1, ..., r̂'_n⟩;
    if Validate(x', L) then
        tree.AddNode(x');
        tree.AddEdge(x, x', a);
        if x' ∈ X_goal then
            return TraceBack(x', tree);
return Failed;
```

2.6 Chapter Summary

In this chapter, we have formally introduced physics-based planning, defining its state, action, domain, and transition models. We have shown how to employ a physics engine to perform the underlying rigid body simulations and we have discussed how simulation quality can be affected by the internal configuration of the physics engine. We have furthermore introduced a hierarchy of body types that we can encounter in physics-based domains and we have discussed the different classes of problems and challenges that a physics-based planner may encounter. Finally, we have presented a basic example for a physics-based planning algorithm.

In the following chapters, we present physics-based planning methods that build upon the basic algorithm presented in this chapter. In particular, we introduce methods that aim to specifically address all of the planning challenges discussed in Section 2.4. The next chapter introduces a set of domains that collectively covers all of these planning challenges, allowing us to exhaustively test our approaches throughout the remainder of this thesis.

Chapter 3

Applications and Domains

This chapter presents target applications for our motion planning approaches and concretely introduces multiple example domains that are used for evaluation throughout this thesis.

3.1 Application in Robot Motion Planning

The primary target application of our work is robot motion control in complex environments. Real-world robot motion control is naturally challenging due to the presence of sensory noise and execution uncertainty. Additionally, many robot domains have strict timing and computational constraints, requiring efficient planning within a limited amount of time. Besides such traditional challenges, we are furthermore interested in testing our planner's performance on the challenges that we discussed in Section 2.4, including the manipulation of passive bodies and the presence of not perfectly predictable foreign-controlled bodies. To evaluate our approach, we introduce multiple robot domains, collectively covering many interesting motion planning challenges.

3.1.1 Navigation Domain

In the *navigation* domain, a robot's objective is to navigate from some initial configuration to some goal configuration while avoiding collisions with obstacles. Obstacles in the domain can be static, such as surrounding walls, or they can be foreign-controlled dynamic obstacles, such as other robots (see Section 2.3 for a formal definition of different body

types). The navigation domain is challenging for several reasons:

- **Dynamics**
  Real robots have a non-zero mass which in turn implies that they are unable to make arbitrary changes to velocity (e.g., a robot is unable to stop instantaneously). Therefore, for safe and precise navigation, a planner must accurately model the robot's dynamics, including momentum, friction, and limits in applicable acceleration.

- **Efficient Exploration**
  The navigation domain may contain a challenging configuration of stationary obstacles (e.g., walls). A planner must be able to efficiently search this space to find a valid trajectory to the goal configuration without getting stuck in local search minima.

- **Foreign-Controlled Obstacles**
  The navigation domain may contain moving obstacles that furthermore might not be perfectly predictable (e.g., foreign-controlled robots). Navigating through an environment with moving obstacles requires a planner to have accurate models for predicting the obstacles' dynamics. Furthermore, because perfect prediction is not always possible, we must find other methods for dealing with the domain's uncertainty.

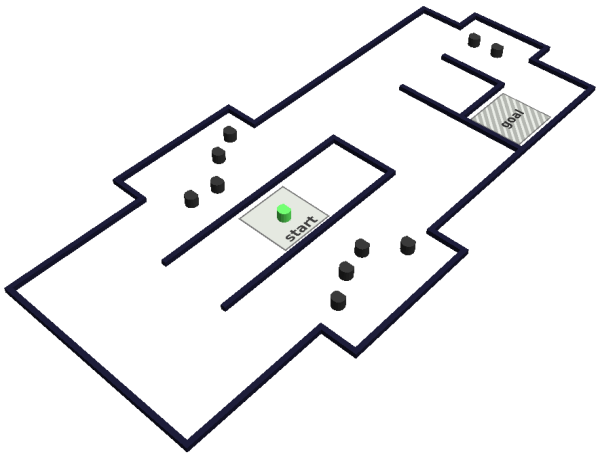


Figure 3.1: The "maze" navigation domain.

We implement the navigation domain in simulation, allowing us to define arbitrary motion planning problem instances with freely configurable stationary walls, moving obstacles, and a controllable degree of simulated domain execution uncertainty. Figure 3.1 shows a "maze" problem instance from the navigation domain, requiring a robot (light green cylinder) to find a trajectory from the start location to the goal area without colliding

with any moving obstacles (dark cylinders). Chapter 9 shows how to efficiently plan in this domain using *Variable Level-Of-Detail* planning.

3.1.2 Robot Soccer Domain

The *robot soccer* domain requires a robot to autonomously play a game of soccer against one or more robot opponents. Modeled after the RoboCup Small Size League [77], the robot can use its built-in ball actuators to dribble or kick the ball. Figure 3.2 shows a real-world RoboCup robot soccer game. In this thesis, we focus on single-robot planning problems within the robot soccer domain, such as one-vs-one, or one-vs-multiple game scenarios that require the controlled robot to out-dribble opponents and score a goal.

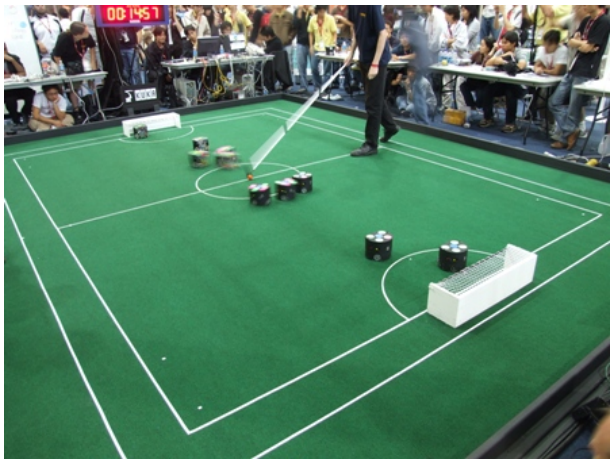


Figure 3.2: A game in the RoboCup Small Size League.

Besides the challenges in the previously described navigation domain, the robot soccer domain features multiple additional challenges:

- **Ball Manipulation**
  To effectively play soccer, a robot must be able to purposefully dribble the ball. As we explain in Chapter 2.4, manipulation of a passive body poses a difficult challenge for a motion planner. Furthermore, it requires an accurate model that can predict the ball's dynamics and its complex interactions with the robots and the playing surface.

- **Adversarial Robots**
  Robot soccer is uniquely challenging due its extremely adversarial nature. Executing a wrong action in real-time robot soccer is not only likely to delay progress toward

the goal state (of scoring a goal), but is in fact very likely to lead further away from it because the other team could take control of the ball and score a goal.

We implement the robot soccer domain both in our planner's simulation environment and on real robots, using the CMDragons multi-robot system (see Appendix A). Chapter 7 presents the integration of our planner into the CMDragons robot system and its evaluation.

3.1.3 Robot Minigolf Domain

Inspired by real-world minigolf, the *robot minigolf* domain requires a robot to putt a golf ball into a hole by purposefully bouncing it off other obstacle bodies. The positions of the ball, obstacles, and hole can all be freely configured. Figure 3.3 shows an example course in the minigolf domain. To achieve a hole-in-one, the robot (A) must find a solution similar to the one indicated by the arrows, bouncing the ball of the wooden bars (C) and into the U-shaped "hole" (B). The following challenges make the robot minigolf domain an ideal testbed for our planning approach:

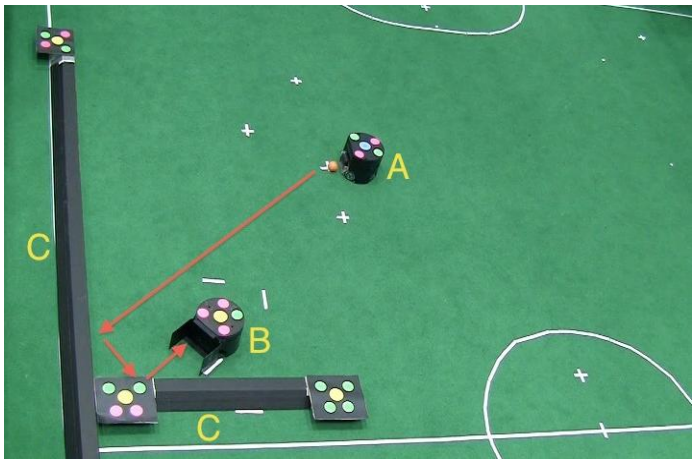


Figure 3.3: A course in the robot minigolf domain.

- **Prediction of Multi-Body Interactions**
  To play minigolf successfully, a motion planner requires an accurate predictive model of the ball's interactions with the environment. Chapter 8 uses the robot minigolf domain to demonstrate how automated model parameter optimization techniques can be used to improve execution success rates.

- **Prediction of Foreign-Controlled Moving Bodies**
  As an additional challenge, courses in the robot minigolf domain may contain moving obstacles that cannot be directly controlled by the robot. To succeed, the robot must be able to observe and extrapolate the motions of the obstacles and carefully time its execution.

We evaluate our planner's performance in the robot minigolf domain in simulation and on real robots, using the CMDragons system.

3.2 Application in Computer Animation

Although the primary target application of our work is robot motion control, much of our planning approach is also highly relevant to problems encountered in the world of computer animation.

Rigid body dynamics simulations are a popular tool for generating realistic looking multi-body motion sequences in computer animated movies and video games. After defining initial states for a set of rigid bodies, a physics engine can be used to automatically compute the forward simulation of Newtonian mechanics, realistically generating the physical motions and interactions of the bodies.

A problem commonly occurs however, when the animation should reach a particular outcome that is not achievable by the pure automatic forward simulation of a physics engine. For example, the goal could be that some of the rigid bodies end up in a particular final state that is unlikely to occur naturally. Additionally, some of these rigid bodies might have to behave actively during the animation and expose some desired visual behavior. Finally, a rigid body might even need to purposefully manipulate other rigid bodies (through the means of collisions) in order to achieve the goal state for the animation.

In order to satisfy such custom constraints on the animation's behavior, intermediate control actions (consisting of forces and torques) need to be applied to some of the rigid bodies. Manually finding a set of acceptable control actions, however, can pose a difficult and labor-intensive challenge. Oftentimes, a human animator may not know how to adjust the available control actions to achieve a particular desired outcome of the simulation. Even with aggressive trial and error, many domains may simply be too complex or too highly constrained to be controlled manually. More importantly, there are several applications which do not have the luxury of relying on the guidance of a human animator. For example, in video games that feature rigid bodies, the automatic generation of intelligent control actions for computer-controlled opponent bodies might be needed. Because the initial state

is subject to change (due to an evolving game-state and unpredictable player-input), an animator cannot manually adjust these controls in advance.

The planning approaches introduced in this thesis can be used to automatically solve difficult rigid body control problems. In many aspects, the planning problems encountered in computer animation are very similar to the planning problems in our robot domains. However, there exist a few significant differences:

- **No Sensory Noise and Better Domain Predictability**
  Computer animation domains do not need to deal with the difficulties of sensing and execution that are common in real-world robot domains. In computer animation tasks, the planner's internal state constitutes the true world state. An execution therefore simply consists of re-playing the state-sequence generated by the planner and consequently carries no execution uncertainty. A notable exception are real-time computer games where some of the system's state might change unexpectedly during execution (e.g., due to a human player's unpredictable control over a rigid body). In this case the problem becomes more similar to robot domains and will require methods for handling the effects of uncertainty, such as the ones we present in Chapters 7 and 9.

- **Actuation of Seemingly Passive Bodies**
  During execution in robot motion planning problems, the only control that a planner can exert is over *actively controlled* bodies (i.e., the robot itself). Passive bodies in the domain can – by definition – not be actively controlled, except through indirect means, such as collisions. In computer animation domains, however, *every* body is in fact actively controllable, because the planner has no real restrictions on exerting forces and torques on any simulated body during execution. Of course, to achieve realistic motion sequences involving interactions of many non-actuated bodies (e.g., a large number of falling dice), the bodies should *appear* to be passive. Nevertheless, in order to achieve a particular desired outcome of an animation sequence, it is often feasible to allow the planner to actively adjust the forces and torques of such *seemingly* passive bodies, as long as the resulting animation looks plausible to the target audience. This concept, also known as *physical plausibility*, is used frequently in computer animation by slightly perturbing and manipulating seemingly natural processes to achieve a specific desired behavior.

We introduce two computer animation domains that we test our approach on.

3.2.1 Many-Dice Domain

The objective in the *many-dice* domain is to control the fall of 400 dice, each initialized to a random position. The fall and the collisions of the dice are supposed to look physically plausible. However, the domain's goal condition requires that all dice land in a way that they form a particular logo as they are coming to a rest, a condition that is extremely unlikely to occur naturally. Figure 3.4 shows an initial state (left) and a goal state (right) of the many-dice domain. At the initial state, the dice are randomly shuffled and assigned positions above a grid of spherical obstacles. The goal is that the dice land in the bucket and form the "EG" logo of the Eurographics association. This domain is representative of many challenging computer animation tasks for two reasons:

- **Physically Plausible Control**
  The domain is challenging from a control standpoint. Although the planner is technically able to fully actuate each individual die, it must do so very intelligently and subtly in order to maintain the illusion of a natural looking free-fall and plausible inter-body collisions.
- **Large Number of Bodies**
  The many-dice domain makes a very interesting planning problem due to its massive scale. Unlike most typical robot domains where a planner controls the actions of a single robot, the many-dice domain constitutes a joint control problem with a large number of controllable degrees of freedom, thus requiring very efficient planning methods.

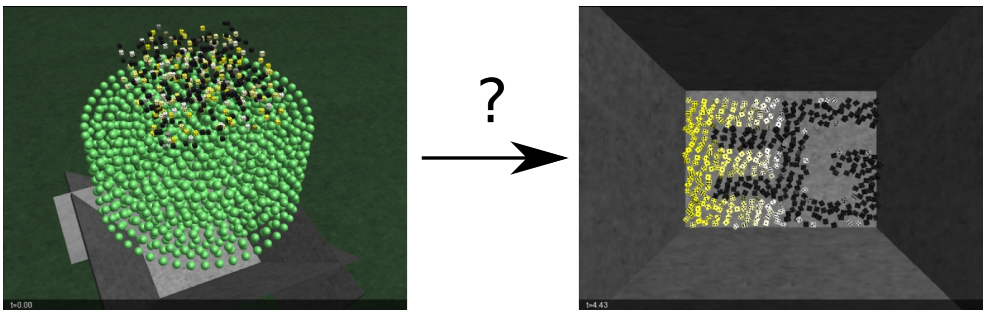


Figure 3.4: An initial and a goal state in the many-dice domain.

3.2.2 Pool Table Domain

The *pool table* domain requires the planner to solve challenging "trick-shot" problems, generating sequences of a single cue ball interacting with multiple pool balls to create a

desired outcome. Figure 3.5 shows an example problem in the pool table domain where the goal is to actuate the cue ball (A) in such a way that it does not touch the three balls near the mid-pocket (B), but hits the solid yellow ball (C), which in turn will roll and bump the solid blue ball (D) into its pocket. From a planning standpoint, the pool table domain is challenging because of the following reason:

- **Plausibility of Ball Trajectories**
  Although the planner technically has full control over each ball's actuation, the resulting motion sequence should give the illusion that only the cue-ball was given an initial impulse with an extremely correctly estimated amount of force and spin. Therefore, any additional actuation exerted on the cue-ball and its interacting balls needs to be applied in a very constrained manner to not appear overly unnatural. In Chapters 4 and 5 we introduce models that allow such accurate behavioral control over the bodies' actuations.

- **Chains of Collisions**
  The pool table domain is interesting because it requires chains of collisions. For example, in Figure 3.5, the cue ball (A) should hit the yellow ball (C) which in turn should hit the blue ball (D). Such chains of collisions are extremely fragile because even a small error at any of the collision points can destroy the success of the remaining sequence (also see Chapter 2.4 for a more in-depth discussion of this challenge). Chapter 6 demonstrates how to chain behavioral control models, allowing robust and efficient planning in the pool table domain.

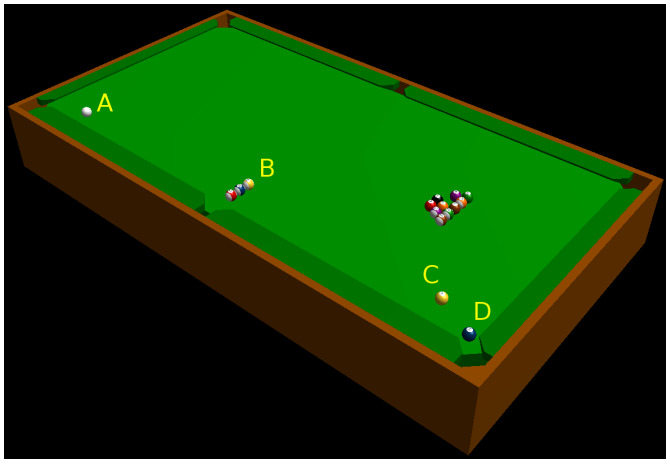


Figure 3.5: A problem in the pool table domain.

3.3 Chapter Summary

We have introduced robot control and computer animation as two possible applications for our physics-based motion planning approaches. To evaluate our work in robotics applications, we have introduced the *navigation*, *robot soccer*, and *robot minigolf* domains. To evaluate our approach's applicability to computer animation tasks, we have introduced the *many-dice* and *pool table* domains. We have discussed the unique challenges for each introduced domain, collectively forming a comprehensive testbed for our motion planning approach.

Chapter 4

Non-Deterministic Skills and Tactics

In Chapters 2 and 3, we have formally introduced the physics-based planning model and several challenging planning domains. As we have discussed in Section 2.4, automatically finding solutions in large continuous action spaces can be difficult, especially if the problem requires purposeful manipulations of passive bodies, contains behavioral constraints on the intermediate states of the solution sequence, or contains foreign-controlled bodies.

Exhaustively searching the action space for solutions is infeasible due to its continuous nature. Instead, our planning approach uses *sampling* to select actions from the continuous action set A. However, using an uninformed approach (e.g., purely random uniform sampling from the set A), is unlikely to find valid planning solutions within a reasonable amount of time, even for simple domains. Consequently, we need to find methods that are able to plan more efficiently in the physics-based space. In this chapter, we introduce non-deterministic *Skills* and *Tactics* as an intelligent model for sampling actions.

4.1 Skills and Tactics

In order to increase planning efficiency, we focus on the problem of reducing the size of the searchable action space. In particular, we introduce a sampling-based action selection model that guides the planner by intelligently selecting actions that are specifically aimed to bring the planner closer to its goal state.

Our model is based on the idea of traditional reactive robot controllers, as they are for example used in robot soccer. Among the many reactive behavioral control architectures, e.g., [8, 25, 56], we choose the *Skills, Tactics, and Plays (STP)* architecture, as it has

effectively been used in real-time adversarial robot domains [13] (also see Section 10.6 for a review of robot control architectures). In our work, each controllable rigid body is synonymous to an agent in STP. In STP, a *Play* captures the behavior assignments of a group of multiple agents. A single agent's behavior is modeled as a reactive *Tactic*, representing a finite state machine (FSM) of lower-level *Skills*, which act as pre-programmed, reactive control blocks, each implementing an intelligent state-dependent control policy. For the scope of this thesis, we assume fixed role assignments for all the controlled bodies in the domain, and therefore we do not require the multi-agent "Play" component of STP.

Traditionally, STP's Skills and Tactics run online during execution, as purely reactive controllers, without any planning at the Tactics level [18]. In our work however, a planner uses Skills and Tactics as an action sampling model. Instead of being a reactive controller as in traditional STP, the Tactics' and Skills' new role is to guide the planner's search by imposing constraints on the searchable action space. For this purpose we introduce and formalize a new, probabilistic version of Skills and Tactics that uses random sampling to choose from a set of possible actions.

4.2 Sampling-Based Skills

The purpose of a Skill is to act as a non-deterministic controller that generates actions through the means of random sampling in combination with some domain knowledge.

Definition 4.2.1 (Skill)**.** We define a Skill $s = \langle C, D, f \rangle$ where

- C is a set of configuration constants,
- D is a set of sampling distributions,
- f is the *operator* function $f : \langle C, D, V, x, a \rangle \rightarrow \langle a', V', b \rangle$, where a and a' are actions (consisting of force/torque sub-actions as introduced in Chapter 2), V and V' are sets of variables and b is a boolean *busy flag*.

Figure 4.1 shows the general structure of a sampling-based Skill. The Skill's operator function f is the part of the Skill that generates control actions, consisting of force/torque sub-actions (see Chapter 2). Part of f's input are the state of the world x and a set of actions a that it can modify. f produces the modified set of control actions a' and the boolean "busy flag" b, to be explained in an example later in this section and in Chapter 5.

The set of constants C contains any configuration parameters that define how the Skill should operate. The precise contents of C depend on the actual Skill. For example, C

could contain the index i of the rigid body r_i that this Skill should generate actions for, or it could contain values that limit the maximum velocity of the controlled body. The values of these configuration parameters remain the same throughout the life of the Skill (thus called *constants*).

The set of variables V also allows storage of arbitrary Skill-dependent parameters, similar to C, except that V is stored externally to the Skill and that the operator function f can change the contents of V by producing a modified set of variables V' as output. As we will show later in Section 5.3, V can be used to maintain state information between multiple successive calls of the same Skill.

Finally, the set of sampling distributions D is the component that makes the Skill non-deterministic because the operator function f can randomly sample values from these distributions and use them for the computation of control actions. The concrete nature and number of sampling distributions stored in D is dependent on the actual Skill. For instance, in the shortly following example Skill, D contains a sampling distribution for randomly selecting a point to drive toward. The fact that f can randomly sample from the distributions in D means that multiple calls to f with the same input arguments can each non-deterministically produce a slightly different set of actions a' where the degree of random perturbation depends on the underlying distributions D.

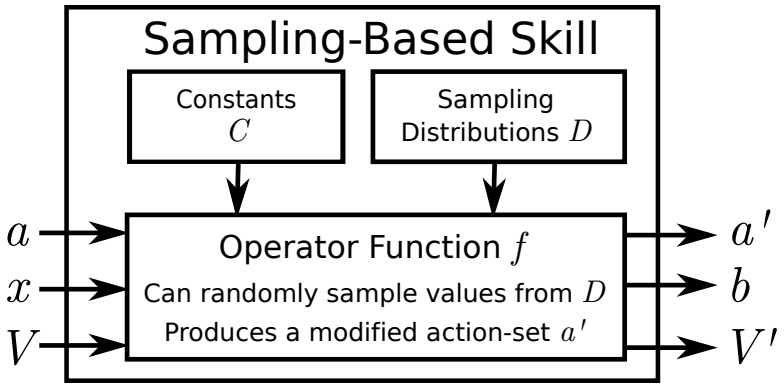


Figure 4.1: Structure of a sampling-based Skill.

An Example Skill

Algorithm 2 shows the internals of an example Skill, including its constants C, its sampling distributions D, and its operator function f. The purpose of this example Skill is to generate control actions that will move a rigid body toward a randomly selected point. The set of constants C contains all the parameters that configure the Skill's behavior,

Algorithm 2: "Drive to Sampled Target" Example Skill

Constants C:

- Index of controlled rigid body: i
- Maximum linear body velocity and acceleration: max_vel, max_acc
- Maximum angular body velocity and acceleration: max_ang_vel, max_ang_acc

Sampling Distributions D:

- Distribution of possible navigation target locations: locations

Function $f(C, D, V, x, a)$:

$a' \leftarrow a$;
$V' \leftarrow V$;
$i \leftarrow C.i$;
if target_location $\notin V$ **then**
  target_location $\leftarrow$ `SampleFromDistribution`(D.locations);
  $V' \leftarrow V' \cup \{\text{target\_location}\}$;
target_heading $\leftarrow V'$.target_location $- x.\hat{r}_i.\alpha$;
target_heading $\leftarrow \frac{\text{target\_heading}}{\|\text{target\_heading}\|}$;

```
// Motion control functions [15] compute the necessary
   forces/torques to turn/drive towards the target location:
```

$a'.\hat{a}_i.force \leftarrow$ `DriveToLoc`($x.\hat{r}_i$, target_location, C.max_vel, C.max_acc);
$a'.\hat{a}_i.torque \leftarrow$
`TurnToHeading`($x.\hat{r}_i$, target_heading, C.max_ang_vel, C.max_ang_acc);

if `ReachedHeading`($x.\hat{r}_i$, target_heading) **and** `ReachedLocation`($x.\hat{r}_i$, target_location) **then**

```
  // Body has reached the target
```

  $b \leftarrow$ *false*;
else

```
  // Body still navigating to the same target
```

  $b \leftarrow$ *true*;

return $\langle a', V', b \rangle$;

such as the index i of the body r_i that the Skill is supposed to control, and limits on the maximum velocity and acceleration that should be achieved by the body. The set D of this example Skill contains a single sampling distribution (locations) from which random target locations can be sampled. The precise configuration of the distribution depends on the domain that the Skill operates in. For example, if the domain is limited to a rectangular area of known size, then it makes sense to configure locations to uniformly sample random points from that particular area.

The Skill's operator function f performs the task of generating the actual control actions for the rigid body. When called, the Skill first checks whether a previously sampled target location already exists in the set of variables V. If not, the Skill non-deterministically samples a new target location from the location distribution (locations) and adds it to the modified set of variables V'. Next, the Skill computes a unit vector, pointing from the controlled body's current position toward the target location, representing the desired target heading of the controlled body. The target location and heading are then passed into two low-level motion control functions that compute the required low-level forces and torques to move the body toward target_location and target_heading. If the controlled body has reached the target location and heading then the busy flag b is set to false, indicating to the Skill's caller (i.e., the planner, as we will introduce in Chapter 5), that on its next call, the Skill should choose a new random target and therefore act non-deterministically. If, on the other hand, the controlled body has not yet reached its target, then b is set to true, indicating to the Skill's caller that the Skill is still in the process of moving the body toward its current target, and therefore should continue to do so when called the next time, without selecting a new target. As we will further explain in Chapter 5, it is up to the Skill's external caller to decide what to do with the modified set of variables V' and with the value of the busy flag b which are returned by the function f. For now, let us assume that, if b is true, then the output V' will be passed back into f as V during the successive call of the Skill. Otherwise, V will be set to the empty set for the next call of f.

The "Drive to Sampled Target" Skill of Figure 2 is only one particular example of a non-deterministic Skill. Appendix C shows several additional example Skills, each implementing unique tasks via their operator function f, and each using different sets of configuration constants C, sampling distributions D, and each storing different types of variables in the set V.

Skills as Reusable Behavior Building Blocks

The Skill's operator function f, and its parameters D and C all need to be manually defined and therefore assume some knowledge about the domain and its physical properties, such as the reasonable ranges of applicable forces and torques. Typically, a single Skill implements

a particular control objective such as "drive towards a sampling-based location," "push some target rigid body towards a sampling-based target region," or "turn a sampling-based amount." Designing these actual Skills does take some programmatic effort and it can be argued that some Skills are fairly domain-dependent. However, many Skills can be used as template-like "building blocks" that apply to a variety of domains. Given a library of Skills, it is possible to rapidly create new Tactics (described in the following section) that are well-suited for solving a particular domain.

4.3 Non-Deterministic Tactics

So far, we have introduced sampling-based Skills as a non-deterministic action sampling model for simple control tasks. Next, we introduce non-deterministic Tactics as a method for constructing more sophisticated control behaviors, consisting of multiple Skills.

Definition 4.3.1 (Tactic)**.** We define a Tactic $\tau = \langle S, \Pi, \sigma_{\text{init}} \rangle$ where

- S: a set of k Skills $\langle s_1, \ldots, s_k \rangle$,
- Π: the function $\Pi : \langle \sigma, \sigma', x \rangle \rightarrow p$, computing p, the transition probability from s_σ to $s_{\sigma'}$, depending on the world state x,
- σ_{init}: the index of the initially active Skill $s_{\sigma_{\text{init}}} \in S$.

A Tactic is composed of a set of Skills and a set of possible transitions between these Skills. This model is similar to a classic Non-deterministic Finite State Machine (with Skills corresponding to FSM states), except that in our case, any transition probability p can change dynamically during the life of the Tactic because it is generated by the function $\Pi : \langle \sigma, \sigma', x \rangle \rightarrow p$, thus depending on the world state x.

Definition 4.3.2 (Tactic State Transition Function)**.** To actually perform non-deterministic state transitions between Skills, using the probabilities generated by Π, we define the transition function

$$g : \langle S, \Pi, \sigma, x \rangle \rightarrow \sigma',$$

taking a Tactic's set of Skills S, its probability function Π, an index σ pointing to a Skill $s_\sigma \in S$, and a state of the world x to produce a new index σ' pointing to a Skill $s_{\sigma'}$. Algorithm 3 shows how g computes these state transitions probabilistically.

Figure 4.2 shows an example diagram of a Tactic. In this case, the Tactic contains four Skills $(s_1, \ldots, s_4)$. There are several possible types of transitions between these Skills:

Algorithm 3: Transition Function g

```
Input: S = {s_1, ..., s_k}, Π, σ, x
probSum ← 0;
for j ← 1 to k do
    probSum ← probSum + Π(σ, j, x);
if probSum > 0 then
    rndVal ← Sample(0, Max(1, probSum));
    tempSum ← 0;
    for j ← 1 to k do
        tempSum ← tempSum + Π(σ, j, x);
        if tempSum > rndVal then
            return j;
return σ;
```

- **Deterministic Transitions**
  The transition between Skills s_1 and s_2 in Figure 4.2 is deterministic. In the deterministic case, the probability function $\Pi(1, 2, x)$ always returns 1.0, independent of the state x, thus always enforcing a transition from Skill s_1 to Skill s_2. Consequently, in the deterministic case, all other outgoing transitions from s_1 will generate a zero-probability, that is, $\forall \sigma' \neq 2 : \Pi(1, \sigma', x) = 0$.

- **Non-Deterministic Transitions**
  The outgoing transitions from the Skill s_2 in Figure 4.2 are non-deterministic. That is, when executing the transition function on s_2, we will either end up in Skill s_3 with a 0.4 probability (because $\Pi(2, 3, x) = 0.4$) or we will remain in Skill s_2 with a 0.6 probability (because $\Pi(2, 3, x) = 0.6$). By being modeled in this non-deterministic fashion, each execution of the Tactic can result in a different sequence of transitions between Skills, thus covering different parts of the search space. Similar to the deterministic case, the transition probabilities in this example are constant, and do not change based on the system state x.

- **State-Dependent Transitions**
  The transition from Skill s_3 to Skill s_4 in Figure 4.2 is state-dependent. The probability function $\Pi(3, 4, x)$ returns 1.0 if *some* property within the world state x is met, and it returns 0.0 otherwise. In the latter case, all outgoing connections from s_3 are in fact zero, which implies a self-transition based on the definition of g, as shown earlier in Algorithm 3. Although not shown specifically in Figure 4.2, it is furthermore possible to combine state-dependent transitions with non-deterministic transitions, i.e., by having multiple non-zero outgoing transition probabilities whose concrete values depend on the state x. Also note, that we do not have to worry that

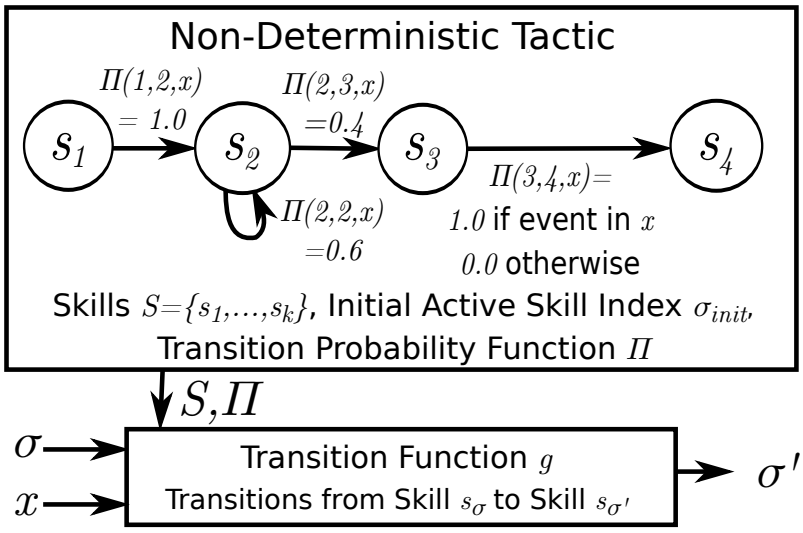


Figure 4.2: Example of a non-deterministic Tactic.

all currently applicable probabilities generated by $\Pi(\sigma, \sigma', x)$ for a particular current σ need to sum to one, because the function g samples uniformly over the sum of all applicable probabilities, effectively auto-normalizing them.

The concrete values for defining a Tactic τ (i.e., the set of Skills S, the probability function Π and the index of the initial Skill σ_{init}) are all provided by the user and are domain-dependent.

Busy Skills Prevent State Transitions

Note that any of the above state transitions are only performed if the currently active Skill is not busy (i.e., the Skill's operator function f returned $b = \mathit{false}$ as its busy flag on its latest call). If, on the other hand, the Skill is currently busy (i.e., $b = \mathit{true}$), then no state transition is performed (i.e., g is never called), and we will remain in the same Skill.

An Example Tactic

Figure 4.3 shows an example Tactic implementing a robot soccer behavior. The Tactic contains Skills to drive toward the ball ("drive to ball"), push the ball toward a sampling-based location ("sampled dribble"), kick the ball toward a random point in the goal ("sampled goalkick"), or kick the ball by a short distance toward a randomly selected point ("sampled

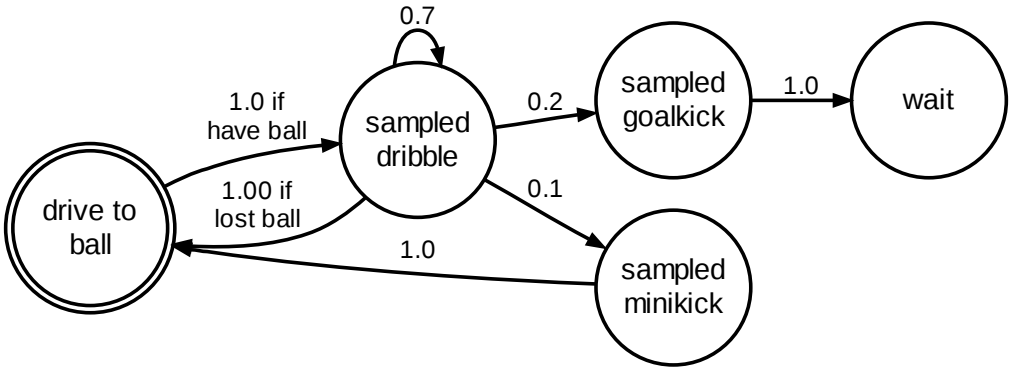


Figure 4.3: A robot soccer example Tactic.

minikick").

This Tactic contains a combination of state-dependent, non-deterministic, and deterministic transitions. For example, the transition from "drive to ball" to "sampled dribble" depends on the state of the world x, because it is only performed if the robot has reached the ball. The outgoing transitions of the "sampled dribble" Skill are non-deterministic: remaining within the "sampled dribble" Skill with a 0.7 probability, or switching into one of the kick Skills with the remaining probability. The transition from the "sampled minikick" to the "drive to ball" Skill is deterministic, letting the controlled robot immediately work on re-obtaining the ball after it has kicked it.

Deterministic Skills and Tactics

It should be noted that our introduced Skills and Tactics model is a superset that contains their traditional deterministic versions. In our formal model, we can define a *deterministic Tactic* as a Tactic which cannot have multiple outgoing state transitions from any s_σ with probability greater than 0 for any x. Similarly, we define a *deterministic Skill* as one where there is a single deterministic mapping from states to actions. Therefore, in a deterministic Skill, $D = \emptyset$, because no sampling distributions are needed for deterministic decision making.

4.4 Chapter Summary

In this chapter, we have introduced non-deterministic Skills and Tactics as an intelligent action sampling-model. In the following chapter, we introduce concrete planning algorithms

that rely on non-deterministic Skills and Tactics during search.

Chapter 5

Efficient Planning via Skills and Tactics

In Chapter 4 we have introduced non-deterministic Skills and Tactics as an action sampling model. In this chapter, we introduce and compare two concrete Skills and Tactics-driven randomized physics-based planning algorithms, namely Behavioral Kinodynamic Rapidly-Exploring Random Trees (BK-RRT) and Behavioral Kinodynamic Balanced Growth Trees (BK-BGT).

5.1 Skills and Tactics in the State Space and Domain

Definition 5.1.1 (State Space with Tactical State)**.** In order to use Skills and Tactics as a sampling model for our planner, we need to store their variable parameters as part of the state space. More formally, we redefine a state $x \in X$ as:

$$x = \langle\ t,\ \ \hat{r}_1, \ldots, \hat{r}_n,\ \ \sigma_1, \ldots, \sigma_h,\ \ V_1, \ldots, V_h,\ \ b_\wedge,\ \ b_1, \ldots, b_h\ \rangle,$$

including the indices to the currently active Skill for each Tactic $(\sigma_1, \ldots, \sigma_h)$, as well as the internal variable state V of the currently active Skill for each of the h Tactics. Finally, $b_\wedge$, and $b_1, \ldots, b_h$ are all boolean values, indicating whether the entire state x and/or any of its active Skills s_{σ_i} are currently "busy", to be further explained later in this chapter. The number of Tactics h is dependent on the number and types of bodies within the domain. Each actively controlled body has a corresponding non-deterministic Tactic consisting of

sampling-based Skills that generate the body's actions. Additionally, we construct reactive, deterministic Tactics with deterministic Skills to act as prediction models for each foreign-controlled body. Even if a foreign-controlled body's exact Tactic is not known (which is typically the case for adversarial robots), it may still be useful to model its roughly expected behavior rather than assuming it is static. Finally, passive bodies in the domain are non-actuated and do not require a Tactic.

In Section 2.1.2, we defined a general physics-based planning domain to consist of a global gravity vector, G, the bodies' immutable parameters $\bar{r}_1 \dots \bar{r}_n$, and the action space A that defines the set of applicable actions. Note, however, that the purpose of Skills and Tactics also is to constrain the set of selectable actions. Therefore, we redefine d, replacing A with the set of Tactics that act as a sampling-based method to constrain the action space.

Definition 5.1.2 (Tactics-Based Domain)**.** A Tactics-based planning domain d is defined as:

$$d = \langle G, \bar{r}_1 \dots \bar{r}_n, \tau_1, \dots, \tau_h \rangle,$$

where h is the total number of Tactics.

5.2 A Tactics-Based Planning Example

Before introducing our actual Tactics-based planning algorithms in full detail, we first present a simple example to illustrate how a physics-based planner can use Skills and Tactics to perform its search.

Figure 5.1 shows a Tactics-based planning example in a simple robot soccer domain. The Tactic for the controlled body (Figure 5.1 (a)) models the behavior of a robot soccer "attacker". Figure 5.1 (b) shows the initial state x_{init}. The goal of the domain is to let the robot body (dark cut-off cylinder) deliver the ball (orange circle) into the goal box (shaded rectangle). In the initial state, the current time index t is 0, and the Tactic is in its initial "get ball" Skill.

Starting from the initial state, the planning algorithm constructs a tree of succeeding states, with the objective that one of the tree's nodes happens to be a valid goal state. The basic algorithm is conceptually similar to the BasicPlanner algorithm presented in Section 2.5, except that its states now include the Tactics state (particularly, the index of the currently active Skill), and its actions are constrained to the ones generated by the Tactic's Skills. To construct the search tree, the algorithm repeatedly selects a source node from the existing tree and generates a new succeeding child node. Figures 5.1 (b) and (c) demonstrate the growth of the first branch in the search tree. Given the initial state (Figure 5.1 (b)), the

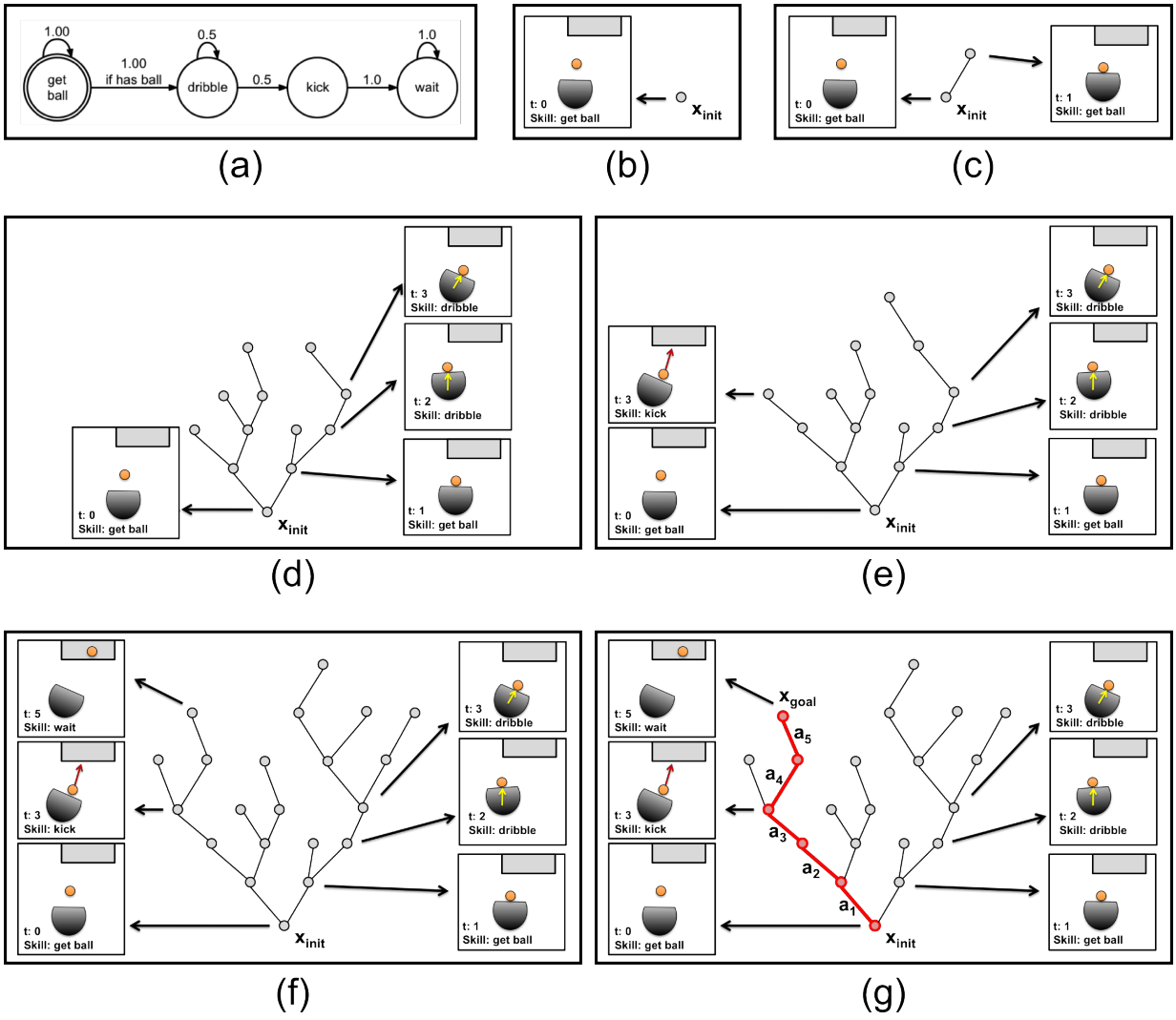


Figure 5.1: A Tactics-based planning example.

planner calls the operator function f of the currently active Skill (given by the source node's state x_{init}). Taking the current state of the world x_{init} as input, the Skill's operator function f then computes an action, consisting of forces and torques, applicable to the controlled robot body. The planner then provides this action, along with the current state of the world, to the physics engine's state transition function e to compute the succeeding state (where $t = 1$). The action is stored within the branch that connects the source node to the succeeding node. Furthermore, from the source state x_{init}, the planner invokes the Tactics state transition function g to compute the successive Tactics state. In this example, the Tactic remains in the "get ball" Skill, because the robot in x_{init} (Figure 5.1 (a)) has not met the condition of reaching the ball (defined in the Tactics model in Figure 5.1 (a)). The resulting state (including the body states, the Tactic state, and the time-index t) is added to the tree as a child of x_{init}.

The planner then repeats this tree growth mechanism, each time selecting *some* source node from the existing tree and computing a new successor state. The actual method of node selection is a key performance factor for the planning algorithm, and will be presented in the following sections. For now, let us assume that the planner selects a random node. Figure 5.1 (d) shows the tree after several iterations, highlighting two new states, where the Tactic now happens to be in the "dribble" Skill which non-deterministically generates actions that control the robot to dribble the ball into some sampling-based direction. Note, however, that also the Tactic itself (Figure 5.1 (a)) is non-deterministic, allowing transitions from "dribble" to "kick" with a 0.5 probability. Figure 5.1 (e) highlights this non-determinism, showing two states, each with time-index $t = 3$, but each with a different active Skill (namely one in "kick" and one in "dribble").

The tree growth continues until, in one of the branches, the robot performs a successful kick toward the goal, and the ball ends up inside of the goal-box (Figure 5.1 (f)). Once such a goal state is found, the planner simply back-traces the branches toward the root and returns the actions contained within them as the final solution sequence (Figure 5.1 (g)).

5.3 BK-RRT Algorithm

We are now ready to introduce concrete physics-based planning algorithms that use Skills and Tactics as their action sampling model. We present two algorithms that share the same fundamental forward-planning loop, but differ in the way they control the growth of the search tree. We name these two algorithms as Behavioral Kinodynamic Rapidly-Exploring Random Trees (BK-RRT) and Behavioral Kinodynamic Balanced Growth Trees (BK-BGT). First, in this section, we present the BK-RRT algorithm. We follow with

BK-BGT in Section 5.4.

Algorithm 4 shows BK-RRT. In terms of notation, we use the dot symbol to indicate access to a member of a tuple (e.g., $x.t$ refers to the t item stored in x). Before we begin to explain the algorithm, we need to define several assumptions about the initial state x_{init}. Recall that a state in our model is defined as:

$$x = \langle\ t,\ \ \hat{r}_1, \ldots, \hat{r}_n,\ \ \sigma_1, \ldots, \sigma_h,\ \ V_1, \ldots, V_h,\ \ b_\wedge,\ \ b_1, \ldots, b_h\ \rangle.$$

For the initial state x_{init}, we assume that the active Skill index for each Tactic is set to the initial Skill index as defined in the corresponding Tactic (i.e., $\sigma_1 \leftarrow \tau_1.\sigma_{\text{init}},\ \ldots,\ \sigma_h \leftarrow \tau_h.\sigma_{\text{init}}$). Furthermore, in x_{init}, all sets of variables are empty (i.e., $V_1 \leftarrow \emptyset,\ \ldots,\ V_h \leftarrow \emptyset$) and all busy flags are set to *false* (i.e., $b_\wedge \leftarrow$ *false* and $b_1 \leftarrow$ *false*, $\ldots$, $b_h \leftarrow$ *false*). Finally, for convention throughout this thesis, we also assume that $x_{\text{init}}.t \leftarrow 0$.

Having defined these requirements for the initial state, we now explain BK-RRT as shown in Algorithm 4. First, we initialize the search with a tree containing the initial state $x_{\text{init}} \in X$, and set the boolean variable busy to *false*. We then enter the main planning loop, which runs for a predefined domain-dependent maximum number of search iterations z, if no solution is found earlier. On each iteration, assuming busy is *false* (which is the case initially), the algorithm calls the `SelectNodeRRT` algorithm to select a node x from the existing tree which it will expand from. The difference between BK-RRT and BK-BGT lies precisely in this node selection function.

BK-RRT selects nodes similarly to the traditional Rapidly-Exploring Random Trees (RRT) search [57]. Algorithm 5 shows the `SelectNodeRRT` function used in our BK-RRT algorithm. The function `SampleRandomState` uses an internal probability distribution to provide a sample y, taken from the sampling space Y that is some predefined subspace of X. The function then iterates over all states in the tree, to find the nearest neighbor to the random sample y by using some distance function `ComputeDistance`. Note, that only "non-busy" nodes (i.e., $x.b_\wedge =$ *false*) are considered during the nearest neighbor search. The precise purpose of the "busy" flags will be explained in Section 5.3.1; for now it suffices to say that we can only grow branches from "non-busy" nodes. The function then returns a tuple with the nearest neighbor x_{nearest} and the sample y. As with traditional RRT, it is important that the sampling space Y, the underlying probability distribution, and especially the distance function are all carefully chosen to match the domain. We postpone the discussion about selecting these parameters for Section 5.3.4 and continue with the explanation of the remaining BK-RRT algorithm.

After selection of the source node x, the algorithm then calls `TacticsDrivenPropagate` with the purpose to generate a succeeding state x' from the source state x. Algorithm 6 shows this Tactics-driven propagation function. The function begins by instantiating the succeeding state x', initially as a copy of x. Furthermore, it generates an empty action

Algorithm 4: BK-RRT

Input: Domain: d, Initial state: x_{init}, set of goal states: X_{goal}, RRT sampling space: Y, timestep: Δt, validation function: `Validate`, max iterations: z.

```
tree ← NewEmptyTree();
tree.AddNode(x_init);
busy ← false;
for iter ← 1 to z do
    if busy = true then
        x ← x';
    else
        ⟨x, y⟩ ← SelectNodeRRT(tree, Y); // See Alg. 5
    ⟨x', L⟩ ← TacticsDrivenPropagate(x, y, Δt, d); // See Alg. 6
    busy ← x'.b_∧;
    if Validate(x', L) then
        tree.AddNode(x');
        tree.AddEdge(x, x', a);
        if x' ∈ X_goal then
            return TraceBack(x', tree);
    else
        if busy = true then
            RollBack(x, tree); // See Alg. 7
        busy ← false;
return Failed;
```

$a \in A$, containing only zero-filled vectors for all force/torque pairs of all sub-actions. Next, it iterates over each Tactic and, assuming the Tactic's active Skill was not marked as "busy" ($x.b_i = \mathit{false}$), performs its state transition by calling g. If the Skill was marked as "busy" then no state transition is performed and its variables are obtained from its previous state. After storing the new Tactics state σ' into x', the algorithm then calls the active Skill's operator function f.

Note, that the contents of the variables V passed into f includes the random RRT sample y because it was added earlier ($V \leftarrow \{y\}$), thus allowing the Skill to make use of y. As we will further explain in Section 5.3.3, this in fact implies, that the traditional RRT algorithm is actually a subset of the introduced BK-RRT algorithm.

Each call to the operator f returns a tuple including the boolean "busy flag" value b_i. As we will explain in more detail in Section 5.3.1, the purpose of this boolean flag is to let a Skill report that it is currently executing a linear control task that does not require any

Algorithm 5: SelectNodeRRT

```
Input: Tree: tree, RRT sampling space: Y
y ← SampleRandomState(Y);
x_nearest ← ∅;
mindist ← ∞;
foreach x ∈ tree do
    if x.b_∧ = false then
        dist ← ComputeDistance(x, y);
        if dist < mindist then
            mindist ← dist;
            x_nearest ← x;
return ⟨x_nearest, y⟩;
```

branching because its action selection is currently deterministic. Furthermore, because b_i is stored into x', it signals to the BK-RRT algorithm to not perform any state transitions on the Tactic τ_i that has a currently busy Skill (by not calling g if $x.b_i = \mathit{false}$). The global flag $x'.b_\wedge$ is set to the logical *and* of all the individual Skill's b_i flags.

The algorithm is now ready to invoke the physics engine by calling the simulation function e, using as input the rigid body states $\hat{r}_1, \ldots, \hat{r}_n$ contained in the source state x, the forces and torques defined by a, the global gravity vector G, the immutable rigid body parameters (contained in the domain description d), and the simulation timestep Δt. The physics engine returns a new set of rigid body states that are stored into the new state x', and the list of collisions L.

After returning from the `TacticsDrivenPropagate` function, the BK-RRT algorithm (Algorithm 4) then stores the busy flag $x'.b_\wedge$ into the local busyvariable and moves on by calling the `Validate` function (see Definition 2.1.8) to check whether the new state x' should be considered valid, based on the state itself and on the list of collisions L that occurred during forward simulation from x to x'.

If the succeeding state x' is determined to be invalid, then the algorithm invokes the `RollBack` function (see Algorithm 7) if the busy flag is currently set to true. The purpose of this RollBack function is explained in Section 5.3.2, following the discussion of the busy flag (see Section 5.3.1).

If successfully validated, the algorithm adds x' to the search tree as a child of its source node x and furthermore stores the action within the branch. The complete loop of node selection and tree expansion is repeated until the algorithm either reaches a goal state (i.e., $x' \in X_{\text{goal}}$), or until it reaches the maximum allowed number of iterations z, at which point

Algorithm 6: TacticsDrivenPropagate

Input: Source state: x, random sample: y, timestep: Δt, domain: d.

$x' \leftarrow x$;
$\langle \tau_1, \ldots, \tau_h \rangle \leftarrow d.\langle \tau_1, \ldots, \tau_h \rangle$;
$a \leftarrow [0, \ldots, 0]$;
for $i \leftarrow 1$ **to** h **do** // Iterate over all Tactics
  $\langle S = \langle s_1, \ldots, s_k \rangle, \Pi, \sigma_{\text{init}} \rangle \leftarrow \tau_i$;
  if $x.b_i = \mathit{false}$ **then** // Check whether τ_i is busy
    $\sigma' \leftarrow g(S, \Pi, x.\sigma_i, x)$; // Compute Skill transition
    $V \leftarrow \{y\}$; // Make y available to Skill's variables
  else
    $\sigma' \leftarrow x.\sigma_i$; // τ_i is busy; maintain previous Skill
    $V \leftarrow x.V_i$; // Maintain previous variables
  $\langle C, D, f \rangle \leftarrow s_{\sigma'}$;
  $\langle a, V, b_i \rangle \leftarrow f(C, D, V, x, a)$; // Apply Skill
  $x'.\sigma_i \leftarrow \sigma'$;
  $x'.V_i \leftarrow V$;
  $x'.b_i \leftarrow b_i$;
$x'.b_\wedge \leftarrow b_1 \wedge b_2 \wedge \cdots \wedge b_h$; // Compute global busy flag
$\langle x'.\langle \hat{r}_1, \ldots, \hat{r}_n \rangle, L \rangle \leftarrow e(x.\langle \hat{r}_1, \ldots, \hat{r}_n \rangle, a, d.G, d.\langle \bar{r}_1, \ldots, \bar{r}_n \rangle, \Delta t)$; // Transition
$x'.t \leftarrow x.t + \Delta t$;
return $\langle x', L \rangle$;

the search returns failure. Once a goal state is reached, the algorithm simply traces back the chain of states and actions from the goal state to the root of the tree and returns it in reverse order as the final solution sequence in the form $\langle x_{\text{init}}, a_1, x_2, a_2, x_3, \ldots, x_{\text{goal}} \rangle$.

5.3.1 Preventing State Duplications: the Busy Flag

The BK-RRT algorithm makes use of the "busy flag" b_i as returned by the currently active Skill's operator function f for each Tactic (see Algorithm 6). The purpose of this boolean flag is to let a Skill signal that it is currently behaving deterministically (even if it is a non-deterministic Skill), meaning that its function f would always make exactly the same modifications to the action a if we would call it multiple times with the same current input arguments. Note, however, that we might have multiple Tactics that contain non-deterministic Skills (e.g., each controlling a separate controlled body). We should only mark an entire state as busy if all the currently active Skills among all Tactics are currently behaving deterministically. Therefore, the algorithm computes the global busy flag $b_\wedge$ to

Algorithm 7: RollBack

Input: State: x, Tree: tree.
while $x.b_{\wedge} = true$ **do**
    $x_{\text{parent}} \leftarrow$ tree.GetParentNode(x);
    tree.DeleteNode(x);
    $x \leftarrow x_{\text{parent}}$;
return;

be the logical *and* of all the individual Skill's busy flags.

A busy state x implies that there currently is no potential for *search*, because none of the currently active Skills can currently perform any random sampling over *different* possible actions. Consequently, it would be a waste of resources if the planning algorithm would grow more than one outgoing branch from the node x if the Skill is currently busy, because each outgoing branch would lead to exactly the same succeeding state x', cluttering the search tree with multiple copies of the same state.

The BK-RRT algorithm does several things to prevent such unnecessary duplication in states. First, the RRT node selection function (see Algorithm 5) ensures that only non-busy nodes are selected to have branches grown from them, because only non-busy nodes are actually decision points in the search. Second, whenever a new busy state is generated, the BK-RRT planning loop (see Algorithm 4) will extend its next iteration from x' instead of invoking the RRT node selection function, thus forcing a continued, greedy single branch expansion for as long as all Skills report that they are busy or until an invalid state is generated.

In order to not "get stuck" indefinitely in a greedy branch expansion loop, it is expected that all Skills will eventually "time out" and report a non-busy flag. Alternatively, one could limit the maximum number of repeated greedy expansions in the BK-RRT algorithm by simply adding a counter variable, allowing the algorithm to force the state to become non-busy after a fixed number of busy iterations.

The overall increase in planning performance due to the use of the busy flag depends strongly on the type of domain and especially on the type of Skills involved. There are many domains where non-deterministic decision making only occurs at certain key events, and the Skills behave deterministically otherwise. In such domains, preventing unnecessary branching and state duplication can have an extremely positive impact on performance. As a simple example, consider a Skill that has the purpose of making a robot drive toward a randomly chosen point, similar to the example shown in Section 4.2. In this case, there really only is a single point in time where the Skill performs random sampling, namely at the beginning of its execution, when randomly selecting which point to drive towards. Once

"committed", the calls to this Skill in succeeding states will simply perform deterministic controls that will make the robot drive to the previously non-deterministically selected point. Letting the planner be aware of where the non-determinism occurs via the busy flag allows the planner to focus its entire search solely on *true* decision points, yielding a much faster search. Appendix C shows the internals of several Skills that make further use of the busy flag.

5.3.2 Removing Stale Branches: the RollBack Function

In the previous section we discussed how the BK-RRT algorithm uses the busy flag to perform greedy state expansions. However, we also need to consider what happens if we encounter an invalid state x' (i.e., `Validate` returns *false*) during a greedy branch expansion induced by the busy flag. As defined in the BK-RRT algorithm, the invalid state x' is of course not added to the tree in this case. However, in the case of a greedy state expansion, we should also take a look at the chain of busy states that led to the invalid state x'. Remember, that, due to our previous definition of the busy flag, busy states are guaranteed to only have a single child. Furthermore, busy states will never be selected again as a source node during the standard node selection process. Therefore, the entire chain of busy states that was generated during the greedy branch expansion (that ultimately ended in failure when generating the latest state in the branch x') can be considered *stale*, meaning that none of the nodes in this busy chain will ever be selected again, and therefore none of them will ever contribute to lead towards a goal state as part of a solution sequence.

Because such stale branches consume memory, it makes no sense to keep them around in the tree. Therefore, we introduce the `RollBack` function (see Algorithm 7) that is called when an invalid state is encountered during a busy state expansion (see Algorithm 4). The concept of the RollBack function is very simple: given an invalid state, we follow up its chain of parents, removing nodes until we encounter a non-busy state. Note, that without the RollBack function, the remaining BK-RRT algorithm would still be completely sound. However, as we will show in Section 6.2, the RollBack function can help to significantly reduce overall tree sizes.

5.3.3 RRT as a Subset of BK-RRT

In this section we investigate how our BK-RRT algorithm relates to the traditional RRT algorithm [57]. Similar to BK-RRT, traditional RRT relies on random sampling to construct a search tree through the continuous state space. Figure 5.2 illustrates the growth

of a traditional RRT search tree through a non-dynamic 2D environment. The RRT algorithm grows the tree by repeatedly sampling a random state configuration y from some underlying continuous sampling space Y, then finding the closest node x to the sample y among all nodes currently in the search tree, and growing a fixed distance branch from x toward the sample y, generating a new child node x'. The result of consistently repeating this node selection and expansion process is a random, but very efficient coverage of the continuous search space.

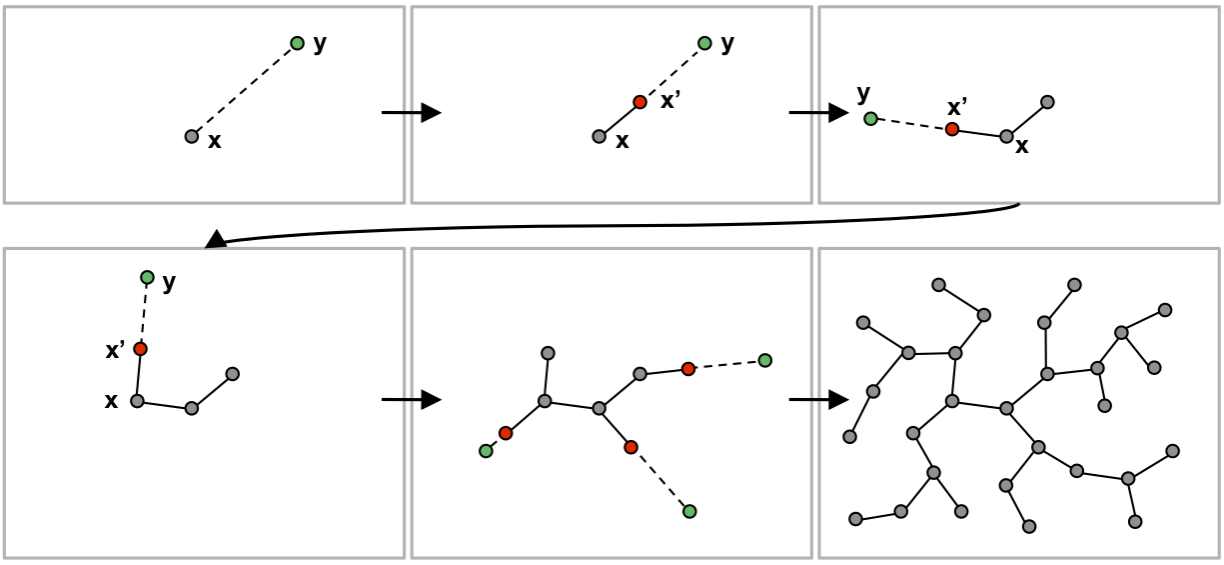


Figure 5.2: A visual demonstration of the standard RRT algorithm.

BK-RRT uses a node *selection* scheme identical to RRT, except that BK-RRT adds the feature of the "busy" flag. Both algorithms sample a configuration y from a space Y and locate the nearest neighbor in the tree, using some distance function.

However, BK-RRT and RRT behave differently when *expanding* from the selected node. Traditional RRT *always* extends its new branch directly toward the sample y. In the non-dynamic example shown in Figure 5.2, this expansion is as simple as constructing and scaling a virtual line from x to y (i.e., $x' = x + dist\ \texttt{Norm}(y - x)$, where `Norm` normalizes a vector to unit length and *dist* is a scalar value indicating the desired expansion step length). Although this expansion step can be more complex in dynamic environments [60], traditional RRT will *always* attempt to grow the branch toward the sample y.

In the case of BK-RRT, however, actions are sampled from our Skills and Tactics model. Because the Skills and Tactics can be configured to generate any action $a \in A$, there is clearly no guarantee that the generated action will actually lead *toward* the random sample y that was constructed and used during the node selection step. Unlike traditional RRT,

however, BK-RRT's Skills and Tactics sampling model allows the selection of actions in a manner that makes *intelligent* decisions based on the current source state x, thus allowing the planner to constrain the search space in a task-oriented manner.

Furthermore, the BK-RRT Skill operator function f does indeed have access to the random sample y that was used during node selection (because it was added to the Skill's Variable set, i.e. $V \leftarrow \{y\}$ in Algorithm 6). A Skill could therefore make use of the sample y and generate actions that depend on its concrete values.

Most interestingly, this availability of y to the Skill's operator function does in fact imply that the traditional RRT algorithm is actually a subset of BK-RRT. If we imagine a Tactic containing a single Skill that only implements the typical RRT "generate an action to directly extend from x towards y" operation, then this BK-RRT search would behave algorithmically identical to standard RRT.

The decision on how much to emulate the standard RRT behavior by generating actions that lead toward y vs. how much to select other actions via Skills and Tactics, depends mostly on the domain. The beauty of BK-RRT is that it allows embedding of arbitrary amounts of Tactics-based domain specialization, with the very general traditional RRT expansion scheme being one extreme on that spectrum.

5.3.4 RRT Parameters and Distance Functions

Both traditional RRT and BK-RRT rely on several crucial parameters to work efficiently. In particular, the `SelectNodeRRT` function (see Algorithm 5) requires that the sampling space Y and the nearest neighbor distance function are defined in a way that allows an effective exploration of the search space, given a particular type of domain.

When employed in non-dynamic domains, RRT can typically be configured to let $X = Y$ and to simply use Euclidean distance for computing the distance between $x \in X$ and $y \in Y$ during the nearest neighbor lookup.

However, because we are planning through a physics-based, second order time-space environment, using a Euclidean nearest neighbor distance approach will fail. This is because the Euclidean distance between two nodes' positions would ignore the fact that the nodes have velocities attached to them and therefore would not provide a good indication of how *long* it would actually take for an actively controlled body to reach a particular sampled configuration y. A much more reliable approach is to define the distance function based on estimated time to get from x to y, rather than pure positional distance.

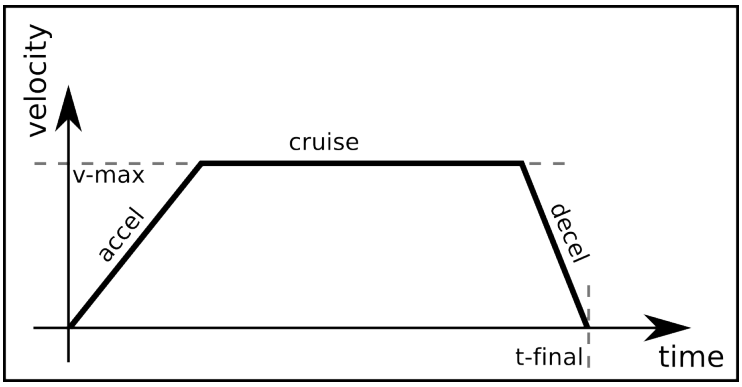


Figure 5.3: A trapezoidal motion model.

For our experimental domains, we use an acceleration-based "trapezoidal" motion model to compute the minimal estimated time for the controlled rigid body in x to reach its target configuration in y. Figure 5.3 shows a visualization of the trapezoidal model for computing the estimated time of a 1-dimensional movement. The model is configured by providing it with values for the body's maximum acceleration, deceleration and velocity (v-max). We can then query this model, providing it the body's initial velocity (which is zero in Figure 5.3), a distance to be traversed, and a desired target velocity to be achieved at the target point. The model will then compute the estimated amount of time required to reach the target. This 1-dimensional trapezoidal motion model can be applied to higher dimensions (e.g., to the 2D translating omni-directional robot model used in robot soccer) by optimally decomposing the higher dimensional vectors into their 1D components, with the goal of minimizing the maximum time over all individual 1D trapezoidal functions.

5.4 BK-BGT Algorithm

RRT's combination of node-selection and extension has the clear advantage of providing a relatively efficient and uniform coverage of the work-space [57]. This uniform expansion however, comes with the trade-off of computational time. Because BK-RRT requires a nearest neighbor distance lookup over all existing nodes in the tree on every iteration, the computational time grows quadratically as the tree size increases. Also note that, unlike in non-dynamic motion planning, this nearest neighbor lookup process cannot always be trivially sped up through the use of pre-sorting the tree in a data-structure (such as KD-Trees), due to the potential non-linearity and assymmetry of the time-based distance function that is required in dynamic environments (see Section 5.3.4). A related concern is that the user might have too many proverbial knobs to tweak to adapt RRT for a particular domain. Wise decisions need to be made about how to define the distance function and

which dimensions of the state space X are to constitute the sampling-space Y. Once defined, the user needs to choose a well-working sampling-distribution.

Considering all these programmatic and computational challenges, it is questionable whether the advantages of RRT, such as its probabilistically uniform and rapid growth through space, are actually a significant requirement for the tactically constrained domains that we encounter. This is particularly true if none of the Tactic's Skills even make use of the random sample y, and thus in no way implement the traditional RRT "extend towards y" operation. For many tactically constrained models, it might in fact make more sense to abandon any attempts to model complicated distance functions that involve knowledge about body motions and the state space, and instead focus on a fast and random growth of the search tree itself.

To test this hypothesis, we introduce a less informed approach that has only the objective to expand the search through our Tactics space in a fast and well-balanced fashion. We call this algorithm Behavioral Kinodynamic Balanced Growth Trees (BK-BGT). Algorithm 8 shows the outer planning loop of BK-BGT which is nearly identical to BK-RRT (Algorithm 4). The significant difference however, is the method of the selection of the node x.

Algorithm 9 shows this new Balanced Growth Trees (BGT) node selection method. Where BK-RRT required a sampling distribution and a nearest neighbor lookup, the only parameter of the BGT node selection is a single constant μ which represents the desired ratio between average leaf depth and average tree branching factor.

At each call, the current ratio of the tree's average leaf depth and average branching factor is compared to μ. If the ratio is greater than μ, then the algorithm will attempt to increase the tree's average branching factor (and thus decrease the overall ratio) by growing its next branch from a randomly chosen non-busy non-leaf node. If the current ratio is less or equal to μ, then the algorithm will attempt to increase the tree's average leaf depth (and thus increase the overall ratio) by extending from a non-busy leaf node.

Effectively this means that a large value of μ leads the algorithm to expand further into the future, but creates a "thinner" tree, whereas a smaller value of μ focuses on a more dense expansion, but with a more limited average time horizon. Figure 5.4 shows the effects of choosing different values of μ to a search tree's depth and breadth.

Algorithm 8: BK-BGT

Input: Domain: d, Initial state: x_{init}, set of goal states: X_{goal}, timestep: Δt, validation function: `Validate`, BGT ratio: μ, max iterations: z.

```
tree ← NewEmptyTree();
tree.AddNode(x_init);
busy ← false;
y ← ∅;
for iter ← 1 to z do
    if busy = true then
        x ← x′;
    else
        x ← SelectNodeBGT(tree, μ); // See Alg. 9
    end
    ⟨x′, L⟩ ← TacticsDrivenPropagate(x, y, Δt, d); // See Alg. 6
    busy ← x′.b_∧;
    if Validate(x′, L) then
        tree.AddNode(x′);
        tree.AddEdge(x, x′, a);
        if x′ ∈ X_goal then
            return TraceBack(x′, tree);
        end
    else
        if busy = true then
            RollBack(x, tree); // See Alg. 7
        end
        busy ← false;
    end
end
return Failed;
```

Algorithm 9: SelectNodeBGT

Input: Tree: tree, BGT ratio: μ.

```
if (AverageLeafDepth(tree) / AverageBranchingFactor(tree)) > μ then
    // Select a random non-busy non-leaf:
    x ← RandomNonLeaf(∀x ∈ tree : x.b_∧ = false);
else
    // Select a random non-busy leaf:
    x ← RandomLeaf(∀x ∈ tree : x.b_∧ = false);
end
return x;
```

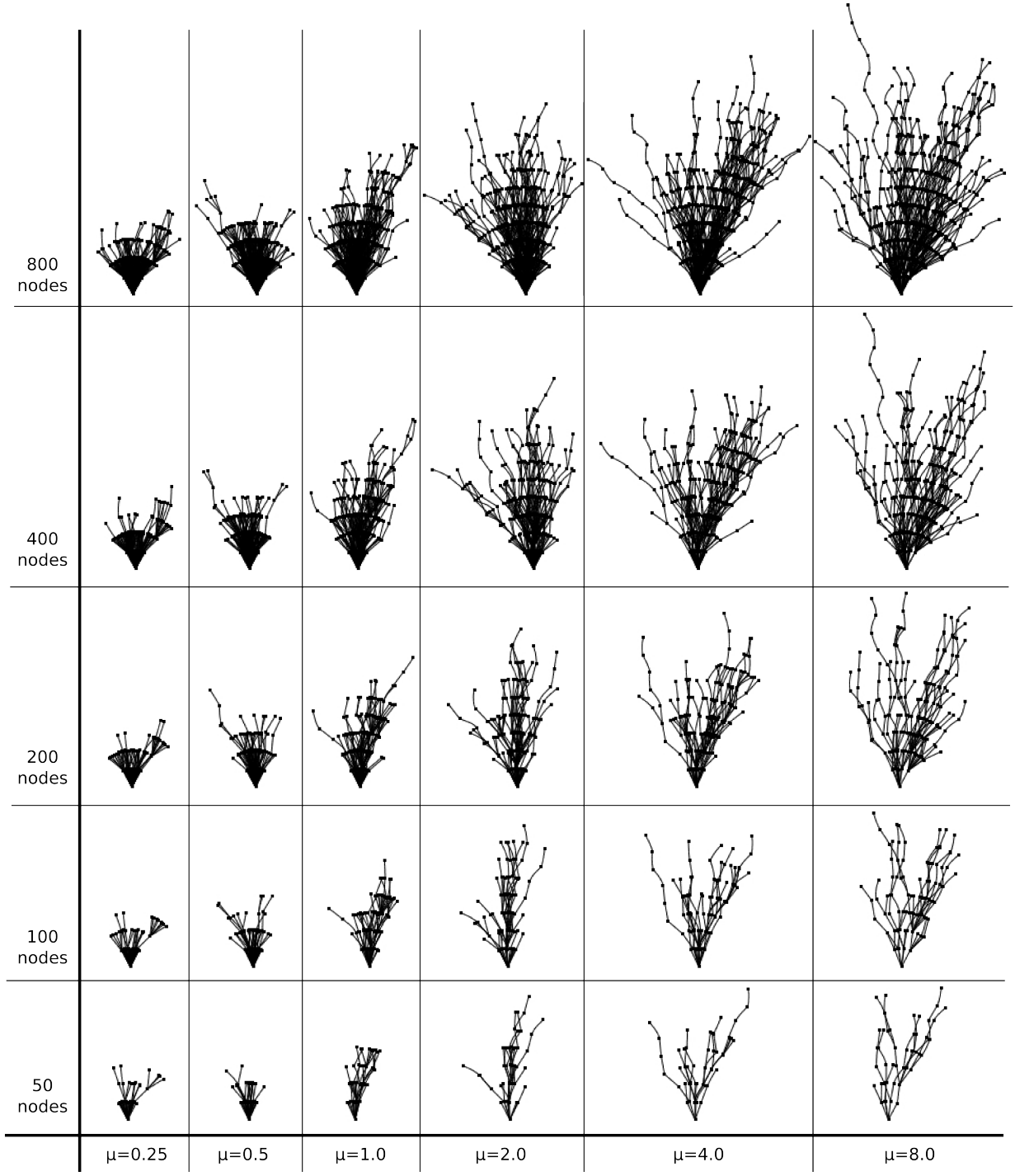


Figure 5.4: Examples of Balanced Growth Trees under different maximum tree sizes (horizontal axis) and values of μ (vertical axis).

5.4.1 Computational Complexity of BK-BGT vs. BK-RRT

One major advantage of BK-BGT, when compared to BK-RRT, is its computational performance. If carefully implemented, BK-BGT's `SelectNodeBGT` function requires in the worst case $O(log\ |\mathsf{tree}|)$ per call, where $|\mathsf{tree}|$ denotes the number of nodes in the tree. This runtime is achieved by keeping running values of the average branching factors and node depths as the tree grows, thus transforming the `AverageLeafDepth` and `AverageBranchingFactor` functions in Algorithm 9 to each only consume constant time per call, independent of the current tree size. Additionally, we can maintain running sets (using e.g. balanced binary trees or a hash table) of what nodes are leaf nodes or non-leaf nodes, resulting in a worst case of $O(log\ |\mathsf{tree}|)$ for the selection of a random non-busy node during `SelectNodeBGT`. Consequently, the BK-BGT algorithm's execution time scales with a worst case time of $O(\mathsf{iter}\ log\ |\mathsf{tree}|)$, where iter is the number of total iterations in the outer planning loop (which can be greater than tree because invalid states are still counted as iterations, but do not contribute to the tree size). In comparison, RRT runs in $O(\mathsf{iter}\ |\mathsf{tree}|)$ time. Furthermore, from a real-world computational time perspective, each of BGT's individual lookup operations are relatively short and simple. BK-RRT's distance function, on the other hand, can take excessive amounts of real-world computational time (depending on the domain), worsening the effects of its approximately quadratic runtime. In Section 6.2, we verify this claim experimentally by comparing observed node selection times of BK-RRT and BK-BGT.

5.4.2 Lack of RRT's Random Sample

Although the BK-BGT outer planning loop (see Algorithm 8) is very similar to BK-RRT (Algorithm 4), there remains one subtle, but important, difference. Unlike BK-RRT's node selection function, BK-BGT's `SelectNodeBGT` does not internally generate nor return a random sample y, because it would serve no purpose during its node selection process. Therefore, the BK-BGT algorithm defines $y = \emptyset$ which is then passed normally into the `TacticsDrivenPropagate` function. In other words, when using BK-BGT, Skills and Tactics are not passed an externally generated sample that they could be used to "extend towards" as is the case in BK-RRT. Consequently, Skills and Tactics are expected to generate useful actions, even without the presence of an externally generated sample y.

5.5 Algorithm Properties and Variations

Now that we have defined BK-RRT and BK-BGT, we discuss their planning behavior in terms of *optimality* and *completeness*. Furthermore, we present several possible variations to the BK-RRT and BK-BGT algorithms and discuss their trade-offs.

5.5.1 Completeness

Completeness is the property that a planner is guaranteed to find a solution if it exists or that it will return failure otherwise, in either case within some *finite* amount of time [58]. Our presented BK-RRT and BK-BGT planning approaches work by randomly sampling specific actions from an underlying continuous action set. Therefore, it is infeasible for the planner to explore *every* possible action sequence in a finite amount of time. Additionally, the presented planners have no direct means of verifying that *no solution* exists for a particular domain. Consequently, as is typically the case with sampling-based planners, neither BK-RRT nor BK-BGT are strictly complete.

A related property, that is relevant for sampling based planners, is *probabilistic completeness*. A planner is *probabilistically complete* if the probability that the planner finds a solution (if a solution exists) goes to 1 as planning time goes to infinity [58]. Traditional non-dynamic RRT has been shown to be probabilistically complete [47]. In our dynamic case, this probabilistic completeness property hinges on whether RRT's nearest neighbor distance function and the extend operator are able to maintain their traditional soundness properties of continually "making progress", namely that a succeeding node x' grown from a node x toward sample y will automatically become the new closest node to the sample y among all nodes in the tree. Unlike in traditional RRT, the physics-based dynamic planning environment of BK-RRT typically requires custom definitions of the sampling space Y, the distance function, and the "extend toward y" operator, all depending strongly on the type of domain and its bodies. Therefore, we cannot make any general claims about BK-RRT's probabilistic completeness.

Unlike BK-RRT however, BK-BGT does not rely on any sampling-based nearest neighbor lookups, distance functions, or extend operators. In fact, assuming an unconstrained Tactics model (i.e., a single Skill that uniformly selects a random action from A), BK-BGT would, in the infinite time limit, converge to cover the entire search space, thus making it probabilistically complete. The reasoning behind this is quite intuitive: over the entire runtime of BK-BGT, the BGT node selection will continually alternate between expanding from a random leaf or a random non-leaf in the tree, thus ensuring that during the entire course of planning, the search tree can be extended from *any* existing node with a

non-zero probability. Each such expansion will select *any* random action from A with a non-zero probability, thus overall ensuring discoverability of all possible action sequences with non-zero probability, leading to an exhaustive, but BGT-balanced, brute force search over the entire continuous action space A.

Of course, such an unconstrained brute force search is computationally infeasible in most finite-time applications. In real-world use, BK-BGT's set of selectable actions at each iteration is constrained by its particular Skills and Tactics model. Solutions that require action sequences not generatable by the provided Skills and Tactics can obviously never be found. Therefore, we can only claim probabilistic completeness for BK-BGT *within* the constraints of its Skills and Tactics model.

5.5.2 Optimality and Multi-Solution Planning

Optimality is the property that the returned solution of a planning algorithm is also guaranteed to be the *best* solution. In non-dynamic planning, the definition for *best* typically means that the *shortest path* leading to the goal state is returned. However, in our dynamic environments, it makes more sense to reason about the shortest trajectory in terms of *time*, rather than in terms of distance. Depending on the domain, other metrics for optimality might be desirable, such as the solution with the *least curvature*, or the solution with the *least multi-body interactions*.

No matter what definition we choose for optimality, neither BK-RRT nor BK-BGT can generally guarantee that their generated solutions are optimal. This is due to the fact that either algorithm will return the first solution it encounters, and that for any solution encountered, there is no definite guarantee that there does not exist a better one. This non-optimality is a common property for sampling-based planners.

A common solution to this non-optimality property of sampling-based planning, is to let the algorithm keep planning even after a valid goal state has been found, thus potentially generating additional valid, but different, solutions. Any of the previously described optimality metrics could then be used to automatically compare the multiple solutions and return the best one. How long to plan for and how many solutions to compare are of course user-defined settings and depend on the circumstances of the application. Although such an automated solution selection process is likely desirable in most applications, including robotics, there are some domains where a manual selection might be preferable. For example, in computer animation applications, it might be desirable to let a computer animator manually inspect the multiple solutions found, and select the visually most appealing one (a metric that is hard to quantify computationally). In the graphics community, this process has also been referred to as "many-worlds browsing" [85].

5.5.3 Hybrid Approaches

The main difference between BK-RRT and BK-BGT is the method of node selection. It is therefore important to point out the possibility of "hybrid" planning approaches which combine RRT and BGT search methods rather than choosing one of the two extremes. One simple way of implementing such a hybrid planner is to non-deterministically select one of the two node selection methods during each planning iteration. Note, that by choosing such a hybrid implementation, one would also have to deal with the configuration of both methods. In other words, one of the main appeals of BGT, namely not having to define sampling spaces, sampling distributions, and distance functions for a particular domain, disappears. Nevertheless, it might make sense for some domains to pursue such a hybrid approach, as it could increase the tree expansion rate of the overall algorithm due to the faster BGT approach, while still providing some geometrically structured coverage of the work-space, due to RRT. We analyze the performance of such a hybrid approach as part of our results in Section 6.2.

5.6 Chapter Summary

In this chapter, we have introduced two concrete Skills and Tactics-driven planning algorithms, namely Behavioral Kinodynamic Rapidly-Exploring Random Trees (BK-RRT) and Behavioral Kinodynamic Balanced Growth Trees (BK-BGT). We have introduced the busy flag and the RollBack function as key features of our algorithms and we have shown how to combine BK-RRT and BK-BGT into a hybrid planning approach.

In the following chapter, we evaluate and compare these algorithms experimentally in several simulated domains.

Chapter 6

Empirical Evaluation in Simulated Domains

In Chapter 5 we have introduced two concrete Skills and Tactics-driven randomized physics-based planning algorithms, namely Behavioral Kinodynamic Rapidly-Exploring Random Trees (BK-RRT) and Behavioral Kinodynamic Balanced Growth Trees (BK-BGT). In this chapter we evaluate these algorithms empirically in several challenging simulated domains and provide a detailed analysis of their performance characteristics.

6.1 Implementation and Visual Results

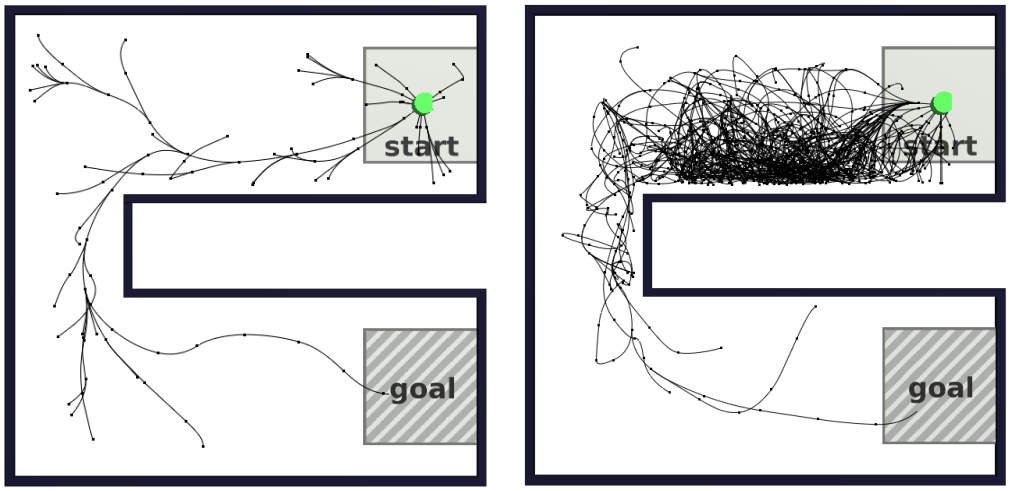
We tested BK-RRT, BK-BGT, and hybrid approaches in multiple simulated domains. We implemented the algorithms in C++, using NVIDIA PhysX Version 2.8.1 [24] as the underlying physics engine. All random sampling functions were implemented to use the Mersenne-Twister pseudo-random number generator [66]. All the results were computed using a single desktop computer with an Intel Core processor.

In this section, we first present the individual domain configurations, including their Skills and Tactics models, interleaved with visual example results for each domain. We then show and discuss our planners' quantitative performance for these domains in Section 6.2.

6.1.1 Navigation Domain

The *navigation* domain is a traditional trajectory planning problem and requires a robot rigid body to reach a target area by navigating through a U-shaped environment (see Figure 6.1).

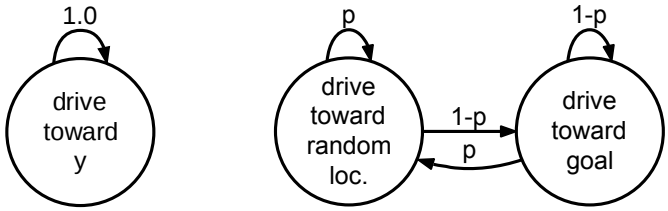
For this domain, the Tactics model used with our BK-RRT algorithm consists of only a single Skill (Figure 6.2 (a)), effectively implementing the traditional, standard RRT algorithm [57]. The internal RRT sampling distribution over the sampling space Y from which y is sampled, is configured to provide either a random location sampled from the domain (with probability p), or a random location from the goal area (with probability 1-p) on each iteration. The Skill will then apply a control action, accelerating the robot toward the sample y, using an internal implementation similar to the one presented in Algorithm 2 of Section 4.2. This effectively leads RRT to expand its branches throughout the domain, while also being biased to grow towards the goal location. Figure 6.1 (a) shows this typical RRT growth (with $p = 0.7$).



(a) BK-RRT Version. (b) BK-BGT Version.

Figure 6.1: Example search trees from the navigation domain. The initial state is in the center of the "start" area on the top right, the goal is to reach the shaded area on the bottom right.

Recall, that for BK-BGT, there is no global sampling distribution from which a sample y is generated. Instead, we implement a functionally identical sampling behavior explicitly through a Tactic with two Skills (Figure 6.2 (b)) that accelerate the robot towards a random location from the domain, or towards the goal location respectively. However, unlike RRT,



(a) BK-RRT Version. (b) BK-BGT Version.

Figure 6.2: Tactics used in the navigation domain.

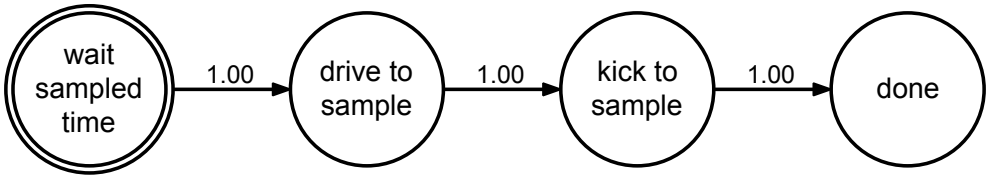


Figure 6.3: The Tactic used in the simulated robot minigolf domain.

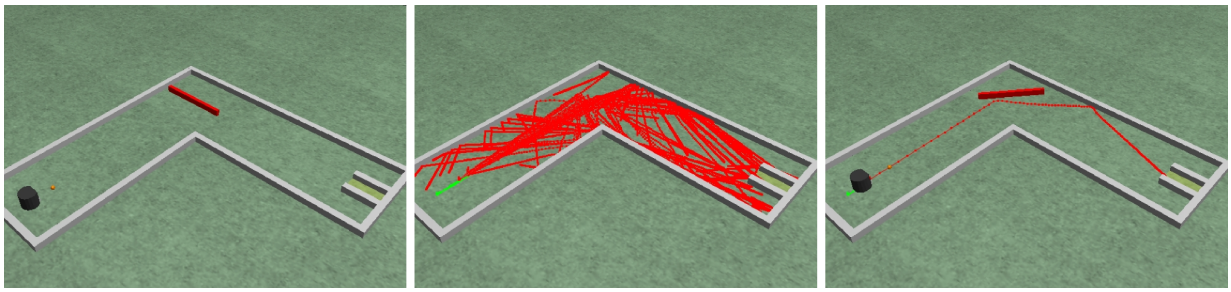


Figure 6.4: The simulated robot minigolf domain. The left image shows the initial state. The controlled robot is the dark cylinder on the left. The bar-shaped obstacle at the center of the course is rotating at constant velocity and thus represents a predictable, foreign-controlled body. The goal is to deliver the ball into the U-shaped area at the bottom right. The center image shows the planner's search tree with all the red nodes representing ball positions. The right image shows a legal solution found by the planner: the robot body waits an appropriate amount of time and then accurately manipulates the ball to use the rotating obstacle as a bounce-platform, leading it into the goal position.

BGT does not have the guarantee that the node from which it expands the tree is actually anywhere close to the sample which it is accelerating the robot towards. Given that there are no other Tactical constraints for this domain, this leads to a less efficient growth of BGT, as can be seen in Figure 6.1 (b).

6.1.2 Simulated Robot Minigolf Domain

In the simulated *robot minigolf* domain (also see Section 3.1.3), a robot rigid body has the objective of putting a ball through a course with a moving obstacle. Figure 6.3 shows the Tactics model, allowing the robot body to wait for a sampling-based amount of time, position itself at a sampling-based location, and then drive into and putt the ball towards a sampling-based target with a sampling-based velocity. Note that the "wait sampled time" Skill, plays a significant role because the obstacles in the domain are moving, thus making the robot's timing critically important. The internals of that particular Skill are given in Appendix Section C.1. Figure 6.4 shows one particular solution from this domain.

6.1.3 Simulated Robot Soccer Domain

The simulated *robot soccer* domain (also see Section 3.1.2) significantly enhances the concept of goal-driven manipulation of a passive body, and aims to demonstrate the unique tactical planning abilities of our planner. This domain models a single attacker vs. multiple opponents soccer situation, containing three dynamic adversary rigid bodies that are running deterministic adversarial Tactics to block the ball from the goal. Note that, for this simulated version of robot soccer, we assume perfect predictability for the opponents' actions. We will discuss how to plan without this assumption when we present a real-world implementation of the robot soccer domain in Chapter 7. Nevertheless, from a planning perspective, the simulated soccer domain represents a very difficult problem. Not only does the actively controlled robot need to navigate around moving bodies, but it also needs to exert accurate control on the ball to achieve the scoring of a goal in an adversarial environment.

To achieve this task, the controlled body features a rather complex tactical model as shown in Figure 6.5. This Tactic allows a robot-body to drive toward the ball, drive with the ball toward a sampling-based location ("dribble"), flat kick or upwards kick the ball towards a sampling-based target near the robot ("minikick" and "minichip"), or towards a sampling-based point in the goal ("kick"). Appendix Section C.2 shows the details of the "Sampled Kick" Skill that is used with slightly different parameters for both the "sampled kick" and "minikick" of this Tactics model. This domain generated various interesting solutions, one

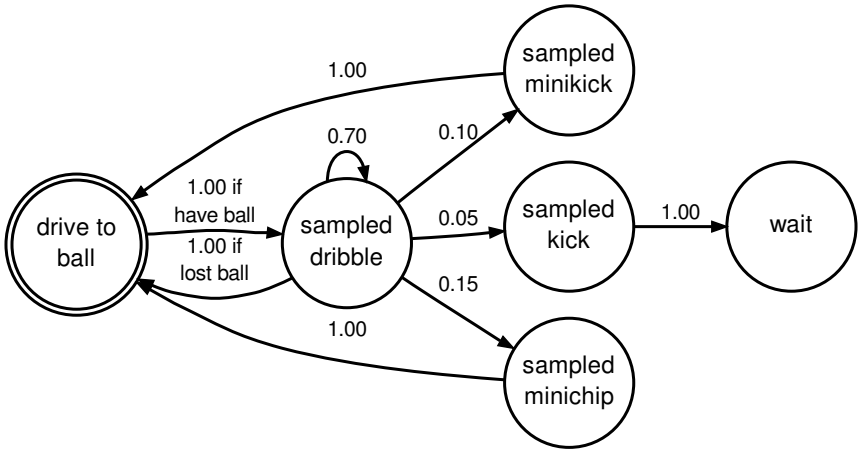


Figure 6.5: The Tactic used in the simulated robot soccer domain.

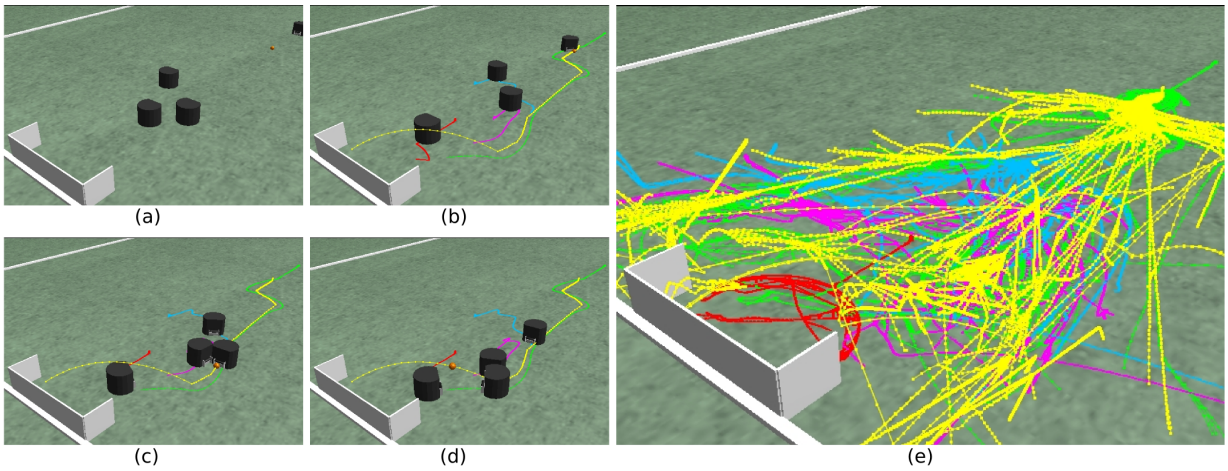


Figure 6.6: An example of the simulated robot soccer domain. (a) The initial configuration of the bodies. The controllable robot-body is initially at the top right, and the planning goal is to deliver the ball to the goal while avoiding the defenders and goalie robot bodies; (b)-(d) Snapshots of one solution found by the planner; (e) The entire search tree representing the positions of all rigid bodies, including ball (yellow nodes), controlled body (green nodes), and defenders (other colors).

of which is shown in Figure 6.6.

6.1.4 Pool Table Domain

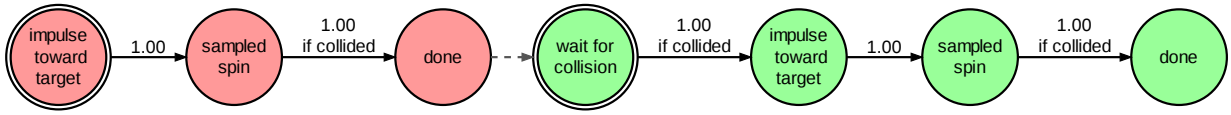


Figure 6.7: The chained Skills and Tactics used for the pool table domain. The cue ball uses the Tactic consisting of the left three Skills (red), whereas the intermediate ball uses the remaining four Skills on the right (green).

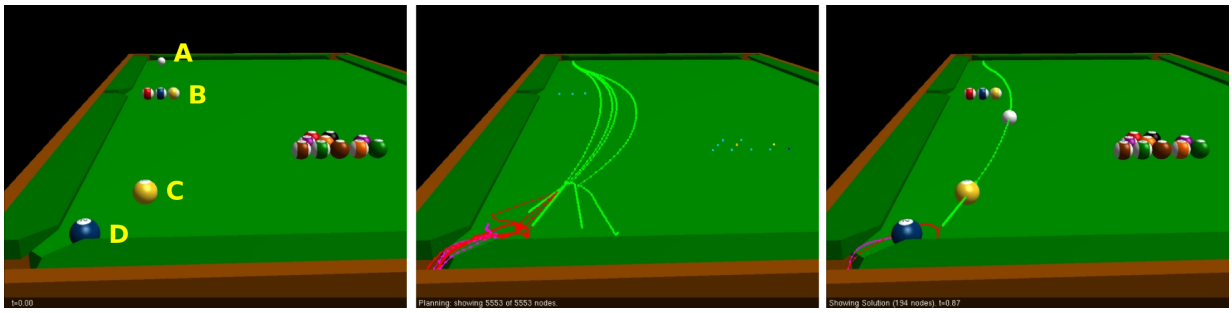


Figure 6.8: A solution in the pool table domain. Given the initial state in the left image, the planner finds several acceptable solutions (center image) and automatically selects one for execution (right image).

The *pool table* domain (also see Section 3.2.2) demonstrates the ability to chain multiple Tactics, as shown in Figure 6.7. The goal of the domain (see Figure 6.8) is to let the cue ball (A) hit the yellow ball (C) which will then roll into the blue ball (D), delivering it into its pocket, all without letting the cue ball collide with any of the three obstacle balls (B). Note, that in this domain, the pool balls are treated as active bodies, but with limited actuation abilities. While the initial force is applied directly to the cue ball, the following balls obtain their force passed on through the collision. The balls' Tactics, and in particular the "sampled spin" Skill is able to slightly perturb their natural path by applying a sampling-based spin towards the target. The ability of the planner to sample from different degrees of these perturbations (thanks to the sampling-based Skill) allows the automatic selection of a solution which will not collide with any striped balls, and yet requires a minimum of perturbation from their natural passive behavior.

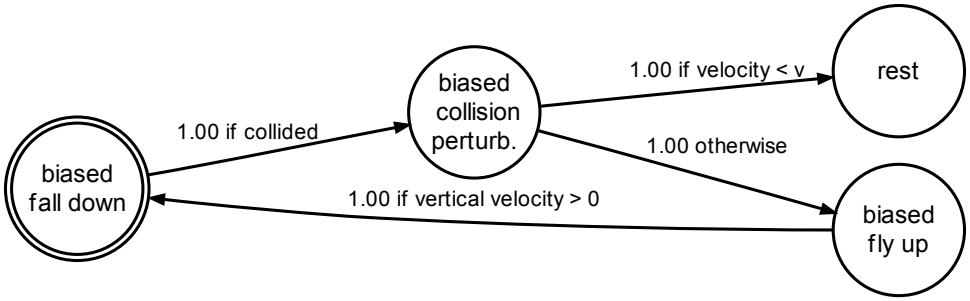


Figure 6.9: The Tactic used for the many-dice domain. Each die carries its own instance of this Tactic.

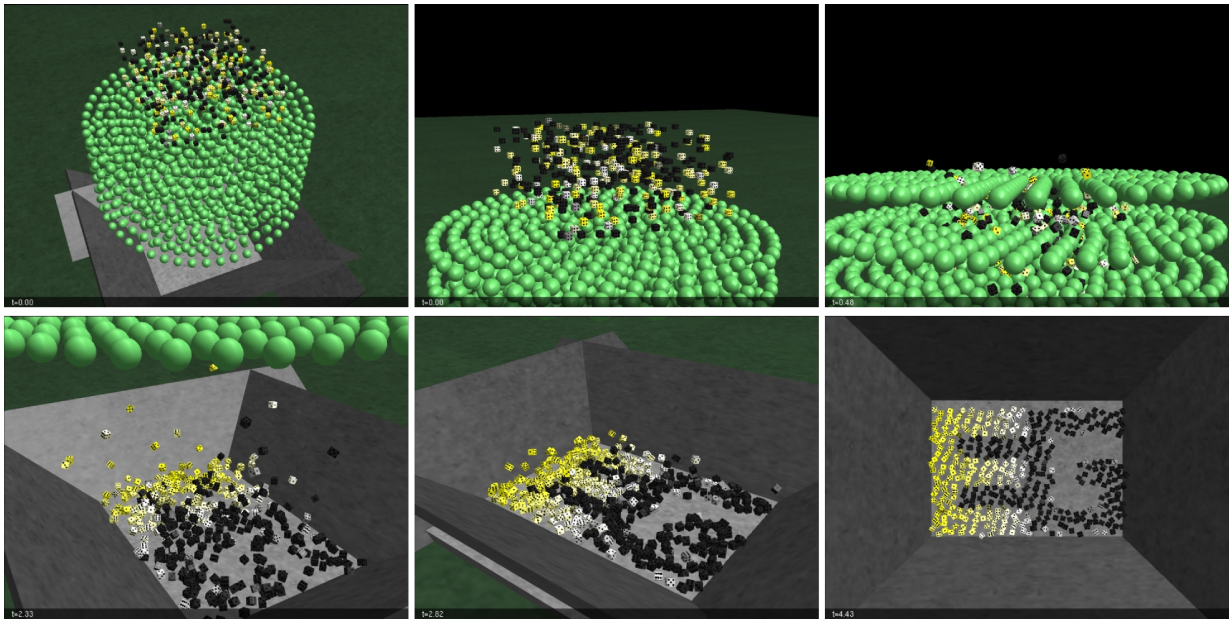


Figure 6.10: A solution sequence of the many-dice domain. 400 randomly initialized dice fall towards a grid of spherical obstacles (top row, left to right). After bouncing through the grid, they fall into a bucket where they come to a rest (bottom row, left to right) and form the Eurographics (EG) logo.

6.1.5 Many-Dice Domain

The *many-dice* domain (also see Section 3.2.1) represents a traditional computer animation task and aims to demonstrate the ability of our planner to run several hundred Tactical instances concurrently. The objective in this domain is to control the fall of 400 dice, each initialized to a random position. While their fall and collisions are supposed to look physically plausible, they also have the target objective of forming the Eurographics logo. This is achieved by providing each die with a separate instance of the Tactic shown in Figure 6.9. This domain is challenging because although we are able to actuate the individual dice, we are interested in achieving the illusion of a natural looking free-fall. At the same time, the dice still have to passively react to various collisions with spherical objects and with other dice. In order to achieve this goal, the Tactical model carries multiple Skills, each representing a different state of the fall based on a die's current position and velocity. These Skills are again sampling-based, but probabilistically biased to create motions towards the die's final target position (which is determined at initialization). Appendix Section C.3 shows the internal details of the "biased fall down" Skill. Figure 6.10 shows a visual result sequence of this domain.

6.2 Performance Analysis

We tested the BK-RRT and BK-BGT algorithm on all of the simulated domains introduced above. Table 6.1 shows average real world clock planning times, tree sizes, planning loop iterations (i.e., final value of iterin the BK-RRT and BK-BGT algorithms), and success rates. We used a planning timestep Δt of 1/60th of a second. BK-BGT μ values were selected experimentally, to be explained in more detail in Section 6.2.1. For the navigation, robot minigolf, robot soccer, and pool table domains, the planner's search was limited to 25000 nodes or 50000 iterations (whichever occurred first). A success in these domains was counted if the planner reached a goal state within these search limits, a failure otherwise. For the many-dice domain, we did not define a boolean goal state. Instead, the planner was setup to search until a maximum number of 15000 nodes was reached and then return the *best* solution, using a *goal evaluator* metric. This metric was defined as the sum of square distances of all dice to their desired goal positions.

In the traditional navigation domain, BK-RRT clearly dominates, requiring less planning time, nodes, and iterations than BK-BGT, and furthermore delivering valid solutions for all of the 100 trials (vs. BK-BGT's 53%). These results clearly emphasize the advantage of a RRT-based search scheme in tactically unconstrained motion planning environments and go hand in hand with the observed visual growth of the search trees in Figure 6.1.

Navigation Domain						
Algorithm	N	μ	Time (s)	Nodes	Iterations	Success
BK-RRT	100	n/a	0.18 ± 0.08	1086 ± 369	1733 ± 686	100%
BK-BGT	100	1000	3.14 ± 1.35	13713 ± 5774	36873 ± 15908	53%

Robot Minigolf Domain						
Algorithm	N	μ	Time (s)	Nodes	Iterations	Success
BK-RRT	100	n/a	4.28 ± 5.20	16898 ± 10606	16898 ± 10606	41%
BK-BGT	100	10	0.33 ± 0.38	3484 ± 4159	3485 ± 4158	99%

Robot Soccer Domain						
Algorithm	N	μ	Time (s)	Nodes	Iterations	Success
BK-RRT	100	n/a	5.34 ± 2.02	21757 ± 7434	22484 ± 8152	20%
BK-BGT	100	100	0.55 ± 0.47	1723 ± 1306	2338 ± 1966	100%

Pool Table Domain						
Algorithm	N	μ	Time (s)	Nodes	Iterations	Success
BK-RRT	100	n/a	1.76 ± 0.64	3887 ± 1670	15033 ± 5510	100%
BK-BGT	100	1000	0.21 ± 0.19	1284 ± 1510	1653 ± 1596	100%

Many-Dice Domain						
Algorithm	N	μ	Time (s)	Nodes	Iterations	Success
BK-RRT	6	n/a	1278.61 ± 7.03	15000 ± 0†	15000 ± 0†	100%‡
BK-BGT	6	1000	304.58 ± 10.08	15000 ± 0†	15000 ± 0†	100%‡

† Many-dice used a fixed preset tree size.

‡ Many-dice did not have a boolean goal state; instead the solution with the smallest overall distance to the target-configuration was selected from the full tree.

Table 6.1: Performance comparison of BK-RRT and BK-BGT across multiple domains. N is the number of trials. The ± symbol indicates Standard Deviation. For all domains except many-dice, each planning trial was limited to 25000 nodes or 50000 iterations (whichever occurred first).

In the robot minigolf, robot soccer, and pool table domains, however, BK-BGT consistently outperforms BK-RRT, requiring less planning time, nodes, and iterations. In fact, in the robot minigolf and robot soccer domains, BK-RRT frequently fails to deliver any valid solution within the allotted planning tree size. These results support our hypothesis that there is little to none benefit of RRT's node selection scheme in domains that are tactically constrained and that do not make use of RRT's random sample y. Within these domains, BK-BGT's fast and well-balanced node selection approach allows an extremely efficient search through the action space that is defined by the Skills and Tactics. Finally, for the many-dice domain, BK-RRT requires an excessive amount of time, due to the heavy computations by the RRT nearest neighbor distance function for a 400 body dynamic environment.

An additional disadvantage of BK-RRT, when compared to BK-BGT, is its computational complexity that grows rapidly with larger tree sizes. Figure 6.11 shows a timing analysis of the BK-RRT algorithm, breaking down the total time spent during search into RRT node selection, Skills and Tactics execution, and physics engine simulations. As expected, RRT's node selection quickly becomes the bottleneck for larger search trees, due to its quadratic scalability. Figure 6.12 shows the direct comparison of the time consumed by BK-RRT's `SelectNodeRRT` vs. BK-BGT's `SelectNodeBGT` functions over growing tree sizes. These results validate our computational complexity discussions from Section 5.4.1.

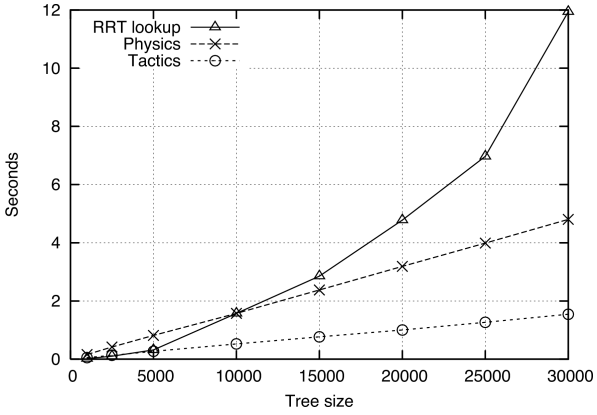


Figure 6.11: BK-RRT analysis of accumulated time spent in node selection, physics, and tactics computations respectively over increasing tree sizes.

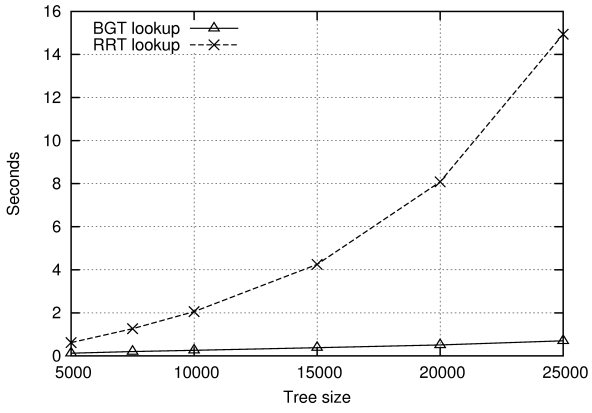


Figure 6.12: Comparison of the computational runtime of the BK-RRT and BK-BGT node selection algorithms over increasing search tree sizes.

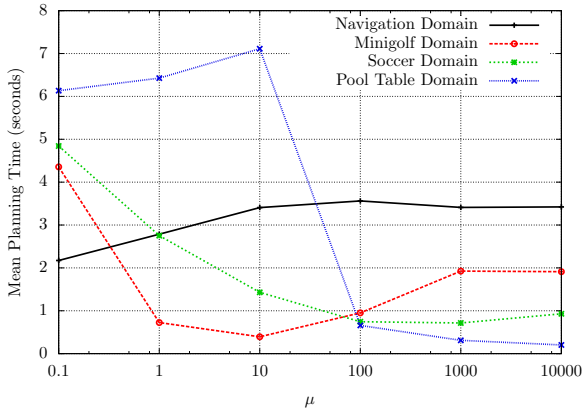
6.2.1 Analysis of BK-BGT's μ Parameter

As we have discussed in Section 5.4, BK-BGT's tree growth is controlled by the value of the μ parameter. In this section, we investigate how the selection of the μ parameter affects planning performance in our simulated domains. Figures 6.13(a) and 6.13(b) show the impact of different μ values on mean planning time and success rate in the navigation, robot minigolf, robot soccer, and pool table domains. Across all the domains, choosing an inappropriately small μ value leads to significantly reduced success rates. This is to be expected, as BK-BGT with a small μ value will focus its search on breadth, not achieving sufficient depth to encounter a goal state in the allotted number of planning iterations. Interestingly, especially in the robot minigolf domain, choosing a large value of μ can also led to diminished performance, because the planner focuses its search on a deep exploration of the state space, not covering as many near term alternatives. As mentioned previously, the many-dice domain does not use a boolean goal state, but instead tries to find the *best* state within a search of limited tree size. Figure 6.13(c) shows the effects of different μ values on the quality of solutions. Based on these results, we have chosen the μ values for our other performance experiments.

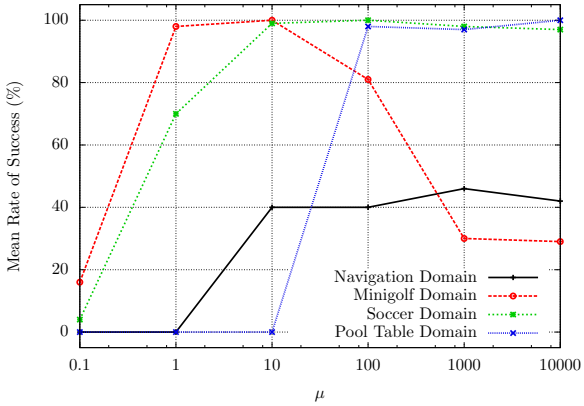
6.2.2 Analysis of Hybrid Approaches

In Section 5.5.3, we have introduced the concept of a hybrid planning approach that randomly selects between an RRT-based and BGT-based node selection at each planning iteration. We implemented support for such a hybrid selection scheme in our planner, which is able to select the BGT node selection behavior with a user-definable probability p and the BK-RRT behavior with the corresponding probability $1 - p$. In Figures 6.13(d), 6.13(e), and 6.13(f) we present the impact of varying this probability on planning time, tree size, and success rate. Note that, for the values of $p = 0$ and $p = 1$, the results correspond with the pure BK-RRT and BK-BGT results shown in Table 6.1.

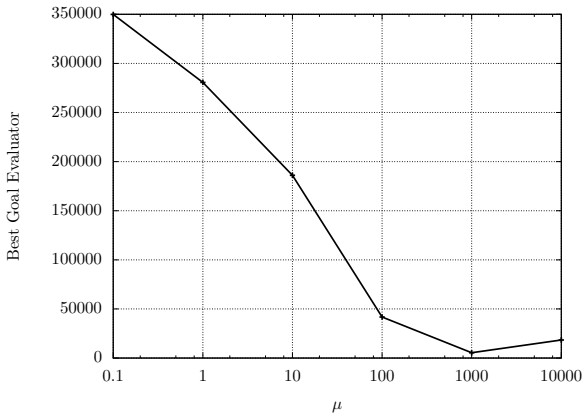
For most of the domains, the transition between the two pure planning methods behaves monotonically. The graphs suggest that for each of the domains, the best planning method is in fact either pure BK-RRT (navigation domain) or pure BK-BGT (other domains). However, one consistent advantage of the hybrid selection scheme is that it delivers better results than the worst case extreme of wrongly choosing either of the pure BK-RRT or BK-BGT planning methods for a particular domain. Therefore, in particular if the domain might not be well known or no previous evaluation of BK-RRT vs. BK-BGT for the domain exists, a hybrid approach should be considered a safe default, delivering acceptable performance, without being fully exposed to certain unique weaknesses of either BK-RRT or BK-BGT.



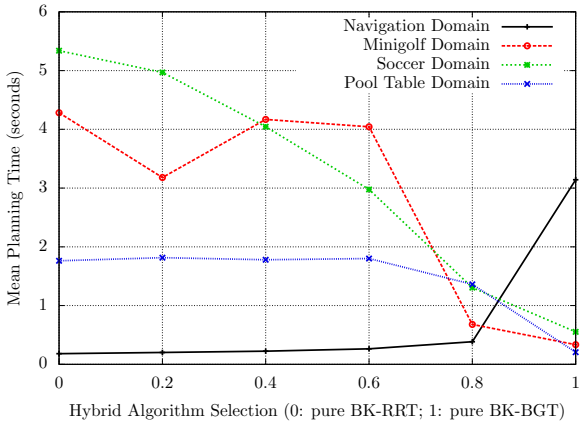
(a) Impact of μ on BK-BGT planning time.



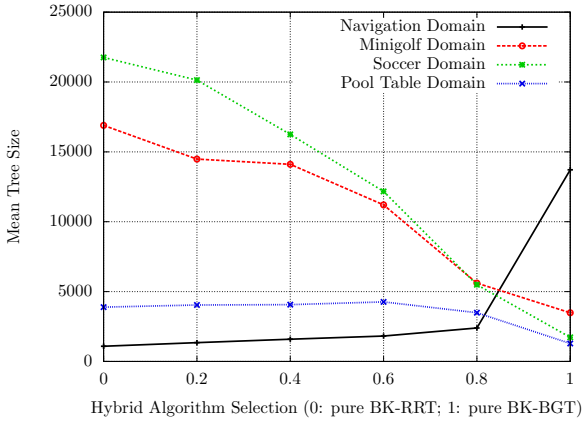
(b) Impact of μ on BK-BGT planner success rate.



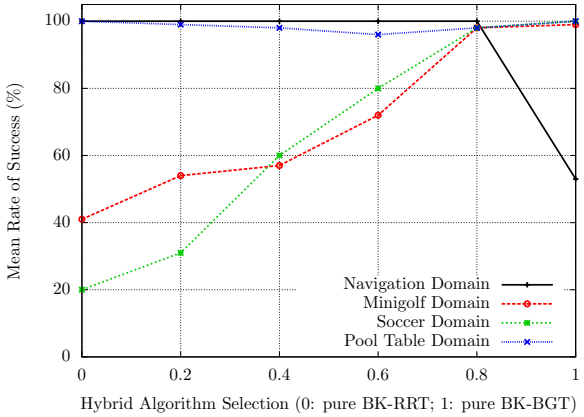
(c) Impact of μ on result quality in the many-dice domain (lower goal evaluator values are better).



(d) Impact of algorithm selection on planning time.



(e) Impact of algorithm selection on tree size.



(f) Impact of algorithm selection on success rate.

Figure 6.13: Performance analysis of BK-RRT and BK-BGT.

6.2.3 Analysis of the RollBack Function

We have introduced the RollBack function (see Section 5.3.2) as a method for removing *stale* nodes that are generated when the busy flag triggers a greedy branch expansion that leads into an invalid state. Table 6.2 compares the performance of BK-BGT with and without the RollBack function enabled, across the navigation, robot minigolf, robot soccer, and pool table domains. In all of the domains, the RollBack function is able to significantly reduce the resulting tree sizes. Only in the navigation domain does the RollBack function increase planning time, because by removing stale nodes, it allows the planner to continue its search longer before hitting the 25000 node limit. Consequently, thanks to the RollBack function, the navigation domain's success rate is also increased by 39 percentage points. In all other domains, planning times are in fact decreased, most likely because reduced tree sizes also lead to faster node selection during each iteration.

	Time (s)			Nodes			Success		
Domain	NRB	RB	%Δ	NRB	RB	%Δ	NRB	RB	Δ
Navigation	2.04	3.14	53.9%	23556	13713	−41.8%	14%	53%	39pp
Minigolf	0.33	0.33	0.0%	3847	3484	−9.4%	100%	99%	-1pp
Soccer	0.63	0.55	−12.7%	2683	1723	−35.8%	100%	100%	0pp
Pool Table	0.22	0.21	−4.5%	1654	1284	−22.4%	100%	100%	0pp

Table 6.2: BK-BGT performance comparison without the RollBack function (NRB) and with the RollBack function (RB), using the same values for μ and N as in Table 6.1.

6.2.4 Tactically Unconstrained Multi-Skill Planning

The Tactics used for the experimental domains (e.g., Figures 6.3 and 6.5) rely on manually defined probabilities and events to constrain the possible transition sequences between individual Skills. An interesting question is whether or not such manual transition constraints are actually necessary for efficient planning, or whether it is possible to completely rely on the planner to automatically choose between any possible Skill transitions to automatically find valid solutions. Figures 6.14 and 6.15 show versions of the Soccer and Minigolf Tactics which do not impose any bias or constraints on the possible transition sequences between Skills, but instead allows the planner to transition between any two Skills in the Tactic with equal probability.

The potential advantage of such an unconstrained Tactics model is that no human domain knowledge is required to define the transition probabilities between Skills. Instead, the Tactic is defined solely by its set of usable Skills, allowing the planning algorithm to freely explore any possible sequence throughout this set of Skills. Of course, by not imposing

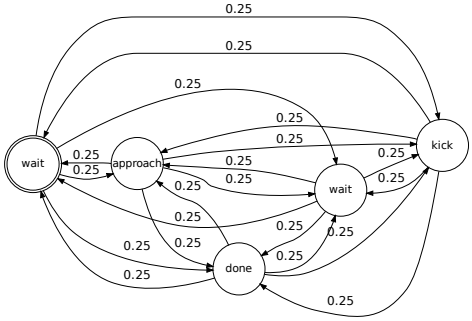


Figure 6.14: A Robot Minigolf Tactic without Skill-transition constraints.

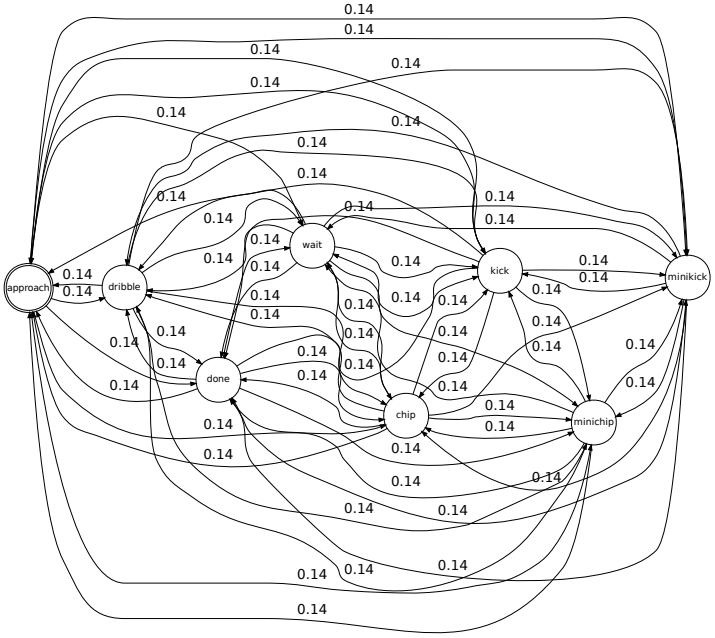


Figure 6.15: A Robot Soccer Tactic without Skill-transition constraints.

any constraints on the possible transitions between Skills, the planner's search becomes less informed, potentially having to cover a larger search space, which can result in longer planning times. On the other hand, an unconstrained Tactic also opens up the possibility of the planner finding novel solutions based on Skill transition sequences that cannot be found with a manually constrained Tactic that prevents certain transition sequences.

Tactically Unconstrained Robot Minigolf Domain						
Algorithm	N	μ	Time (s)	Nodes	Iterations	Success
BK-RRT	100	n/a	2.74 ± 0.98	22192 ± 7503	22488 ± 7597	13%
BK-BGT	100	10	2.48 ± 1.03	20854 ± 8892	20944 ± 8905	18%

Tactically Unconstrained Robot Soccer Domain						
Algorithm	N	μ	Time (s)	Nodes	Iterations	Success
BK-RRT	100	n/a	5.95 ± 2.20	21416 ± 7461	22362 ± 7782	25%
BK-BGT	100	100	1.56 ± 1.33	4340 ± 3630	6113 ± 5265	100%

Table 6.3: Performance of tactically unconstrained planning.

We test the ability of our planner to find solutions with unconstrained Tactics in the simulated robot minigolf and soccer domains. Our experimental setup is identical to the one of Section 6.2, except that the planner now uses the unconstrained Tactics (Figures 6.14

and 6.15). Table 6.3 shows the planning performance results. Comparing these results with the previous ones from the manually constrained Tactics (see Table 6.1), we make several key observations.

Using an unconstrained Tactics model in the minigolf domain, the overall success rates (given a limited planning tree size of 25000 nodes) are low, indicating that the planner struggles to find transition sequences between Skills that lead to a valid solution. Interestingly, a planning time comparison between the BK-RRT experiments of the unconstrained and traditional minigolf domain suggests that the planner is actually exploring the space more quickly (i.e., taking less time per node) when using the unconstrained Tactic. A likely reason for this reduction in planning time is the fact that the unconstrained Tactic allows the planner to more frequently select Skills that do not actually manipulate the ball (such as the "wait" Skill), leading to less computational overhead by the Skill, and by the physics engine. Visual results in the unconstrained minigolf domain support this evidence, as the planner frequently selects sequences of Skill executions that perform nonsensical, but computationally cheap sequences of actions (e.g., a robot positioning itself, waiting, re-positioning itself, waiting more, without actually manipulating the ball).

In the robot soccer domain, overall success rates are relatively similar between unconstrained and constrained Tactics, with a slight advantage to the unconstrained Tactic, suggesting that the planner is able to find valid sequences of Skills, even without manual Skill transition constraints. However, the unconstrained search consumes slightly more time on average than a manually constrained Tactic, suggesting that the manual definition of transition probabilities has at least a small impact in increasing planning performance for this domain.

The Lack of Behavioral Modeling Abilities of Unconstrained Tactics

Success rates and planning times are only a partial indicator for the planner's overall performance. Another metric to consider is the *quality* of the solutions generated by the planner. Note, that the manually defined transition probabilities of a Tactic not only serve the purpose of reducing planning times, but also simultaneously act as a tool to constrain the resulting behavior sequences of the bodies, by preventing the search over transition sequences that are undesired. For example, in the simulated robot soccer domain, the Tactics model (see Figure 6.5) ensures that the robot will only enter a "dribble" Skill after obtaining the ball, and that the robot will only "wait" after having kicked the ball. Without such manual guidance, the planner can find action sequences that, despite reaching a valid goal state, are highly non-optimal, and behaviorally odd. For example, when using the unconstrained Tactic in the robot soccer domain (Figure 6.14), many solutions contain sequences where the controlled robot would completely stop and wait for the ball to stop

rolling, while being in the midst of out-dribbling an opponent, because the planner was allowed to select the "wait" Skill at any point in time. Although, the overall sequence still eventually reaches the goal state, such a solution would be highly undesirable in a real robot soccer environment.

In computer animation domains, the resulting visual behavior of the bodies is especially important. The reason why Table 6.3 omits results in the pool table and many-dice domains is because the planner is not able to generate plausible results for these domains without the behavioral model provided by the manual Tactic. For example, in the many-dice domain, the Tactics model (see Figure 6.9) is designed to achieve the illusion of physically plausible movements of the dice, by only allowing forces and torques to be applied during certain key-events, such as collisions. Removing these manually defined constraints, and letting the planner freely choose when to transition between Skills, yields solutions that reach the goal state through a visually implausible sequence of events, such as dice changing their directions in mid-flight.

The pool table domain further demonstrates how crucial the manually defined Skill transitions in the Tactics model are for enforcing visually plausible solutions. The chained Tactics model (see Figure 6.7) ensures that the intermediate ball (labeled C in Figure 6.8) only is actuated toward the target *after* being hit by the cue ball (A). Without such manual constraints, the planner could simply generate solutions where the intermediate ball starts rolling even before being hit by the cue ball, which makes no sense given the rules of pool, and therefore defies the animation's goal of being visually plausible.

Conclusively, Tactics should be seen not only as a means for increasing planning efficiency through informed decision making, but also as an important behavioral control tool for constraining the visual sequences of the bodies.

6.2.5 Comparison to Traditional Skills and Tactics

In addition to analyzing our algorithm's performance and showing visual solutions, we are also interested in comparing the qualitative performance of our planning approach to traditional reactive control methods as they are currently used in many robotics applications, such as robot soccer. To do so, we designed an experiment in the simulated soccer environment where the controlled robot body and the soccer ball are placed at a randomly initialized location. The controlled body's goal is to deliver the soccer ball into the goal box which is protected by two reactive defenders. We executed a traditional, deterministic version of Skills and Tactics on this controlled body over multiple randomly initialized trials. The deterministic control approach followed a simple reactive soccer policy of dribbling the ball and shooting it into the corner of the goal box with the widest opening. We

compared the success rate (resulting in a goal) of the deterministic execution with trials generated with our BK-BGT approach and the Tactic shown in Figure 6.5, using different maximum search tree sizes. Table 6.4 shows the results.

Method	Tree Size	Success
Deterministic Tactic	n/a	30%
BK-BGT	1000	25%
BK-BGT	2500	40%
BK-BGT	5000	55%
BK-BGT	10000	60%

Table 6.4: Average success rate in a simulated "attacker vs. two defenders" scenario. Each row was computed from 20 randomly initialized trials.

We can see that even with relatively small search tree sizes, we obtain a greater average success rate than traditional methods. The reason for this outcome is that our physics-based planning approach has the power to find many intricate, physics-based solutions which the reactive version will never execute based on its fixed policy.

6.3 Chapter Summary

In this chapter, we have evaluated and compared the BK-RRT and BK-BGT algorithms empirically in several simulated domains. We have presented concrete Tactics models, visual results, and an exhaustive performance analysis. We have furthermore evaluated a hybrid planning scheme, the impact of the RollBack function, and an unconstrained Tactics approach.

In the following chapters, we discuss how our introduced planning approaches can be extended to be efficiently applied in uncertain and real-time robot environments.

Chapter 7

Planning in Real-World Dynamic Environments

In the previous chapters, we have introduced our Skills and Tactics-driven, physics-based planning approaches and have demonstrated their performance in fully predictable, simulated domains. In this chapter, we address the question of how to integrate these planning algorithms into a real-world robot system. In particular, we introduce a real-world version of the robot soccer domain as a testbed for demonstrating how to efficiently plan in an adversarial and not fully predictable environment, requiring real-time decision making. We present an *anytime planning* version of the BK-BGT algorithm that guarantees to deliver partial planning solutions within a fixed finite amount of planning time. Finally, using the same robot platform, we introduce a real-world version of the robot minigolf domain, and demonstrate our planner's execution timing capabilities in minigolf courses containing a moving obstacle body.

7.1 Physics-Based Planning in the CMDragons Multi-Robot System

We evaluate our physics-based planning algorithms using the Carnegie Mellon *CMDragons* multi-robot system. The CMDragons platform is a collaborative effort and builds upon the research and development contributions of its current and past team members [18, 87]. Special acknowledgment goes to James Bruce, for developing most of the current CMDragons software framework, and Mike Licitra for developing the current CMDragons robots.

Developed for its primary application of playing robot soccer in the RoboCup Small Size League (SSL), the CMDragons system consists of seven homogeneous robots that are wirelessly controlled by an offboard computer. Robot and ball localization is provided by two overhead-mounted cameras connected to the external computer, running the *SSL-Vision* open source vision system (see Appendix B). Each robot (see Figure 7.1) is approximately $18cm$ in diameter and features a 4-wheeled omni-directional drive system allowing stable driving speeds of over $2m/s$. Each robot is equipped with two solenoid-based kicking devices, allowing it to kick a golf ball at variable strength, flat at up to $15m/s$, or at an upward angle to a landing distance of about $4.5m$. Each robot furthermore carries a rubber-coated motorized "dribbler-bar" that is able to exert back-spin onto the golf ball for manipulation purposes. The entire CMDragons system is explained extensively in Appendix A.

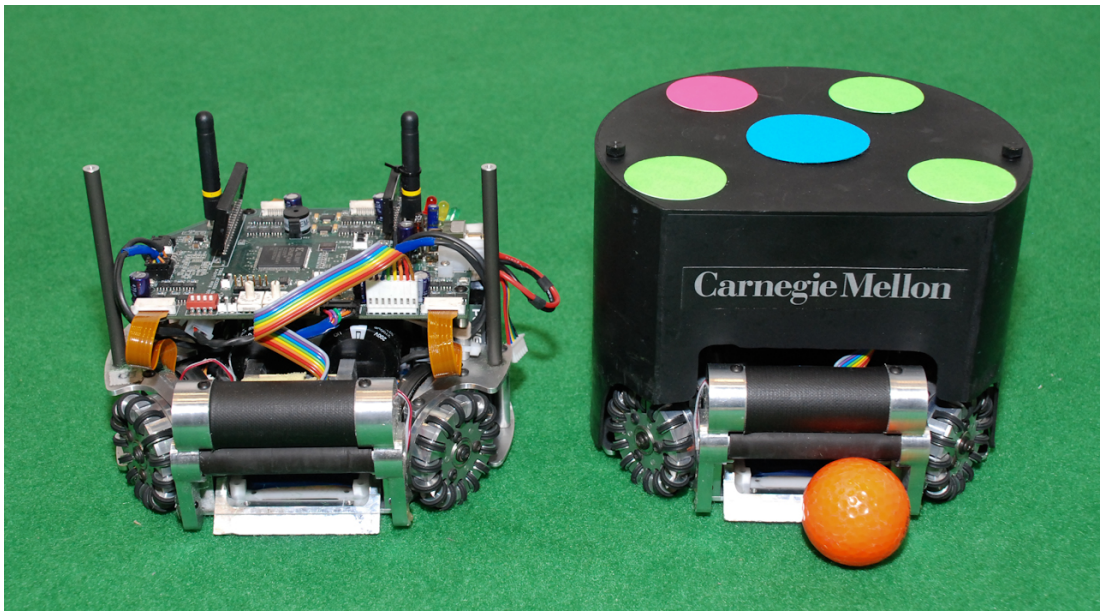


Figure 7.1: A CMDragons robot shown with and without its protective cover.

7.1.1 Physics-Based Planning as a Single-Robot Behavior

In order to evaluate our physics-based planning approaches using the CMDragons system, we must first investigate how to best integrate our planner into the existing system infrastructure. CMDragons features a modular software infrastructure (see Figure 7.2) consisting of the four components *Vision*, *Server*, *Soccer*, and *GUI*, that are interconnected via UDP networking. The Server module receives the raw positions and orientations of all robots and the ball from the Vision module at 60Hz intervals. The server's Tracker then post-processes this raw data using a Kalman filter [89], providing a reasonably accurate

estimate of the positions and velocities of the robots and ball. These state estimates are then provided to the Soccer module's world model. The Soccer module implements all the intelligent robot soccer decision making, including multi-robot strategy, and single-robot control. The output of the Soccer module is a set of low level motion and actuation commands that are sent back to the Server, which in turn communicates them to the robots via the wireless radio.

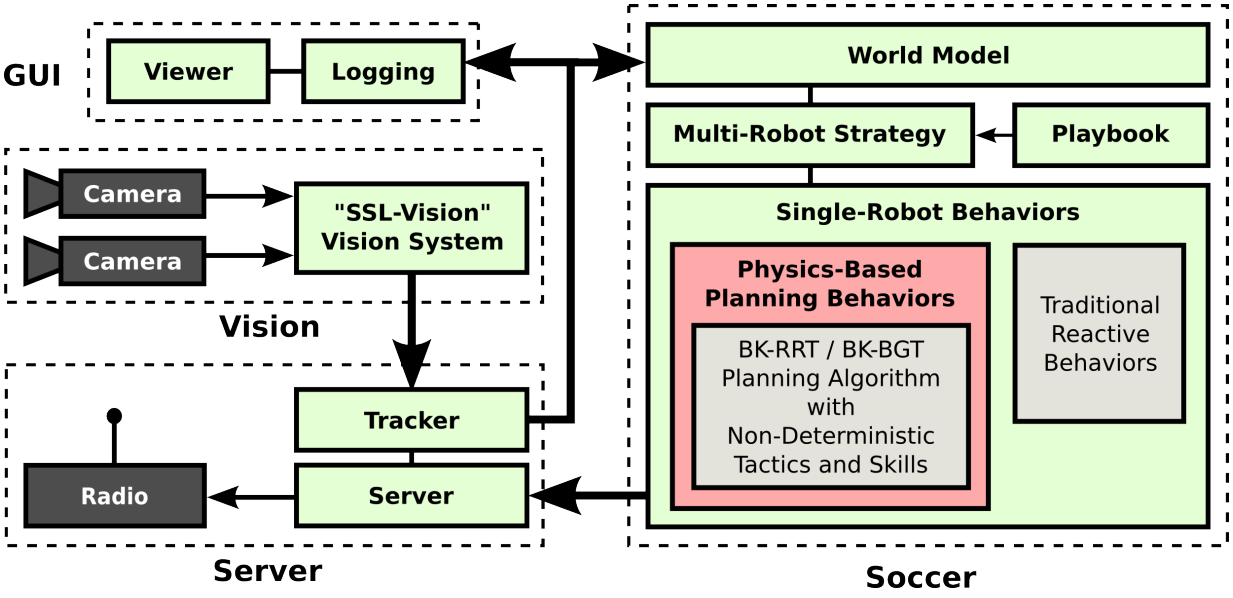


Figure 7.2: The CMDragons system infrastructure, including the integration of physics-based planning.

The Soccer component relies on the Skills, Tactics, and Plays (STP) [13] model to perform its intelligent decision making. The multi-robot strategy layer uses a *Playbook*, consisting of multiple *Plays*, each describing a set of individual "roles" to be distributed among the available robots. Plays are selected based on a set of pre-conditions. For example, a "defensive" play will only be assigned if its precondition is met that the opponent currently has ball possession. If multiple plays are simultanously applicable, then one of them is chosen randomly.

Once selected, a play will assign the individual single-robot behaviors to the robots. Traditionally, in the STP-model, these individual behaviors are implemented as reactive Skills and Tactics that do not perform any physics-based planning. Each such reactive behavior typically investigates the current state of the world, performs some analytic decision making (also see Appendix A), optionally invokes a traditional navigation planner (that is unaware of the tactical goals and not capable of planning multi-body interactions or ball manipulations), and then returns a final robot motion command to the Server.

We integrate our physics-based planner within this single-robot component of the Soccer module. In particular, we introduce a new type of *Physics-Based Planning Behaviors* that can exist side-by-side to the traditional reactive behaviors already present in the system. Internally, each physics-based planning behavior carries its own non-deterministic Skills and Tactics model (see Chapter 4) and uses the BK-RRT/BK-BGT algorithm (see Chapter 5) to search over possible action sequences sampled from that model. The physics-based planning behavior then forwards the planner's solution to the Server for execution, which we explain in more detail in Section 7.2.3.

7.1.2 Modeling a CMDragons Robot

In order to perform physics-based planning in the CMDragons system environment, it is crucial that the robot and its environment are accurately modeled. The CMDragons server component is able to provide us with accurate estimates of the location, orientation, and velocity of the robots and ball, thus providing us all the values to define the initial body state parameters $\hat{r}_1, \ldots, \hat{r}_n$ that are part of x_{init}. However, we also need to define the fixed rigid body parameters $\bar{r}_1, \ldots, \bar{r}_n$ that describe the bodies' shapes, masses, and material properties.

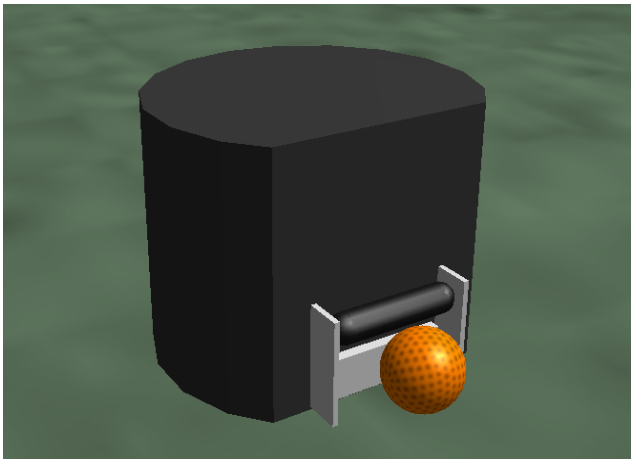


Figure 7.3: The rigid body models of a CMDragons robot and ball.

Figure 7.3 shows a rendering of the robot and ball rigid bodies. The CMDragons robot rigid body is modeled using a convex triangulated mesh in combination with other convex body primitives. The golf ball is approximated by using a simple sphere.

In order to deliver solutions that are executable in the real world, it is important that our planner has an adequate model of the robot's ball manipulation capabilities. A particularly

challenging issue is to correctly model the robots' dribbler bar. One possible approach is to model the dribbler implicitly, by defining an actively actuated, rotating, high-friction rigid body cylinder that is connected to the robot rigid body by using a rotary joint. In practice however, this approach led to simulation instability and failed to accurately model the dribbling behavior, possibly due to the accumulation of joint-error and an insufficient simulated torque transfer between the bar and the ball.

A more stable approach, is to model the dribbling action explicitly: whenever the ball is in contact with the dribbler-bar, the currently active Skill can directly apply a torque to the ball body. The exact dribbling behavior can be controlled by modifying the static rigid body parameters that influence the ball's dynamics. Figure 7.4 shows a diagram of the dribbling model. Model parameters that strongly affect the ball's behavior during dribbling are the applied torque (*torque*), the friction in relation to the floor ($fric_2$), and the friction in relation to the dribbler bar ($fric_1$). The dribbler bar uses an anisotropic friction model, defining low friction against the vertical direction of the bar, thus roughly simulating a freely spinning bar, and higher friction against the horizontal direction of the bar, thus modeling the high friction of the bar's rubber. In addition to friction, we can control the ball's coefficient of restitution against the dribbler ($rest_1$) and the floor ($rest_2$) respectively. Naturally, $rest_1$ will be a fairly low coefficient, thus roughly modeling the dribbler's softness and damping, whereas $rest_1$ is likely to be a slightly higher coefficient to model the bounciness of the carpet. Note that, given our formal domain model (see Section 2.1.1), the interaction coefficient of two materials is actually computed as the mean of the individual coefficients of the two materials involved. For example, the restitution parameter between the ball and the floor, $rest_2$, is computed as the mean of the two individual restitution parameters of the ball and floor rigid bodies (i.e., $\bar{r}_{\text{ball}}.\phi_{\text{Rest}}$ and $\bar{r}_{\text{floor}}.\phi_{\text{Rest}}$). We manually tweaked all of these individual parameters of the dribbling model to best resemble the robot's actual behavior. In Chapter 8 we present *automated* parameter optimization techniques that can be used to further improve physics model accuracy.

We model the robot's driving motions in an explicit manner as well. Instead of modeling the wheels as separate rigid bodies that impose friction on the floor, we model the robot as having a flat, low friction base-plate. The Skills then directly apply forces and torques to the robot body to simulate its motions. Kicks and chip-kicks are simulated in a similar fashion, by exerting linear force impulses directly on the ball body.

7.2 Single-Shot vs. Anytime Planning

Another interesting question, when planning in real-world environments, is *when* and *how long* to plan. Generally, we can distinguish between two fundamental paradigms: *single-*

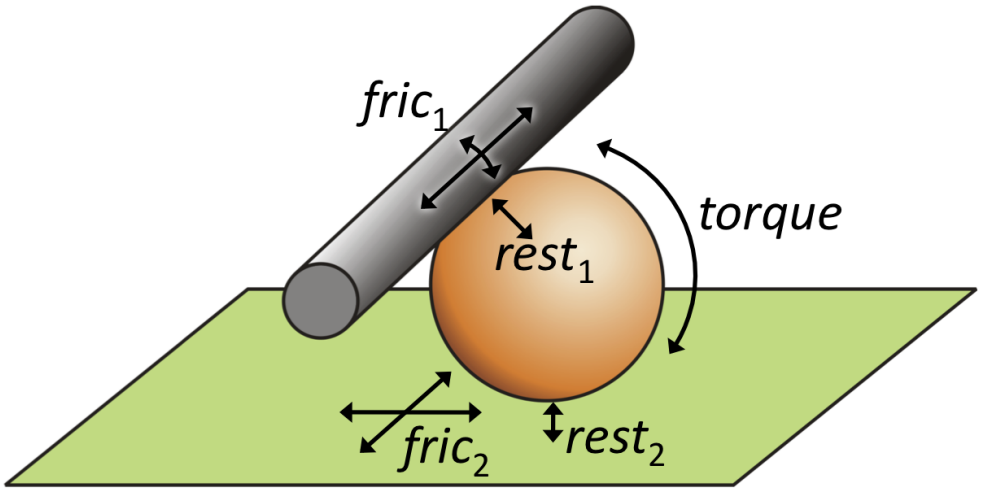


Figure 7.4: The physical model of the dribbler bar.

shot planning and *anytime* planning. We explain each of these approaches and discuss their trade-offs.

7.2.1 Single-Shot Planning

By single-shot planning, we refer to the traditional planning task of finding a complete solution sequence from an initial state x_{init} to some goal state $x_{\text{goal}} \in X_{\text{goal}}$. The key requirement for single-shot planning is that there exists a clearly defined set of goal states that can either be reached or not reached. Reaching a valid goal state would be considered a *success*, anything else would be considered a *failure*. The BK-BGT and BK-RRT algorithm versions as presented in Chapter 5 use this single-shot paradigm: the algorithms return a solution sequence if a goal state is found, or failure if the maximum allowed number of iterations is exceeded.

Of course, one problem of single-shot planning is that it is difficult to predict *how long* the planner will need to find a solution that leads to a goal state. As we have seen in our results in Section 6.2, planning times for successful single-shot solutions in our testing domains are typically in the range of seconds and furthermore have a significant amount of deviation between individual trials, mostly due to the random nature of our planning approaches. Requiring such a long and not fully predictable amount of time to find a solution might be an acceptable trade-off if the solutions do indeed lead to a valid goal state when executed, and in particular if the domain does not change unpredictably during the course of planning. Minigolf is a good example for such an environment: it is perfectly acceptable for the player to require several seconds of planning time before executing its move. Because the golf-ball

and the environment do not move during planning (with a few predictable exceptions, as we will discuss in Section 7.4), the resulting solutions remain executable, even if they are delivered after a several second delay of planning time. However, there are many other domains where this type of timing unpredictability of single-shot planning is simply not acceptable.

7.2.2 Anytime Planning

Robot soccer is an example of a domain where single-shot planning does not suffice. Because the environment in robot soccer changes constantly and in not fully predictable ways, it is extremely important that planning time is reduced to a minimum, so that a solution delivered by the planner is still applicable to the state of the world at the time of execution. Single-shot planning, however, cannot guarantee that a solution is found within such a short and finite amount of time, and would report planning failure if aborted prematurely. In a domain like robot soccer, such planning failures could have disastrous consequences, letting the controlled robot be "dead in the water" without any actions to execute. Instead of risking the occurrence of such complete planning failures, it would be much preferable that the robot would always have *some* plan to execute, even if this plan is only partial and does not reach a goal state from the set X_{goal}.

Anytime planning has the ability to *always* provide a plan, even when interrupted before an ultimate goal state from the set X_{goal} is reached [27, 37]. Instead of returning a failure, anytime planning returns the sequence of actions leading to the *best* state encountered during the search so far. The notion of *best* is defined through the evaluation function `EvalState`(x) that acts as a heuristic to quantify how "good" a particular state x is. We define the output of `EvalState` to be a single continuous value, with smaller values indicating "better" states. Thus, in an anytime planning scenario, the planner's objective should be to find actions that lead to states which minimize the output of `EvalState`. The internal definition of `EvalState` has to be provided by the user and depends on the domain and the particular goals.

Although the presented BK-RRT and BK-BGT algorithms in Chapter 5 were formulated in a single-shot manner, we can easily transform either algorithm to their respective anytime versions. Algorithm 10 shows *BK-BGT-Anytime*, the anytime version of standard BK-BGT. This algorithm adds two new key variables to the outer planning loop:

- x_{best} : the best state found so far,
- bestEval : the evaluation value of x_{best}, i.e. the value returned by `EvalState`(x_{best}).

Algorithm 10: BK-BGT-Anytime

Input: Domain: d, Initial state: x_{init}, set of goal states: X_{goal}, timestep: Δt, validation function: `Validate`, state evaluation function: `EvalState`, BGT ratio: μ, max iterations: z.

```
tree ← NewEmptyTree();
tree.AddNode(x_init);
bestEval ←EvalState(x_init);
x_best ← x_init;
busy ← false;
y ← ∅;
iter ← 1;
while (PlanningInterrupted() = false) and (iter ≤ z) do
    iter ← iter + 1;
    if busy = true then
        x ← x′;
    else
        x ← SelectNodeBGT(tree, μ); // See Sec. 5.4
    end
    ⟨x′, L⟩ ← TacticsDrivenPropagate(x, y, Δt, d); // See Sec. 5.3
    busy ← x′.b_∧;
    if Validate(x′, L) then
        tmpEval ←EvalState(x′);
        if tmpEval < bestEval then
            bestEval ← tmpEval;
            x_best ← x′;
        end
        tree.AddNode(x′);
        tree.AddEdge(x, x′, a);
        if x′ ∈ X_goal then
            return TraceBack(x′, tree);
        end
    else
        busy ← false;
    end
end
return TraceBack(x_best, tree);
```

The core of the algorithm is relatively similar to BK-BGT. One key modification, however, is the exit condition of the planning loop. Now, instead of only aborting after a predetermined number of iterations z, the planner will abort at any iteration if the boolean interrupt function `PlanningInterrupted` returns true. If aborted, the planner will return the sequence of actions that led to the state x_{best}. Typically, the current output value of the interrupt function should be controlled by the software framework that is external to the actual planning algorithm and that can therefore request when the planner should stop planning and deliver its current *best* solution. During planning, the algorithm ensures that x_{best} always reflects the state with the lowest output of the evaluation function `EvalState` found so far, by comparing the evaluation of any newly validated and added state x' to the current value of bestEval, and updating x_{best} with x' if necessary.

Note that, for simplicity, we have left out the RollBack function (see Section 5.3.2) from the anytime BK-BGT algorithm, because a state rollback could include a deletion of the current state x_{best}, and therefore would also require a rollback of the x_{best} value to its original state at the point before the greedy state expansion caused by the busy flag took place (see 5.3.1). Also note, that BK-RRT can be adapted to its anytime version using exactly the same modifications as shown for BK-BGT.

The beauty of the anytime algorithm is that with increasing planning times, the quality of x_{best} is likely to increase, as better nodes are found. The trade-off however, is that the anytime algorithm requires the definition of a well-suited evaluation function that is able to accurately express the notion of a state's quality, given the goals of the domain. Defining such a continuous evaluation can be more difficult than only defining a set of ultimate goal states as is the case in single-shot planning.

7.2.3 Planning and Execution in a Closed-Loop System

Like many real-world robot systems, the CMDragons system uses a fixed-interval, *closed-loop* control approach consisting of *sensing*, *thinking*, and *execution*. The term *thinking* in this case may describe any type of autonomous decision making, including its traditional reactive behaviors as well as its fully featured physics-based planning behaviors with Skills and Tactics. We refer to each iteration of this control loop as a *frame*. The CMDragons system loop runs in sync with the SSL-Vision system at a 60Hz frame-rate, providing approximately 16.7 milliseconds of computational thinking time per frame.

Planner Invocation Strategies

An important question is when exactly to invoke the planner in such a closed-loop system. Figure 7.5 shows an example of possible periods for planning and plan execution, using an anytime planning strategy. Here, the actual planning task is always performed *within* a single frame period, allowing the start of plan execution already within the same frame that the planning was performed, thus minimizing planning latency. Note, that in this particular example, the produced plans are executed for 4 frames, before replanning. This replanning interval can be adjusted depending on the uncertainty of the domain. Generally, choosing a shorter replanning interval will yield more robustness against uncertainty. However, shorter replanning intervals can also introduce a certain amount of oscillations in the controlled robot's motion, because the randomized nature of the planner might deliver rapidly varying plans between replan iterations. Also, note that, due to the nature of anytime planning, it is not guaranteed that the resulting plans are always sufficiently long to enforce a fixed replanning interval. Under certain conditions, the solution delivered by the planner might only contain a few frames, thus requiring replanning as soon as the previous plan is fully executed.

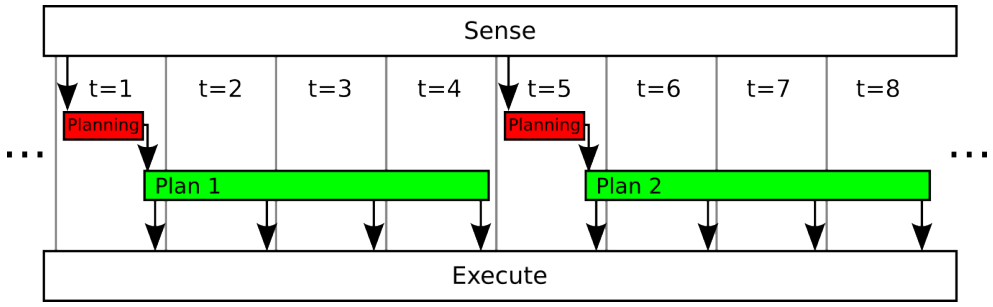


Figure 7.5: Sequential anytime replanning, using a single-frame planning duration and a 4-frame replanning interval (t indicates the frame number).

Execution Strategies

When implementing a physics-based planner within a closed-loop system (such as the CM-Dragons platform), a remaining question is how to actually execute the solutions delivered by the planner. Recall, that the planner delivers a sequence of states interleaved with actions of the form $\langle x_{\text{init}}, a_1, x_2, a_2, x_3, \ldots, x_{\text{goal}} \rangle$. Generally, there are two fundamental alternatives:

- **Action-Driven (Open-Loop) Plan Execution**
  Action-driven plan execution uses the individual actions $(a_1, a_2, \ldots)$ of the solution

sequence directly for execution on the robot. The states that were returned as part of the solution sequence $(x_{\text{init}}, x_2, \ldots, x_{\text{goal}})$ are ignored for execution purposes.

Recall, that an individual body's sub-action $\hat{a} \in a$ consists of a $\langle \textit{force}, \textit{torque} \rangle$ pair. When being executed on a physical robot, the actions are typically first converted into accelerations (e.g., $\textit{acceleration} = \textit{force}/\textit{mass}$) and then directly applied on the robot's hardware.

A problem with this action-driven execution scheme is that it represents an *open-loop* control, where the sequence of actions is executed without monitoring whether the resulting sequence of real-world robot states corresponds with the sequence of states that the planner predicted. Such a control scheme can be acceptable if replanning occurs very frequently and if the physical planning model of the robot's actuations matches its real-world actions very closely. However, in most real-world scenarios, a solely action-driven execution scheme can lead to a quickly accumulating execution error and, ultimately, to unsuccessful executions.

- **State-Driven (Closed-Loop) Plan Execution**
  State-driven plan execution attempts to prevent the problem of execution error accumulation that can occur with action-driven plan execution. Instead of blindly executing the actions of the solution sequence, state-driven execution attempts to re-create the same sequence of *states* that were provided by the planner's solution sequence (i.e., $x_{\text{init}}, x_2, \ldots, x_{\text{goal}}$). To do so, the actual actions to be executed by the robot are generated on the fly during execution, by repeatedly invoking a *controller* that takes as an input the robot's current observed real-world state and the desired state from the solution sequence, i.e.:

$$\hat{a}_{\text{robot}} \leftarrow \texttt{Controller}(\texttt{ObserveRealRobotState()}, (x_i.\hat{r}_{\text{robot}})).$$

  The above controller function is called during each frame i of the solution sequence. The internal mechanism of the controller depends on the particular types of actuators of the robot, but typical examples include PD-controllers, or in the case of the CMDragons system, a trapezoidal motion controller (also see Section 5.3.4). The key however is, that such motion controllers typically require only a minimum amount of computation and do not need to perform any higher level planning, because the difference between the desired solution state (as delivered by the actual planner) and the observed real-world states are never allowed to significantly deviate and therefore can be resolved with simple motion control techniques.

  The great advantage of such closed-loop control is that it allows execution of very long sequences without an increasing accumulation of execution error by the controlled body. Nevertheless, it is important to point out that even a closed-loop controller cannot directly influence the real-world execution of passive or foreign-controlled

bodies. Therefore, the likelihood of execution success in environments with unpredictable non-controlled bodies is still greatly determined by the plan length, often leaving frequent replanning as the only solution.

7.3 Robot Soccer

We test our planning approaches in the Small Size League robot soccer domain. The robot soccer domain has many interesting challenges, but one particularly difficult problem from a motion planning perspective, is ball dribbling. Dribbling is the task of *purposefully* manipulating the ball over sustained periods of time, without loosing control over it. *Purposeful* means that the dribbling task should make some progress toward the overall goals of the game. In robot soccer, this generally means several things:

- maintain control of the ball,
- prevent opponent robots from gaining possession of the ball,
- reach a state where the robot has a clear shot towards the opponent's goal box (or towards a teammate).

Performing this dribbling task is very difficult for several reasons. As discussed elaborately in Section 2.4, solving manipulation tasks involving passive bodies is a challenging planning problem to begin with. Additionally, in robot soccer, the controlled robot body has to deal with opponent bodies that are actively trying to stop it from succeeding and that are never fully predictable in their actions. Finally, the Small Size League is extremely fast paced. Actions have to be computed with a minimum amount of latency in order to still be applicable at the time of execution.

Traditional control techniques used in robot soccer, such as deterministic reactive controllers without physical modeling or planning capabilities, fail to solve the dribbling task. The core reason for this is that it is not clear how to manually define a programmatic control policy that produces useful actions under all of the possible continuous input states in the dribbling problem. The best action for the robot to take may depend not only on the precise combination of the robot, ball, and opponent positions and orientations, but also on each of their current velocities, and furthermore on the intricate physical interactions between the robot and ball. The de facto result is that none of the current teams in the RoboCup Small Size League are able to purposefully and robustly dribble the ball in the presence of opponents.

In order to solve the dribbling problem, we take a fundamentally different approach from traditional reactive techniques, and instead use physics-based, Tactics-driven anytime planning to implement the dribbling behavior. We introduce a new single-robot behavior called *PhysicsDribble* to the CMDragons system. This PhysicsDribble behavior is invoked by the CMDragons system whenever one of our robots has obtained possession of the ball, but does not yet have a clear shot to the opponent's goal box. In the following subsections we explain the detailed implementation of the PhysicsDribble behavior, in particular its Tactics model, and its anytime state evaluation function. Finally, we present results where we compare the new PhysicsDribble behavior with the traditional reactive behavior used previously in the CMDragons system.

7.3.1 Tactics Model

In order to efficiently find solutions to the dribbling problem, our planning approach requires the definition of effective sampling-based Skills and Tactics that are able to constrain the size of the searchable action space in a task-oriented manner. Figure 7.6 shows the the non-deterministic Tactics model for the controlled robot in the PhysicsDribble behavior. The model consists of three Skills, each implementing a dribbling control strategy, but each differing in the particular sampling distribution from which they choose the direction of dribbling. Transition probabilities between these Skills are defined by the values of p and q, where p is the probability of dribbling towards some random point, $(1-p)$ is the probability of dribbling away from the nearest opponent body, and q is the probability of dribbling toward the goal box, but only if there currently exists a clear shot from the ball to the goal box. Note, that all applicable transition probabilities are always auto-normalized by the transition function g (see Section 4.3), and therefore do not need to sum to one.

After selecting its dribble target point using its unique sampling distribution, each of the three dribbling Skills uses the same `ComputeDribbleAction` function (see Algorithm 11) to compute the action for the controlled robot. Note, that $\angle$ denotes the angle of a 3D vector if down-projected to 2D global frame field coordinates, and `AngDiff` computes the shortest difference between two angles. We use subscript shorthand notations to refer to the controlled robot ($\hat{r}_{\text{robot}}$), the ball ($\hat{r}_{\text{ball}}$) and the opponent robot that is currently nearest to the ball ($\hat{r}_{\text{opp}}$). The `ComputeDribbleAction` function furthermore requires the definition of multiple constants: the robot and ball radii ($\mathsf{radius}_{\text{robot}}$, $\mathsf{radius}_{\text{ball}}$), and the minimum and maximum desired forward velocities during dribbling ($\mathsf{minForwardVel}$, $\mathsf{maxForwardVel}$). Finally, the `ramp` function linearly maps an input range to some bounded output range:

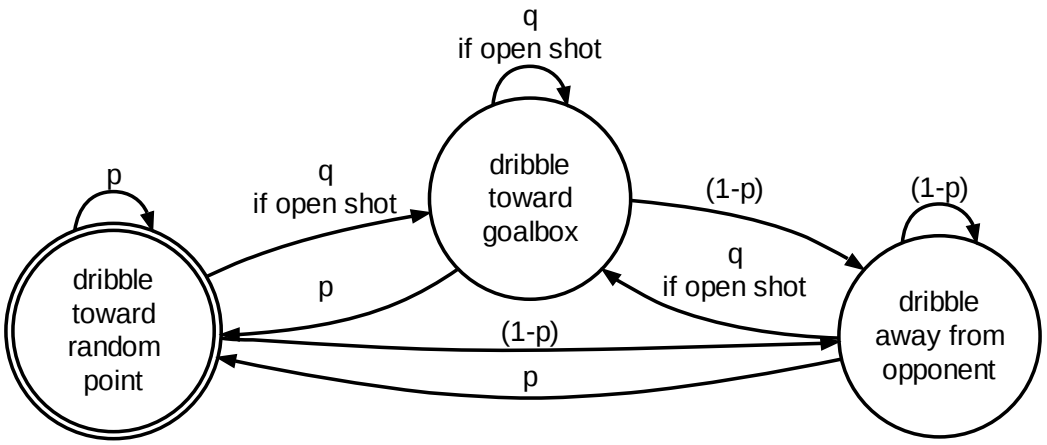


Figure 7.6: The Tactic used for the controlled robot in the PhysicsDribble behavior.

$$\texttt{ramp}(\mathsf{in}, \mathsf{in}_{\min}, \mathsf{out}_{\min}, \mathsf{in}_{\max}, \mathsf{out}_{\max}) =$$
$$\begin{cases} \mathsf{out}_{\min} & \textbf{if } \mathsf{in} < \mathsf{in}_{\min} \\ \mathsf{out}_{\max} & \textbf{if } \mathsf{in} > \mathsf{in}_{\max} \\ \mathsf{out}_{\min} + (\mathsf{out}_{\max} - \mathsf{out}_{\min})(\mathsf{in} - \mathsf{in}_{\min})/(\mathsf{in}_{\max} - \mathsf{in}_{\min}) & \textbf{otherwise}. \end{cases}$$

The `ComputeDribbleAction` function takes as input the current state of the world (x) and the dribble target point (targetPoint) that was sampled by the current Skill. It then computes the relative angle towards its dribbling target (deltaAngle), and uses linear ramping to compute the desired forward velocity component of the robot (targetVel.x), slowing down to minForwardVel when moving directly away from the target, and speeding up to maxForwardVel when directly approaching it. Next, the algorithm computes a sideways velocity component (targetVel.z) based on the current angular velocity of the robot. This sideways velocity component effectively transforms the center point of rotation from the center of the robot to the ball itself, thus allowing the robot to "turn around the ball", while pushing it. Finally, the `ComputeForceFromVelocity` function converts the desired target velocity into a force-vector by comparing the current velocity to the target velocity and applying a trapezoidal motion controller (also see Section 5.3.4) to generate the resulting force. Similarly, the torque is computed by applying a trapezoidal motion controller on the current angular distance, to reach the target angle as quickly as possible within some pre-defined rotational acceleration, deceleration, and velocity limits.

Algorithm 11: `ComputeDribbleAction`$(x, \mathsf{targetPoint})$

```
// This algorithm assumes a right-handed 3D coordinate system, i.e.
   seen form the robot's local reference frame, the x-axis points
   forward, the y-axis points upwards, and the z-axis points right.
```

targetAngle $\leftarrow \angle(\mathsf{targetPoint} - x.\hat{r}_{\text{ball}}.\alpha)$;
currentAngle $\leftarrow \angle((x.\hat{r}_{\text{robot}}.\beta)\ [1,0,0])$;
deltaAngle $\leftarrow$ `AngDiff`(targetAngle, currentAngle);
targetVel $\leftarrow [0,0,0]$;
`// Compute forward target velocity:`
targetVel.x $\leftarrow$ `ramp`$(|\mathsf{deltaAngle}|, 0, \mathsf{maxForwardVel}, \pi, \mathsf{minForwardVel})$;
`// Compute sideways target velocity:`
targetVel.z $\leftarrow ((\mathsf{radius}_{\text{robot}} + \mathsf{radius}_{\text{ball}})(x.\hat{r}_{\text{robot}}.\omega.\mathsf{y}))$;
$\hat{a}_{\text{robot}}.force \leftarrow$ `ComputeForceFromVelocity`$((x.\hat{r}_{\text{robot}}.\gamma), \mathsf{targetVel})$;
$\hat{a}_{\text{robot}}.torque \leftarrow$ `ComputeTorqueFromAngle`$((x.\hat{r}_{\text{robot}}.\omega), \mathsf{deltaAngle})$;
return $\hat{a}_{\text{robot}}$;

7.3.2 Opponent Prediction Model

We have defined the Tactic and its Skills to generate the actions for the *controlled* robot body. Note however, that some of the controlled body's decision making depends on the extrapolation of the foreign-controlled opponent body's position (i.e., the "dribble away from opponent" Skill in Figure 7.6). Therefore, in order to predict the opponent body's motions, we model its likely actions via its own *deterministic* Tactic. Because we are unaware of the true internal strategies of the opponent robot, we employ a relatively generic opponent Tactic, consisting of only a single deterministic Skill that implements a "drive toward ball" control strategy for the opponent robot.

Of course, such a simplistic opponent model is not likely to perfectly predict the exact opponent's behavior. However, recall that for the robot soccer domain we use an anytime planning approach that is able to replan at frequent intervals (60Hz). Therefore, with replanning enabled, any incorrect predictions about the world (including the opponent model, or other physical predictions such as ball/robot interactions) are not given the chance to accumulate to longer term prediction errors, because before each replanning step the true state of the world is re-observed thanks to the CMDragons vision system.

7.3.3 State Evaluation Function

To implement an anytime planning approach in the robot soccer domain, we need to define how the `EvalState` function (see Section 7.2.2) computes its output value. For the PhysicsDribble behavior, we define the `EvalState` function to compute the negated product of multiple individual evaluation functions that are specific to the goals of the dribbling task:

$$\texttt{EvalState}(x) = 1 - (\texttt{EvOpp}(x)\ \texttt{EvHandling}(x)\ \texttt{EvBoundary}(x)\ \texttt{EvAim}(x)\ \texttt{EvTime}(x))\,.$$

Each of these individual evaluation functions returns a result normalized to the range $[0, 1]$, where a zero output indicates the "worst" possible evaluation of a state and a one indicates the "best" possible evaluation. Note, that the following definitions use shorthand notations to refer to the controlled robot ($\hat{r}_{\text{robot}}$), the ball ($\hat{r}_{\text{ball}}$) and the opponent robot that is currently nearest to the ball ($\hat{r}_{\text{opp}}$).

- **Opponent Distance to Ball (`EvOpp`)**
  `EvOpp` evaluates the distance between the ball and the closest opponent body. It returns zero if the opponent is in direct contact with the ball, and linearly ramps up the value with increasing distance (using the `ramp` function defined in Section 7.3.1), until some user-defined maximum distance threshold (maxDistBallOpp):

  $$\texttt{EvOpp}(x) = \texttt{ramp}\,(\texttt{distBallOpp}(x), 0.0, 0.0, \mathsf{maxDistBallOpp}, 1.0)\,,$$

  where `distBallOpp` is a function computing the physical distance between the ball body and the opponent body, assuming predefined constants for the radii of the opponent robot and ball:

  $$\texttt{distBallOpp}(x) = \texttt{max}\,(0.0, |(x.\hat{r}_{\text{ball}}.\alpha) - (x.\hat{r}_{\text{opp}}.\alpha)| - (\mathsf{radius}_{\text{opp}} + \mathsf{radius}_{\text{ball}}))\,.$$

- **Ball Handling (`EvHandling`)**
  `EvHandling` evaluates the quality of ball handling by computing the distance of the ball towards the controlled robot's dribbler bar. Algorithm 12 shows the implementation of the `EvHandling` function. The algorithm first computes the relative ball position toward the robot's dribbler in the robot's local coordinate frame (robotBallLocal). The algorithm relies on predefined constants for the dribbler's width (dribblerWidth) and frontal offset to the robot's center (dribblerFront). An additional user-defined threshold (distControlled) defines how close the ball needs to be to the dribbler bar to be considered "in control". In that case, the algorithm returns 1 immediately, otherwise, it will compute its output value based on the distance of the ball towards the dribbler bar which is scaled by a penalty scalar constant in the range from [0,1] (badHandlingPenalty) to further emphasize that such a state is undesirable.

Algorithm 12: `EvHandling`(x)

```
// This algorithm assumes a right-handed 3D coordinate system, i.e.
   seen form the robot's local reference frame, the x-axis points
   forward, the y-axis points upwards, and the z-axis points right.
```

$\mathsf{robotBall} \leftarrow (x.\hat{r}_{\text{ball}}.\alpha) - (x.\hat{r}_{\text{robot}}.\alpha)$;
$\mathsf{robotBallLocal} \leftarrow ((x.\hat{r}_{\text{robot}}.\beta)^T\ \mathsf{robotBall}) - [\mathsf{dribblerFront}, 0, 0]$;
if $(\mathsf{robotBallLocal.x} > 0)$ **and** $(\mathsf{robotBallLocal.x} < \mathsf{distControlled})$ **and**
$(|\mathsf{robotBallLocal.z}| < \mathsf{dribblerWidth}/2)$ **then**
  return 1;
else
  $\mathsf{robotBallLocal.z} \leftarrow \mathtt{max}(0, |\mathsf{robotBallLocal.z}| - (\mathsf{dribblerWidth}/2))$;
  $\mathsf{robotBallLocal.y} \leftarrow 0$;
  return $\mathsf{badHandlingPenalty}\ (1/(1 + |\mathsf{robotBallLocal}|))$;

- **Ball Distance to Field Boundary (`EvBoundary`)**
  `EvBoundary` evaluates the distance between the ball and the field boundary. First, the `distanceToBoundary` function computes the actual shortest distance to the field boundary. Then, in order to penalize states that will lead the robot to dribble the ball off the field, a linear ramping function is used which reduces its output value to close to zero when getting within a 10cm to the field boundary:

$$\mathtt{EvBoundary}(x) = \mathtt{ramp}(\mathtt{distanceToBoundary}(x.\hat{r}_{\text{ball}}), 0.1\text{m}, 0.001, 0.6\text{m}, 1).$$

- **Robot Heading (`EvAim`)**
  `EvAim` evaluates the aim of the robot. It invokes the `angleToGoal` function to compute the difference in angle between the controlled robot's current heading and the desired heading towards the opponent's goal box. This angular difference (measured in radians) is then used as the input to the ramping function:

$$\mathtt{EvAim}(x) = \mathtt{ramp}(|\mathtt{angleToGoal}(x)|, 0, 1, \pi, 0),$$

  where:

$$\mathtt{angleToGoal}(x) = \mathtt{AngDiff}(\angle(\mathsf{goalLocation} - (x.\hat{r}_{\text{ball}}.\alpha)), \angle((x.\hat{r}_{\text{robot}}.\beta)\ [1,0,0])).$$

  Note, that $\angle$ denotes the angle of the vector if down-projected to 2D global frame field coordinates, and `AngDiff` computes the shortest difference between two angles.

- **Short-Term Penalty (`EvTime`)**
  `EvTime` has the purpose to penalize short-term states. Recall that, in anytime planning, the state with the best evaluation is returned by the planner. However, this

could imply a scenario where the initial state x_{init} happens to also be evaluated as the *best* state. Although x_{init} might in fact be the "best" state from the perspective of the other evaluation functions, it would be much more desirable that the planner returns a solution of some minimum required length, so that the robot will actually have *some* actions to execute. The `EvTime` function attempts to achieve this by penalizing the evaluation of states that are too short term to be considered a useful result. It relies on two user defined thresholds: a *strict* minimum plan length (tMinStrict) and a *desired* minimum plan length (tMinDesired), where tMinStrict $<$ tMinDesired. The `EvTime` function then uses linear ramping to evaluate the quality of a state, based on its time index t:

$$\texttt{EvTime}(x) = \texttt{ramp}(x.t, \mathsf{tMinStrict}, 0, \mathsf{tMinDesired}, 1).$$

7.3.4 Results

We tested the PhysicsDribble behavior using the BK-BGT-Anytime algorithm with state-driven execution in the CMDragons system, both under lab conditions, and during the actual RoboCup 2009 competition.

To objectively evaluate the PhysicsDribble behavior, we compared it against the state-of-the-art CMDragons "attacker" behavior that uses a traditional reactive tactic to obtain the ball and score a goal. The experiment was setup as a 1-vs-1 robot soccer scenario. Each trial was started at the center of the field, with each robot on their own side of the field, in a symmetric configuration. A trial was aborted and restarted from its initial configuration if the ball went out of bounds. A score was counted and a trial concluded if the ball entered the goal.

The experiment consisted of two parts. First, as a baseline, we let both robots run the traditional attacker behavior. We performed 20 trials (switching sides after the first 10) and recorded the numbers of goals scored by each robot.

For the second part of the experiment, we assigned robot A to use the new PhysicsDribble behavior with BK-BGT-Anytime planning, while leaving robot B with the traditional "attacker" tactic. Again, we ran 20 trials, switching sides after the first 10.

Table 7.1 shows the result of the experiment, presenting the percentage of how many of the 20 goals were scored by robot A. As expected, both robots performed similarly in the baseline configuration, with robot A scoring 55%, and robot B scoring 45%. The 5% difference from a perfect balance lies within the margin of error of a 20 trial setup and is not considered statistically significant (t-test p-value of 0.76). When using the PhysicsDribble behavior, however, robot A scored 85% of the goals, whereas robot B scored 15%. Given

Robot A Behavior	Robot A Success Rate
Baseline (Reactive)	55%
PhysicsDribble	85%
Improvement	30 pp

Table 7.1: Success rate of reactive behavior (baseline) vs. physics-based planner. Robot B always used the reactive behavior.

the 20 trial experiment, this difference is considered statistically significant (t-test p-value of 0.039).

These quantitative results are furthermore underscored by the visual performance of the robot. Figure 7.7 shows an example sequence taken during the experiment where robot A runs the PhysicsDribble behavior. After obtaining the ball, the PhysicsDribble behavior quickly leads robot A to dribble the ball away from the opponent, temporarily even away from the opponent's goal box (images 2-4). Shortly thereafter, however, the behavior leads robot A to perform a rapid turn, while maintaining full control of the ball (images 5-6), bringing it quickly into position for an open shot, at which point the CMDragon's reactive "Kick" Tactic takes control and delivers the ball into robot B's goal box (images 7-8).

	Mean	StdDev	Median	Min	Max
Tree Size (nodes)	87.9	4.59	87	64	101
Solution Size (nodes)	10.7	3.13	13	6	13

Table 7.2: Planning statistics collected from the PhysicsDribble behavior. Sample size: $N = 861$ ($\approx 14s$). Planning parameters: $\Delta t = 1/60$s, $\mu = 0.1$, anytime planning time limit: 10ms, replan interval: 60Hz.

Table 7.2 shows planning statistics collected while running the PhysicsDribble behavior. Note, that the maximum planning time for the BK-BGT-Anytime planner was set 10ms (60% of a frame period), after which the planner is interrupted, and the best result is executed. A μ-value of 0.1 was chosen to allow sufficient breadth coverage, even in extremely small trees. As expected, both tree size and solution length are fairly small in terms of node count, due to the short planning time available. However, as we have shown in Table 7.1, this coverage still seems sufficient to perform the dribbling task well. The reason for this is that it does not take many simulation steps to predict significant implications of the robot's actions on the outcome of ball manipulation, mostly because the robot is already closely interacting with the ball during the initial state of each planner invocation.

Besides testing the PhysicsDribble behavior in a controlled lab environment, we also field-

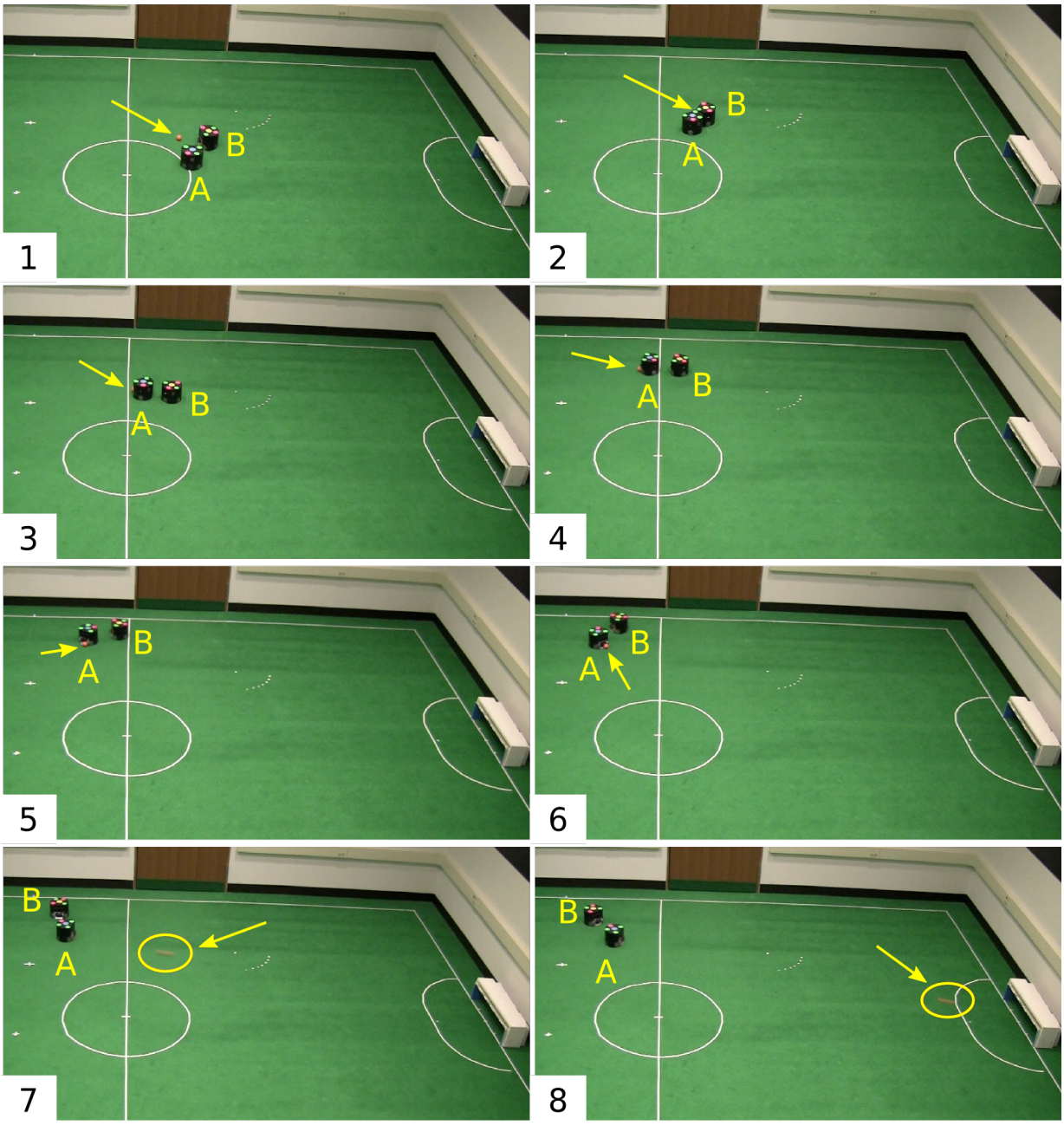


Figure 7.7: Frames from an example video sequence demonstrating the PhysicsDribble behavior. The controlled robot body (A) uses physics-based planning with the PhysicsDribble behavior to out-dribble and score a goal against the opponent body (B) that runs a traditional reactive behavior. The ball location is indicated with an arrow.

tested it in a real-world robot soccer competition environment at RoboCup 2009 in Graz, Austria. During the competition, there were several instances where the PhysicsDribble behavior was able to successfully control the robot and keep ball control in situations where the traditional Tactics would likely have struggled and possibly given up ball possession to the opposing team.

Figure 7.8 shows one particularly good example from RoboCup 2009 where the application of the PhysicsDribble behavior directly led to a goal against the opponent team. After obtaining possession of the ball, the physics dribble behavior leads the robot away from the opponents (images 1-2). Then, due to the field boundary evaluation metric `EvBoundary`, the planner changes the direction of the robot back toward the center of the field (images 3-4), consistently maintaining perfect ball control. Finally, as an open shot opportunity emerges, the PhysicsDribble behavior biases the robot's motion towards the goal, leading to a successful goal-shot against the other team (images 5-6).

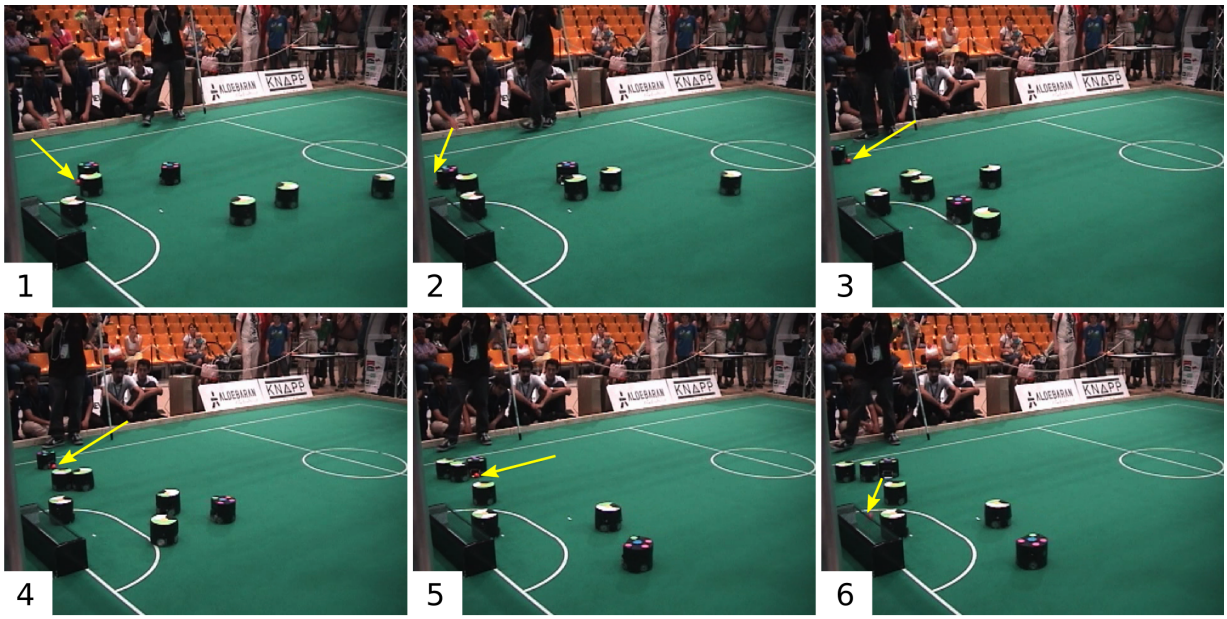


Figure 7.8: A sequence of frames from a Small Size League game at RoboCup 2009. A robot from the CMDragons team uses the PhysicsDribble behavior to out-dribble the opponent team and eventually scores a goal. The ball location is indicated with an arrow.

The above experimental results strongly emphasize the new advantages that physics-based planning provides when compared to traditional reactive control techniques in the robot soccer domain. So far, in the RoboCup Small Size League, teams have strongly depended on the presence of direct shot lines toward the goal or toward a teammate. In the case of shot-blockage by an opponent, most current teams resort to a waiting strategy, with

a small number of teams at least attempting to turn in place with the ball. As can be observed in various game recordings from the past several years, these types of reactive behaviors often lead to situations where both teams get "stuck" attempting to steal the ball back and forth from each other in place, without neither team taking the initiative to dribble the ball and actually make progress in the game. Our novel physics-based planning approaches apply well in such challenging dribbling situations, because the planner is able to autonomously generate control solutions that are tactically sound and that maintain dynamic control of the ball, thus providing the robot with new options in situations that were considered too difficult to be solved via reactive control.

Although it is difficult to isolate and precisely measure the positive effects of physics-based dribbling in real-world RoboCup matches, our first application in RoboCup 2009 has been positive, allowing our robots to more persistently remain in control of the ball in dribbling situations than in previous years. The lab-based experimental results (Table 7.1) clearly support this claim.

7.4 Planning in Predictable Dynamic Domains

The robot soccer domain presents an ideal application for anytime planning, due to its fast paced nature and due to the presence of unpredictable foreign-controlled bodies, requiring frequent replanning. However, there exist some dynamic domains where a single-shot planning approach with longer term planning and execution is more viable, even in the presence of foreign-controlled bodies. In this section, we present a particular example of such a problem, namely robot minigolf. We show how a single-shot planning approach in combination with accurate sensing and prediction can lead to successful executions in robot minigolf problems containing moving obstacles.

7.4.1 A Robot Minigolf Problem

We implement robot minigolf using the CMDragons robot system. In our implementation, arbitrary minigolf courses can be constructed on the CMDragons robot soccer field by freely choosing the position of the hole, the ball, and multiple obstacles. The shapes and other non-mutable parameters of all bodies in the domain (including the hole and obstacles) are manually modeled. The mutable states of all bodies are detected by the CMDragons vision system: each body is supplied with a unique color pattern that is used for identification and localization. Changes to the course's layout are observed in real time.

The robot minigolf domain allows the creation of several different types of planning prob-

lems. In the following chapter we focus extensively on solving minigolf problems requiring collisions, whereas in this section we focus on the problem of planning and execution in the presence of moving obstacles.

In particular, we introduce a rotating bar obstacle as an example for a foreign-controlled, but predictable body. Similar to a rotating "windmill" found in some of the fancier real-world minigolf courses, the rotating bar obstacle in robot minigolf is actuated at a constant angular velocity and placed somewhere in the direct path between the ball's initial position and the hole, thus requiring accurate timing by the robot to putt the ball into the hole without hitting the obstacle.

7.4.2 Challenges and Approach

The dynamic minigolf course with rotating obstacle poses a challenging problem, both from a planning and execution perspective. First of all, the planner must be able to correctly observe and predict the motion of the foreign-controlled rotating bar body. To obtain an accurate measurement of the bar's angular velocity, our approach employs a mean window filter that accumulates measurements from the most recent 60 frames (1 second) and that then provides its filtered measurement to a deterministic Tactic that will extrapolate the foreign-controlled rotating bar body's motions during planning, using this measured velocity.

Another challenge is that the planner will need to plan far enough into the future in order to determine whether its selected actions will result in a valid goal state or not. Such a single-shot search will require significantly more time than our single-frame anytime planning approach in the robot soccer domain. During planning, the rotating obstacle will continue to move, therefore raising the question how the planner's solution will still be applicable to the state of the world at execution time.

Luckily, with a rotating obstacle, the state of the world changes in a repeated infinite sequence. Within such a repeated environment, the single-shot planner can continue to plan for an arbitrarily long amount of time until it delivers its solution. Execution of this solution can then be postponed until the initial state of the solution sequence x_{init} matches the current observed state of the world as closely as possible (i.e., the rotating bar reaches the same position again that it had in the state x_{init}). Because we employ a single-shot approach, and because there is no replanning, the execution must be performed in a state-driven closed-loop fashion (see Section 7.2.3).

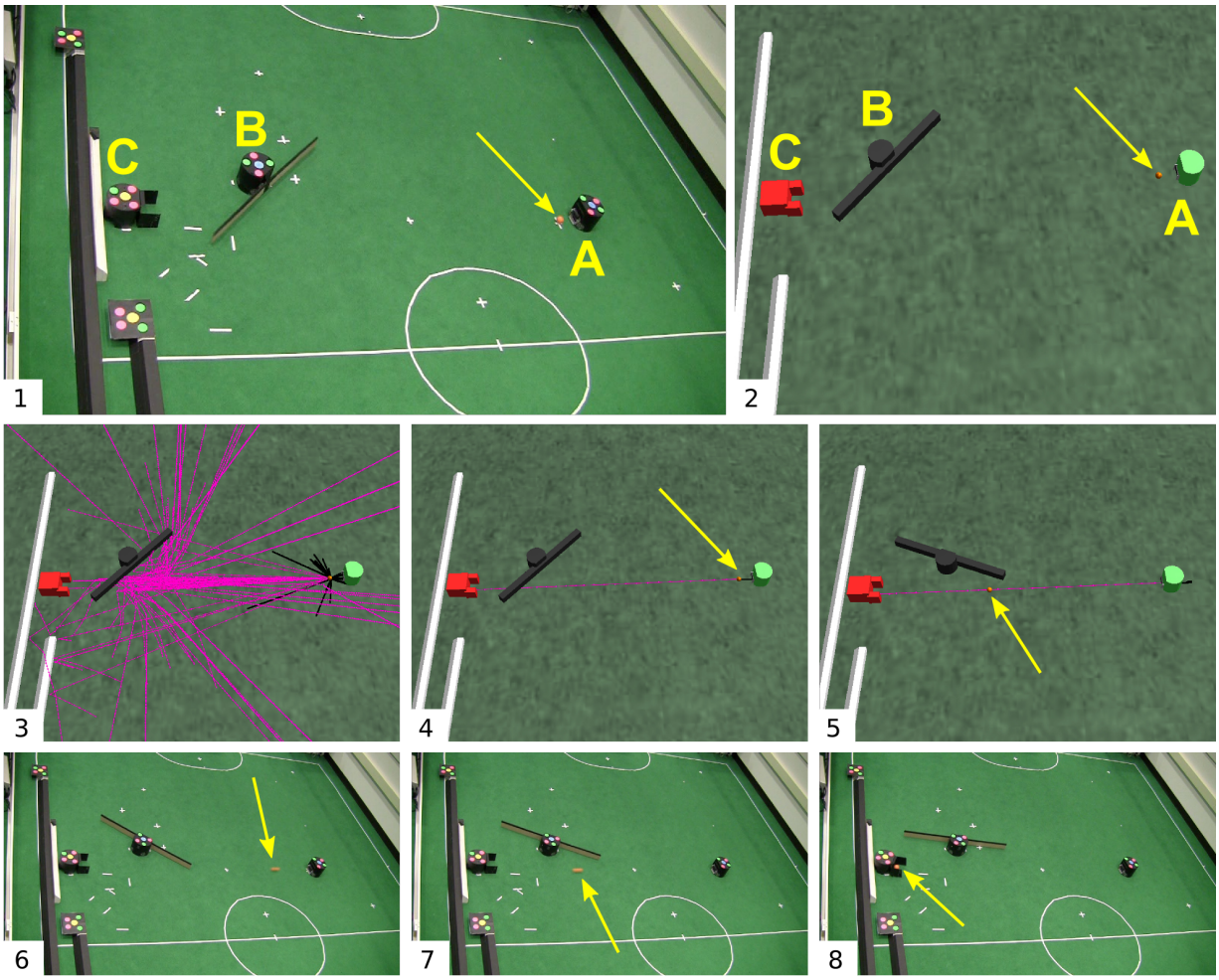


Figure 7.9: Images from the robot minigolf domain showing real world footage (images 1, 6-8) and the planner's internal representation (images 2-5). The controlled robot (A) has the objective to putt the ball (arrow) into the hole (C) without touching the rotating obstacle (B). Images 1 and 2 show the initial state, image 3 shows the planner's search tree, with pink nodes representing the ball's predicted position. Images 4-8 show a valid solution.

7.4.3 Results

We used the BK-BGT planner with the same controlled body Tactics model as used in the simulated minigolf domain (see Section 6.1.2). Image 1 in Figure 7.9 shows the general experimental setup. We performed 4 different experimental configurations, varying the distance between the ball and the hole between 1.4m and 2.4m, and varying the angular velocity of the rotating bar between a 5s and a 3s full rotation period. We ran 10 trials in each configuration, resulting in 40 trials total. Figure 7.9 shows an example result sequence in the 2.4m, 3s setup. Table 7.3 shows the execution success rate of our planner (trials resulting with the ball in the hole). Solely for comparison purposes, we have also provided the mean success rate of 3 human minigolf players in the same experimental setup.

Configuration		**Success Rate**	
Period	Distance	Human	Planner
5s	1.4m	70%	90%
5s	2.4m	30%	80%
3s	1.4m	67%	70%
3s	2.4m	30%	50%
	Mean	49%	73%

Table 7.3: Hole-in-one success rate with rotating obstacles (BK-BGT, $\mu = 10$).

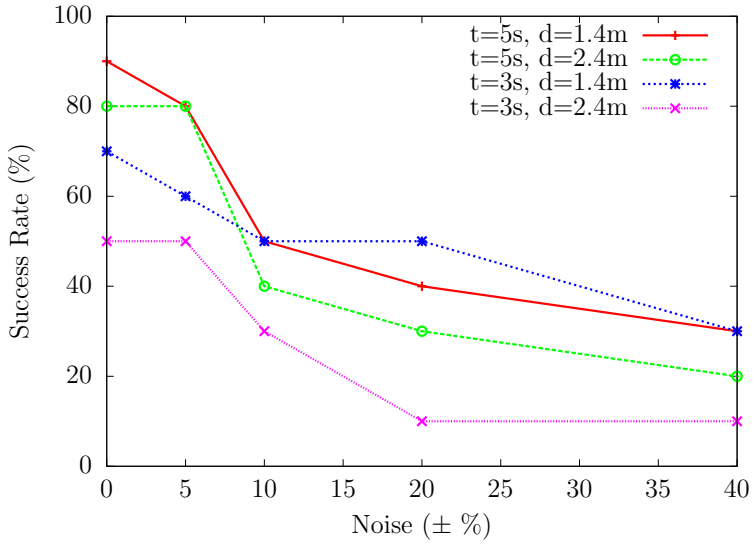


Figure 7.10: Impact of observation noise on execution success rate in the robot minigolf domain.

Finally, to investigate how important timing and prediction quality are in the robot minigolf domain, we have performed additional trials that analyze the impact of observation accuracy on execution performance. Figure 7.10 shows the execution success rate for the same 4 individual experimental setups as previously shown, but under varying levels of artificially generated noise applied to the observed velocity of the rotating obstacle that was used by the planner to make its predictions. Across all setups, increasing the observation noise, even only by a 10% margin, has a significantly adverse effects on overall execution success rate. These results further underline how important accurate planner predictions are in dynamic environments, and also serve as additional evidence to the great accuracy of the CMDragons observation and execution systems.

7.5 Chapter Summary

In this chapter, we have shown how to apply our Skills and Tactics-driven, physics-based planning approaches to real-world dynamic robot domains. We have integrated our planner into the CMDragons multi-robot system and have discussed the trade-offs between single-shot and anytime planning, and between state-driven and action-driven execution. We have introduced the BK-BGT-Anytime algorithm, a concrete anytime extension to BK-BGT. Furthermore, we have presented the concrete application of anytime planning in robot soccer with the PhysicsDribble behavior, and have shown (under lab-conditions and at a real RoboCup competition) that such a physics-based planning approach is able to outperform traditional reactive approaches. Finally, we have presented a particular problem in the robot minigolf domain as an example application for single-shot planning in predictable dynamic environments.

In the following chapter, we focus on the problem of maximizing the planner's prediction accuracy by optimizing the parameters of its physical model. We focus especially on the robot minigolf domain, introducing problems that require purposeful collisions between passive objects, making a particularly interesting challenge for our planning approaches.

Chapter 8

Parameter Optimization of Physics-Based Models

In the previous chapters, we have shown how our physics-based motion planning algorithms rely on a rich domain model to predict the physical interactions of the real world. As we have introduced in Chapter 2, the model's behavior depends on the values of its various parameters, describing the physical properties for each of the domain's bodies, including their shape, mass, coefficients of friction, restitution, and damping. Achieving good planning accuracy requires that the relevant parameters of all bodies in the domain are accurately chosen.

Finding the actual values for all the model parameters, however, is difficult. Although a few special parameters can typically be measured accurately (i.e., a body's shape and mass), the remaining model parameters cannot be determined so directly. A significant challenge is the fact that most of the model parameters are not fully independent with regards to the resulting behavior of the model. For example, the deceleration behavior of a rolling ball is determined by a combination of the friction, restitution, and damping parameters of both the ball and the surface that it is rolling on. Tweaking each of the parameters individually is therefore likely to lead to globally non-optimal configurations. Consequently, manual approaches to parameter configuration typically consist of many iterations of labor-intensive trial-and-error, relying on a lot of guess-work to determine a good set of parameter values. To no surprise, the resulting models tend to be non-optimal and ultimately lead to reduced success rates during real-world execution.

To alleviate the problem of model parameter configuration, we present an automated parameter optimization approach that aims to find a globally optimal set of model parameters for a physics-based planner. More specifically, our algorithm searches for the set of pa-

rameter values that minimizes the error between the simulated model and the real-world. To compute the model error, we introduce an evaluation function that compares a set of recorded ground-truth executions with the outcome of physics-based simulations that are generated on-the-fly during the model parameter optimization process.

To test our approach, we introduce a new set of problems in the robot minigolf domain that require the controlled robot to putt a golf ball into a hole by purposefully bouncing it from wooden bars. The initial positions of the hole, bars, and golf ball can be freely configured to build novel and challenging course configurations. We show that our physics-based planner is able to successfully find solutions in this domain and that its execution success rate can be improved significantly by employing our parameter optimization approach.

8.1 Approach Description

The goal of our approach is that the planner's transition function e is able to simulate the dynamics of the real physical world as closely as possible. To do so, we need to find the set of optimal values for all of the rigid body parameters in $\bar{r}$ for all rigid bodies in the domain.

We denote the set of parameters to be optimized as $\Phi = \{\phi_1, \phi_2, \dots\}$. For some of the parameters in $\bar{r}$, accurate values can be determined by simply measuring the real body that is being modeled (e.g., ϕ_{Mass}, ϕ_{Shape}, and even ϕ_{MassC}). These parameters can therefore be excluded from Φ.

In order to run a parameter optimization algorithm, we first need to define an error function that can quantitatively evaluate the model accuracy of the parameters in Φ. To evaluate model accuracy, the algorithm requires a fixed set of ground truth observations $\{\mathsf{ObsReal}_1, \dots, \mathsf{ObsReal}_m\}$. Each ground truth observation consists of a sequence of states pre-recorded from a real-world execution, interleaved with the actions that the robot executed between these states, that is: $\mathsf{ObsReal}_i = \langle x_1, a_1, x_2, a_2, \dots, a_{q_i-1}, x_{q_i} \rangle$ where q_i is the number of states in the solution sequence $\mathsf{ObsReal}_i$.

Given a set of ground truth observations, we can now define the error function $\texttt{Eval}(\Phi)$ that computes the overall error between simulated executions (using the current model parameters in Φ) and the recorded ground-truth execution sequences. Algorithm 13 shows how $\texttt{Eval}(\Phi)$ computes its result. First, the current parameter values in Φ are applied to the physics engine's current domain model d. Then, for each ground truth observation, the algorithm simulates an execution, starting from the same initial state and using the same sequence of actions as the ground truth observation. The result is a

Algorithm 13: Eval(Φ)

$d \leftarrow$ `UpdateDomainModel(`d`,`Φ`)`;
foreach $\mathsf{ObsReal}_j$ in $\{\mathsf{ObsReal}_1,\ldots,\mathsf{ObsReal}_m\}$ **do**
  $\langle x_1, a_1, x_2, a_2, \ldots, a_{q-1}, x_q\rangle \leftarrow \mathsf{ObsReal}_j$;
  $x'_1 \leftarrow x_1$;
  for $i \leftarrow 1$ **to** $q-1$ **do**
    $x'_{i+1} \leftarrow x'_i$;
    $x'_{i+1}.\langle \hat{r}_1, \ldots, \hat{r}_n\rangle \leftarrow e(x'_i.\langle \hat{r}_1, \ldots, \hat{r}_n\rangle, a_i, d, \Delta t)$;
    $x'_{i+1}.t \leftarrow x'_i.t + \Delta t$;
  $\mathsf{ObsSim} \leftarrow \langle x'_1, \ldots, x'_q\rangle$;
  $\mathsf{err}_j \leftarrow$ `ComputeError(`$\mathsf{ObsReal}_j, \mathsf{ObsSim}$`)`;
return $\left(\sum_{j=1}^{m} \mathsf{err}_j\right)/m$;

simulated sequence of states $(x'_1, \ldots, x'_q)$. This sequence (ObsSim) is then quantitatively compared with its corresponding ground truth ($\mathsf{ObsReal}$) using the individual error function `ComputeError`. Finally, `Eval`(Φ) returns the mean over all the individual errors. We show concrete examples of ground truth data and the corresponding individual error functions in the following section.

Using the set of parameters Φ and the evaluation function `Eval`(Φ), we then employ a function minimization algorithm that attempts to find values for Φ that minimize the output of `Eval`(Φ). We choose the Nelder-Mead Simplex nonlinear optimization method [49, 71] because it has the advantage of not requiring explicit knowledge of the error function's gradient, while at the same time being faster than evolutionary approaches [3].

8.2 Application in Robot Minigolf

We use robot minigolf as a concrete testbed for our parameter optimization approach. The details on the general robot minigolf domain and the implementation on the CMDragons multi-robot system are explained in Chapter 7. In this chapter we focus our tests on problems in the robot minigolf domain that are particularly affected by the accuracy of the physics parameters. Figure 8.1 shows a typical example of a robot minigolf problem that requires an accurate predictive model of multi-body interactions. The robot (A) has the objective to putt the ball from some starting location into the U-shaped hole (B), with a single shot. To do so, it must bounce the ball off one or more wooden blocks (C).

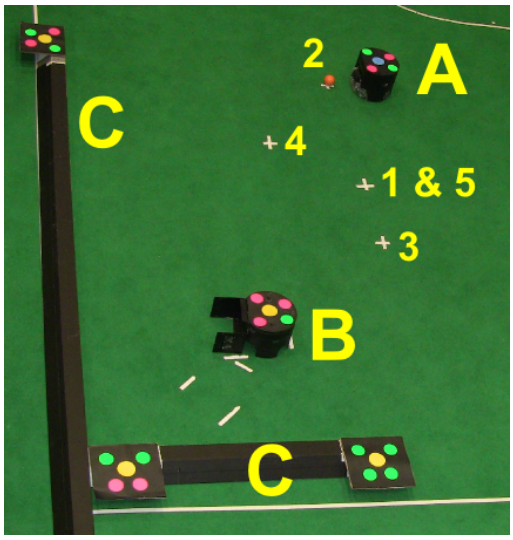


Figure 8.1: The robot minigolf domain.

8.2.1 Parameters

We choose to optimize all parameters that actively influence how the ball moves in the robot minigolf domain. The mass and dimensions of all the bodies were manually determined. For the ball body itself, the parameters to be optimized are ϕ_{FricS}, ϕ_{FricD}, ϕ_{Rest}, ϕ_{DampL}, and ϕ_{DampA}. For the wooden bars and the carpet, we can leave out the damping parameters (because the objects do not move), leaving ϕ_{FricS}, ϕ_{FricD}, ϕ_{Rest}. Note, however, that ϕ_{FricS} and ϕ_{FricD} are extremely similar for many materials. Furthermore, as the ball is constantly in motion, the static friction parameters become mostly irrelevant for accurate execution. To exploit this fact, we let $\phi_{\text{FricD}} = \phi_{\text{FricS}}$ in this domain by treating it as a single parameter, leaving us with a total number of 8 distinct parameters in the set Φ.

8.2.2 Ground Truth Data

It is important to carefully select ground truth data that can be used to best evaluate the accuracy of the model parameters that are being optimized. For the minigolf domain, we devised two different sets of ground truth data that, when combined, fully reflect effects of all the parameters in Φ .

The first set of ground truth data involves interactions between the ball and the carpet. The set consists of the robot putting the ball across the carpet at varying speeds, without any wooden bars being present. To cover the full action space, the robot putted the ball at

12 different initial velocities, ranging from 2m/s to 7.5m/s at 0.5m/s increments. To reduce observation noise, we repeated this sequence 3 times, resulting in 36 sequences total.

The second set of ground truth data involves interactions between the ball, the carpet, and the wooden bars. The set consists of the robot putting the ball against a bar at varying incoming angles and speeds. The angle was varied in 11.25° increments (ranging from 11.25° to 78.75°). At each angle, the putt was repeated at 10 different velocities ranging from 3m/s to 7.5m/s with 0.5m/s increments, thus generating 70 sequences in total.

8.2.3 Error Functions

We can now define the error functions that compare a simulated sequence to a ground truth sequence. Concretely, we provide two specific implementations of the `ComputeError` function as needed by the `Eval`(Φ) function that we showed in Algorithm 13.

Algorithm 14: ComputeVelError(ObsReal,ObsSim)

$\langle x_1, a_1, x_2, a_2, \ldots, a_{q-1}, x_q \rangle \leftarrow$ ObsReal;
$\langle x'_1, \ldots, x'_q \rangle \leftarrow$ ObsSim;
for $i \leftarrow 1$ **to** q **do**
  $\langle \alpha, \beta, \gamma, \omega \rangle \leftarrow x_i.\hat{r}_{\text{ball}}$;
  $\langle \alpha', \beta', \gamma', \omega' \rangle \leftarrow x'_i.\hat{r}_{\text{ball}}$;
  $\text{err}_i \leftarrow ||\gamma| - |\gamma'||^2$
return `Normalize`$\left(\sqrt{\left(\sum_{i=1}^{q} \text{err}_i\right)/q}\right)$;

The error function `ComputeVelError` (see Algorithm 14) is used in conjunction with the first set of ground truth data (the ball rolling on a carpet). The algorithm effectively compares the ball's velocity profile between the simulated and ground truth observation and then returns a normalized root mean square error. The idea is that a perfect physics model would also perfectly predict the ball's deceleration as it rolls on the carpet, for any given initial velocity.

The error function `ComputeAngleError` (see Algorithm 15) is used in conjunction with the second set of ground truth data (the ball bouncing against the bar). This function observes the ball's post-collision behavior and computes the ratio between the ball's outgoing angle toward its incoming angle (each in respect to the wooden block's surface normal). This computation is performed on each post-collision frame, for both the simulation and ground truth, and each time the square difference between simulation and ground truth is computed. `AngDiff` denotes the shortest difference between two angles, and $\angle$ denotes the angle of a vector as it is down-projected to the 2D carpet surface using a global coordinate

Algorithm 15: ComputeAngleError(ObsReal,ObsSim)

$\langle x_1, a_1, x_2, a_2, \ldots, a_{q-1}, x_q \rangle \leftarrow$ ObsReal;
$\langle x'_1, \ldots, x'_q \rangle \leftarrow$ ObsSim;
$c \leftarrow$ `FindIndexOfCollision(ObsReal)`;
$k \leftarrow$ `FindIndexOfCollision(ObsSim)`;
$i \leftarrow c$; $j \leftarrow k$; $h \leftarrow 1$;
repeat
  $\langle \alpha, \beta, \gamma, \omega \rangle \leftarrow x_i.\hat{r}_{\text{ball}}$;
  $\langle \alpha', \beta', \gamma', \omega' \rangle \leftarrow x'_j.\hat{r}_{\text{ball}}$;
  if $i = c$ **then**
    `// Compute surface normal angle of obstacle at collision point:`
    normAngle $\leftarrow \angle((x_c.\hat{r}_{\text{wood}}.\beta)\ [1,0,0])$;
    normAngle′ $\leftarrow \angle((x'_k.\hat{r}_{\text{wood}}.\beta)\ [1,0,0])$;
    `// Compute incoming angle toward surface normal angle:`
    inAngle $\leftarrow$ `AngDiff`$(\angle((x_1.\hat{r}_{\text{ball}}.\alpha) - \alpha)$, normAngle$)$;
    inAngle′ $\leftarrow$ `AngDiff`$(\angle((x'_1.\hat{r}_{\text{ball}}.\alpha) - \alpha')$, normAngle′$)$;
    `// Store positions where collision occurred:`
    $\alpha_c \leftarrow \alpha$;
    $\alpha'_k \leftarrow \alpha'$;
  else
    $\text{err}_h \leftarrow \left(\frac{|\texttt{AngDiff}(\angle(\alpha - \alpha_c), \text{normAngle})|}{|\text{inAngle}|} - \frac{|\texttt{AngDiff}(\angle(\alpha' - \alpha'_k), \text{normAngle}')|}{|\text{inAngle}'|} \right)^2$;
    $h \leftarrow h + 1$;
  $i \leftarrow i + 1$; $j \leftarrow j + 1$;
until $((i \geq q)$ **or** $(j \geq q)$ **or** $(|\gamma| < \text{MinVel})$ **or** $(|\gamma'| < \text{MinVel}))$;
return `Normalize`$\left(\sqrt{\left(\sum_{i=1}^{h} \text{err}_i \right) / h} \right)$;

frame for the angle computation. normAngle is the angle of the wooden block's surface normal that is used for the collision, and inAngle is the ball's incoming angle relative to the surface normal's angle. Note, that because there is no guarantee that the simulated ball reached the bar at exactly the same point in time as in the ground truth version, the algorithm first needs to search for the state where the collision with the bar occurred. The function `FindIndexOfCollision` does so by parsing the sequence of states and detecting the first significant change in angle that happened at a sufficiently large velocity. The comparison stops once the ball (either simulated or ground) has fallen below a certain minimum velocity MinVel (0.1m/s for our purposes). Again, the function returns a normalization of the root mean square error computed from the individual errors of all post-collision frames.

An ideal physics model configuration should reduce the error of both `ComputeVelError`

and `ComputeAngleError`. Thus, we want to combine the two separate evaluators into a single one. We do so by redefining our main model evaluation function `Eval` to consist of the weighted root mean square error of two individual evaluator functions:

$$\texttt{Eval}(\Phi) = \sqrt{w_1 \ (\texttt{Eval}_1(\Phi))^2 + w_2 \ (\texttt{Eval}_2(\Phi))^2}.$$

Here, $\texttt{Eval}_1$ is exactly as shown in Algorithm 13, but using `ComputeVelError` (Algorithm 14) in conjunction with the first set of ground truth data. Similarly, $\texttt{Eval}_2$ is exactly as shown in Algorithm 13, but using `ComputeAngleError` (Algorithm 15) in conjunction with the second set of ground truth data. Note, that because both error functions normalize their results, we are able to use equal weights of 0.5 for our experiments, however other weights could be selected if one were to prefer one evaluator's quality over the other.

8.2.4 Experimental Evaluation

We applied the parameter optimization algorithm in the robot minigolf domain. Initial values for all parameters in Φ were determined using publicly available information about typical material coefficients where applicable and best-guessed otherwise. The parameter optimization algorithm ran for 50 iterations, taking approximately 27 minutes. However, a clear convergence was already visible after 30 iterations. Figure 8.2 shows the individual parameters over the course of optimization and Figure 8.3 shows the root mean square error as returned by $\texttt{Eval}(\Phi)$. Table 8.1 shows the values of the initial and final optimized parameters.

Object	**Parameter**	**Initial**	**Optimized**
Ball	$\phi_{\text{FricS}}, \phi_{\text{FricD}}$	0.3	0.103
	ϕ_{Rest}	0.78	0.835
	ϕ_{DampL}	0.1	0.145
	ϕ_{DampA}	0.1	0.191
Wood	$\phi_{\text{FricS}}, \phi_{\text{FricD}}$	0.3	0.336
	ϕ_{Rest}	0.5	0.570
Carpet	$\phi_{\text{FricS}}, \phi_{\text{FricD}}$	0.8	0.759
	ϕ_{Rest}	0.2	0.196

Table 8.1: Values of initial and optimized parameters.

To evaluate whether this parameter optimization led to a significant improvement in real-world execution accuracy, we devised five standardized test configurations in the minigolf domain. The first four configurations require the robot to perform a single bounce off a wooden bar into the hole, using different starting locations (1-4 in Figure 8.1). The

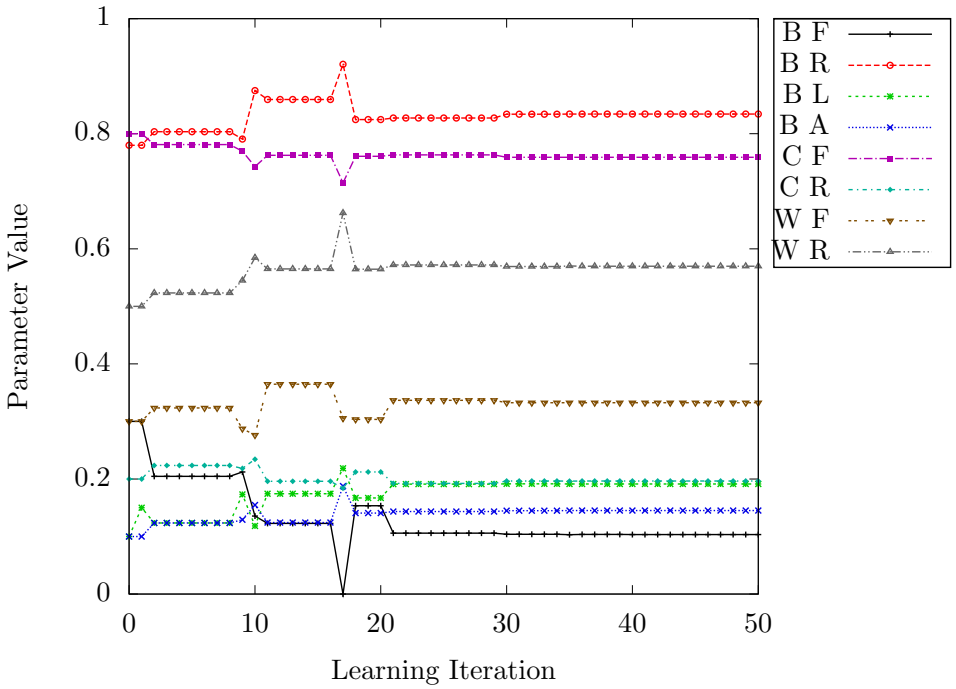


Figure 8.2: Parameter values over optimization iterations (B: Ball, C: Carpet, W: Wood, F: Friction, R: Restitution, L: Linear Damping, A: Angular Damping).

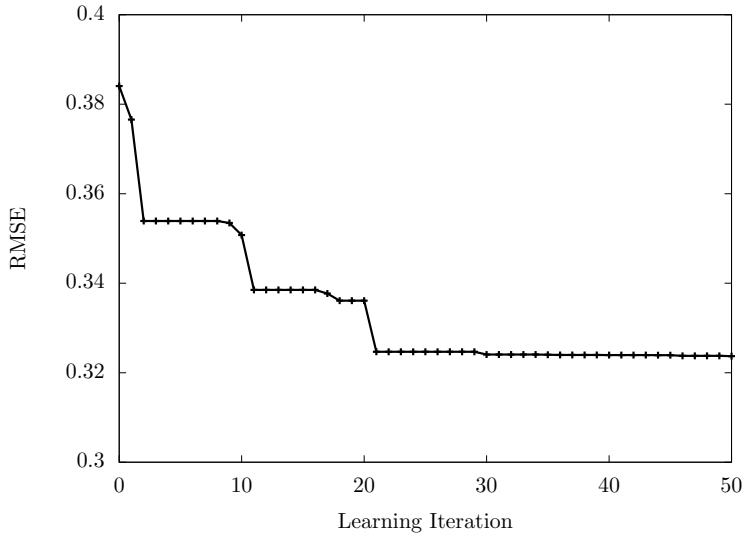


Figure 8.3: Root mean square error over optimization iterations.

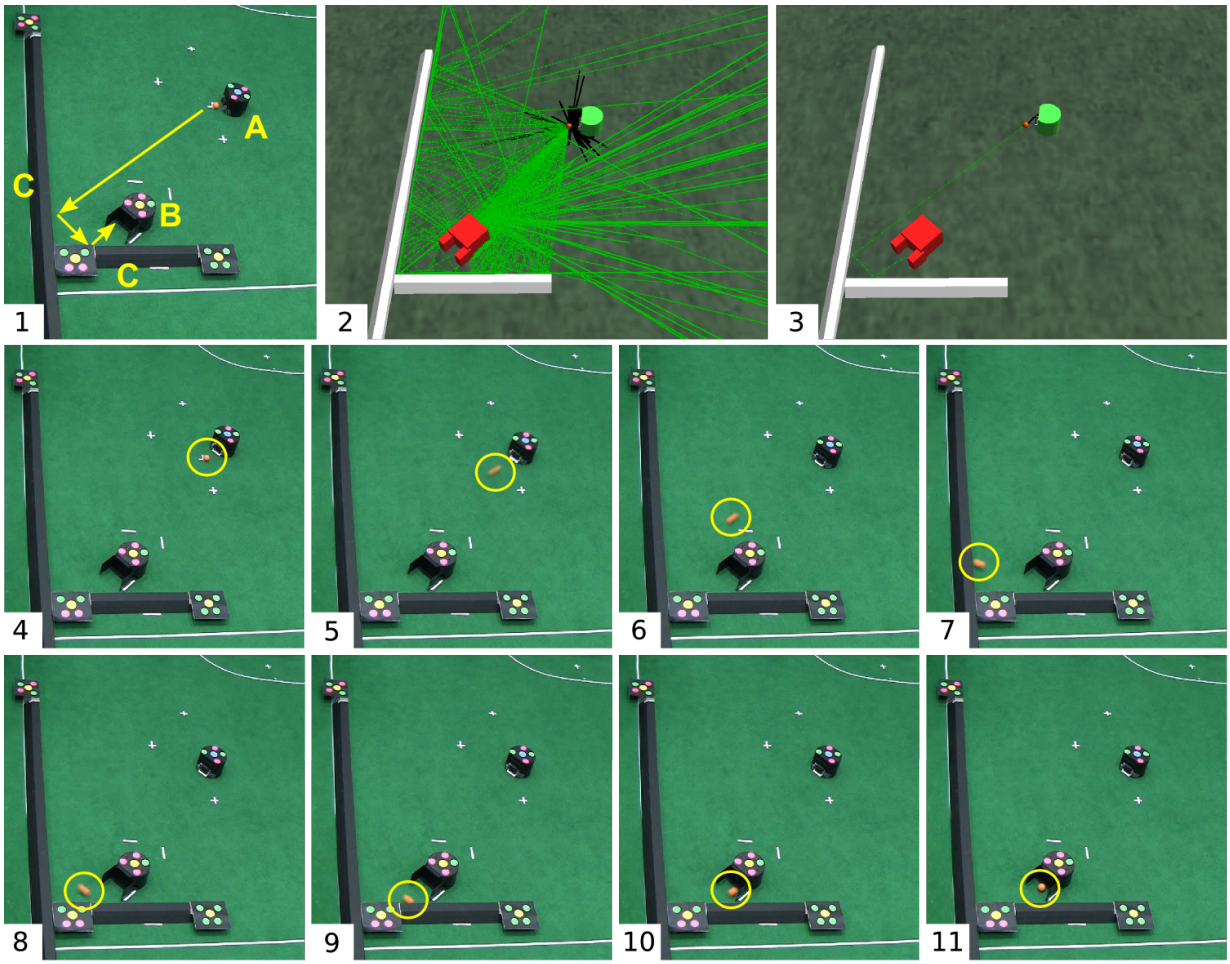


Figure 8.4: A "double bounce" example from the robot minigolf domain. Image 1 shows the initial state of the robot and ball (A), the hole (B), and the wooden obstacles (C). Image 2 shows the planner's internal search tree, with green nodes representing predicted ball positions. Image 3 shows the solution found by the planner. Images 4-11 show the execution of the solution: the robot putts the ball (yellow oval) against both wooden bars using a double bounce, delivering it into the hole.

fifth configuration (see Image 1 in Figure 8.4) requires the robot to perform a double bounce off two wooden bars to putt the ball into the hole with a single shot. We ran each configuration 10 times for both initial and optimized parameter values, thus performing 100 real-world trials in total. A success was counted when the robot achieved a hole-in-one, a failure otherwise. In either configuration, the physics-based planner found valid simulated solutions, typically within a minute. Images 4-11 in Figure 8.4 show a successful execution of the planner's solution in the double bounce problem configuration.

Config.	Success Rate			Improvement
	Human	Initial	Optimized	(Optimized-Initial)
1	43%	60%	80%	20 pp
2	13%	40%	60%	20 pp
3	47%	80%	100%	20 pp
4	33%	50%	90%	40 pp
5	40%	10%	50%	40 pp
Mean	35%	48%	76%	28 pp

Table 8.2: Success rates of initial vs. optimized parameters. The mean execution success rate of 3 untrained human subjects is also shown, for comparison purposes only.

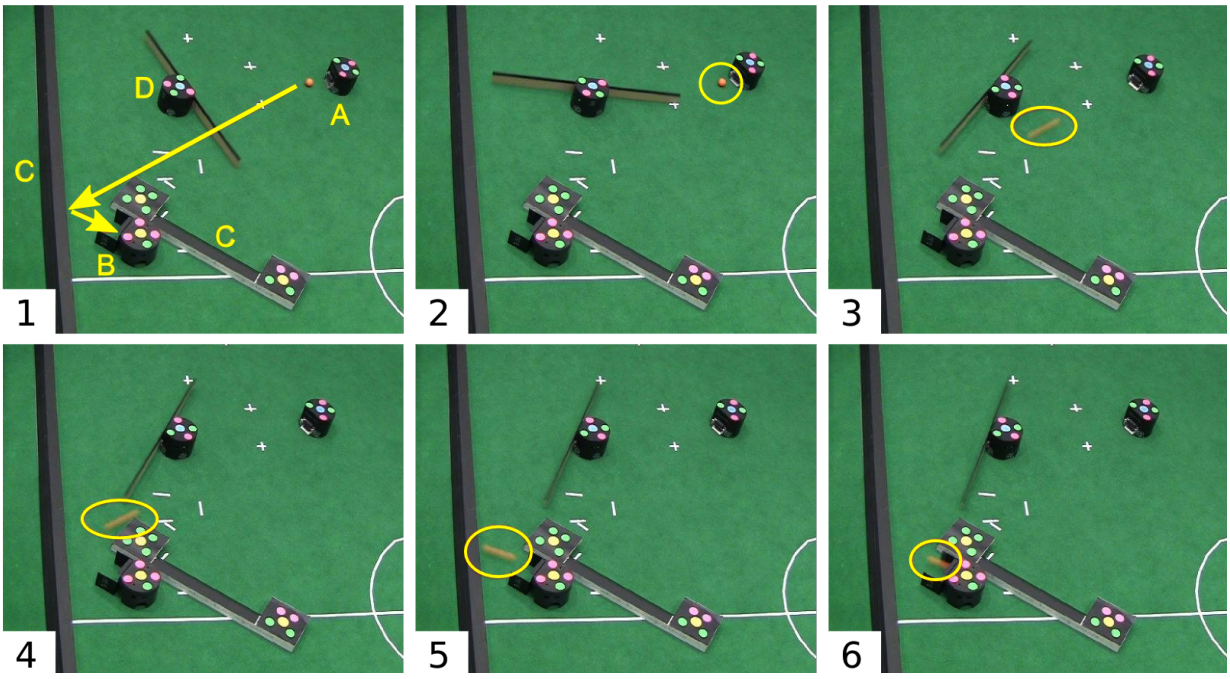


Figure 8.5: A robot minigolf example combining a dynamically rotating obstacle with a single bounce. Image 1 shows the initial state of the robot and ball (A), the hole (B), the wooden bars (C) and the rotating obstacle (D). Images 2-6 show the plan execution: the robot accurately times its approach to the ball with the motion of the rotating bar, allowing a successful shot of the ball against the wooden bar, and into the hole.

Table 8.2 shows the real world execution accuracy from all five minigolf test configurations obtained using both initial and optimized parameters. Furthermore, solely for the purpose of obtaining a rough subjective idea of the difficulty of the domain, the table also shows the mean accuracy values obtained from three untrained human test subjects who performed the same series of experiments (each taking 10 shot per configuration). The parameter optimization results clearly show an increase in the average execution success rate by 28 percentage points (from 48% to 76%) when comparing non-optimized and optimized parameter models. These improvements are statistically significant (p-value of 0.004 in *t-test*).

Given the optimized domain model, and given the planning techniques for predictable foreign-controlled bodies that were introduced in Chapter 7, our approach is able to solve many new and interesting robot minigolf courses. Figure 8.5 shows a successful execution of one particularly challenging problem, combining a dynamically rotating bar obstacle with a bounce-shot. We would like to emphasize that we believe this to be the world's first minigolf playing robot system that is able to autonomously solve never-before-seen course configurations involving collisions and moving obstacles by using physics-based planning techniques.

8.3 Chapter Summary

In this chapter we have presented an approach to optimize the model parameters of a physics-based planner. We have introduced a method to evaluate model accuracy by comparing generated simulations with pre-recorded ground truth data. We have grounded our approach in the robot minigolf domain. We have shown that our approach works successfully by significantly improving the robot's real-world execution accuracy.

Chapter 9

Variable Level-Of-Detail (VLOD) Planning

Motion planning in dynamic environments is particularly challenging when the domain contains uncertainty, and especially when there are foreign-controlled bodies that are not fully predictable. Accurate single-shot motion planning in such uncertain domains is a futile task: as soon as the robot starts executing its plan, the real world's state is likely to quickly diverge from the predictions that were made during planning. A well-known sensible solution to this problem is continuous replanning. When using replanning, the robot only executes a portion of a generated plan. It then re-observes the world's true state and plans a new solution, using the latest observations. This process is repeated as a *replanning loop* until the robot reaches its goal state.

A major problem of replanning is its computational cost. At each replanning iteration, the planner performs an intensive search for a complete solution that will bring the robot from its current state all the way to the final goal state. During this search, the motion planner will ensure that the resulting plan is dynamically sound and collision-free by employing sophisticated, computationally expensive physical models (i.e., the physics engine, as described in Chapter 2) to predict the robot's motions and its interactions with the predicted environment. Such an elaborate search might seem unnecessary, given the fact that the robot will only execute a small portion of the resulting plan, before discarding it and re-invoking the planner to start from scratch in the next replanning iteration. However, simply limiting the planner's search depth, an approach also known as *finite-horizon planning*, is dangerous because it can lead to a robot becoming stuck in a local search minimum as there are no guarantees that a partial plan will actually lead to the final goal state.

In this chapter, we introduce Variable Level-Of-Detail (VLOD) planning, a novel approach to reduce the computational overhead of replanning in physical robot environments. Unlike finite-horizon planning, VLOD planning maintains a full search to the goal state, and is therefore unaffected by local minima.

VLOD planning is based on the idea that a planner in a replanning environment should be able to speed up its search by relaxing the treatment of computationally intensive domain details that occur too far in the future to be relevant during near term execution and that are unlikely to be accurately predicted due to uncertainty anyhow. We present a binary LOD model that allows the planner to selectively ignore future interactions with bodies that are considered to be difficult to predict and that can be safely ignored in a replanning scenario. The time threshold that defines what is to be considered "far enough in the future to be ignored", is a controllable parameter, and can be adjusted based on the replanning interval, which in turn is typically chosen based on the amount of unpredictability in the domain (thus the term *Variable* LOD planning).

In the following sections we present our VLOD approach, and we show how to integrate it into our existing planning algorithm. We test our approach on two domains and present a detailed analysis of the results.

9.1 Algorithm Description

To be useful for execution in real-world domains containing uncertainty, a motion planning algorithm can be wrapped into a continuous, fixed-timestep replanning loop (see Algorithm 16). After observing the initial state of the world, the robot generates a plan using some motion planning algorithm (`Plan`). The robot then executes a fixed, pre-determined amount of this plan, before repeating the loop of re-observing the environment, updating the initial state, and generating a new plan. The replanning interval t_{replan} is set by the user, and generally depends on the expected domain uncertainty. Uncertain domains and robots with unreliable executions tend to require more frequent replanning as the true world state will more quickly diverge from the predicted planning solution while the robot is executing.

In this type of replanning environment, a planner will perform many computationally intensive searches for detailed solutions, only to have them be partially executed and then scrapped for the next replanning iteration. To alleviate this situation, we now introduce *Variable Level-Of-Detail* (VLOD) planning, that is able to find global planning solutions while ignoring execution-irrelevant domain details that lie far in the future. We introduce the *LOD-horizon* t_{LOD}. This horizon acts as a threshold in the time component t of our

Algorithm 16: ExecuteAndReplan

```
while true do
    x_init ← ObserveWorldState();
    solution ← Plan(x_init, ...);
    if solution ≠ Failed then
        i ← 1;
        repeat
            ⟨x, a⟩ ← solution[i];
            Execute(a);
            i ← i + 1;
        until x.t > t_replan or i > length(solution) ;
```

planning space X (see Chapter 2). The purpose of t_{LOD} is to control at what point the planner should begin to ignore certain domain details during its simulated state transitions. The value of t_{LOD} is a global parameter, to be set by the user. A reasonable guideline is that t_{LOD} should be greater than the replanning interval t_{replan} because there is the inevitable assumption that the plan will be executed up to length t_{replan} and as such should be planned with maximum detail for at least that length.

9.1.1 Definition of Detail and Collision Matrix

To apply Variable Level-Of-Detail planning, we need to clearly define what we mean by *details*. There are multiple possible approaches that come to mind to "relax" the planner's search beyond the LOD-horizon t_{LOD}. In this work, we use a binary notion of detail, allowing the planner to selectively ignore particular pairwise multi-body interactions.

A general core requirement for *details* is that their absence beyond the LOD-horizon should not make the planner deliver solutions that lead the controlled body into a local minima during the course of execution. In other words, we can regard interactions as *details* if they are solvable through a local finite horizon search (within the LOD-horizon) without requiring a global change of plans beyond the LOD-horizon. In our particular planning model, we rely on the body classification hierarchy that was introduced in Section 2.3. Using this model of body types, we define *detail* as follows:

Definition 9.1.1 (Detail)**.** Interactions between *controlled* bodies and other *manipulatable* bodies are *details*, whereas any other interactions are *essential*.

The idea behind this particular definition of detail is that the avoidance and/or manipulation of moving or manipulatable bodies can normally be considered a locally solvable

problem, whereas global navigation, such as finding the path through a maze of static wall bodies, requires full-depth planning to successfully reach the goal state without ending up in local minima. Another line of reasoning is that foreign-controlled bodies are not accurately predictable in the long term and as such qualify as details that are only relevant for short term planning. Because foreign-controlled bodies can interact (e.g., push) any other manipulatable (i.e., passive) body, we consider all manipulatable bodies to fall into the unpredictable *detail* category.

Limitations

Of course one could imagine special cases of domains where even interactions with manipulatable bodies have implications on the global topology of the plan that go beyond the LOD search horizon t_{LOD}. For example, one could imagine a long corridor that, at some point, contains a very dense accumulation of manipulatable bodies that effectively prevents passage for the controlled body. If these obstacle bodies are located beyond the LOD horizon, then the planner will happily ignore their existence and lead the robot down the corridor until the obstacle bodies fall within its LOD horizon. However, as the problem is not locally solvable, the planner will essentially "get stuck". For such domains, it might make sense to either increase the value of t_{LOD}, or – in extreme cases – manually reduce the selection of pairwise interactions that should be considered "details".

Collision Matrix

In order to perform VLOD planning, a planning algorithm needs to be able to selectively ignore *details* during its search. In particular, given our previous definition of detail, the planner should ignore interactions between the controlled body and any other manipulatable bodies, if such interactions occur beyond the replanning interval t_{LOD}. To do so, we introduce an extended version of the state transition function e (see Chapter 2), called e_{LOD}:

$$e_{\text{LOD}} : \langle \hat{r}_1, \ldots, \hat{r}_n, a, M, G, \bar{r}_1, \ldots, \bar{r}_n, \Delta t \rangle \rightarrow \langle \hat{r}'_1, \ldots, \hat{r}'_n, L \rangle.$$

The difference between the two functions is that e_{LOD} takes an additional parameter, the *collision matrix* M that allows the configuration of the physics engine's collision simulation behavior. More specifically, M is a symmetric matrix of size $n \times n$ (where n is the number of rigid bodies in the domain, as presented in Chapter 2) and defines whether pairwise collisions between any two rigid bodies r_i and r_j should be resolved or ignored. A value of 1 for a matrix entry m_{ij} (and therefore also m_{ji}) implies that a collision should be resolved as if the two bodies were rigid, i.e., the bodies should not penetrate one another. A value of 0 implies that collisions should be ignored by treating the two bodies as non-rigid

with respect to each other, i.e., the bodies should pass through one another. Ignoring a collision also means that it will not be reported as part of the simulation function's output collision list L. By default, M is filled with ones, except for its main-diagonal that is always zero-filled, because a body is unable to collide with itself.

9.1.2 VLOD Planning Algorithm

Algorithm 17: BK-RRT-VLOD

Input: Domain: d, Initial state: x_{init}, set of goal states: X_{goal}, default collision matrix: M, RRT sampling space: Y, timestep: Δt, validation function: `Validate`, max iterations: z.

```
tree ← NewEmptyTree();
tree.AddNode(x_init);
busy ← false;
for iter ← 1 to z do
    if busy = true then
        x ← x';
    else
        ⟨x, y⟩ ← SelectNodeRRT(tree, Y); // See Section 5.3
    M ← SetupCollisionMatrix(x, M);
    ⟨x', L⟩ ← TacticsDrivenPropagateVLOD(x, y, Δt, d, M);
    busy ← x'.b_∧;
    if Validate(x', L) then
        tree.AddNode(x');
        tree.AddEdge(x, x', a);
        if x' ∈ X_goal then
            return TraceBack(x', tree);
    else
        if busy = true then
            RollBack(x, tree); // See Section 5.3
        busy ← false;
return Failed;
```

Algorithm 17 shows BK-RRT-VLOD, an extended version of the BK-RRT planning algorithm (see Section 5.3), adding support for Variable Level-Of-Detail planning. The core modification to the standard BK-RRT algorithm is that BK-RRT-VLOD reconfigures the collision matrix M to set the currently appropriate Level-Of-Detail, before every state transition. This reconfiguration is performed by the `SetupCollisionMatrix` function call,

taking as input the current collision matrix and the source state x, and producing as output the new updated collision matrix. The resulting collision matrix is then passed to the `TacticsDrivenPropagateVLOD` which is exactly identical to BK-RRT's Tactics-based state propagation function `TacticsDrivenPropagate` (see Section 5.3), except that the VLOD version uses the state transition function e_{LOD} that makes use of the collision matrix M, instead of just using the plain simulation function e that always performs the default collision behavior.

The `SetupCollisionMatrix` function is shown in Algorithm 18. The configuration performed by the function depends on the time index of the current source state x. If the current source state x has a time index less than t_{LOD}, then all collisions are fully simulated and resolved. However, if $x.t$ lies beyond t_{LOD}, then the collision is set to be ignored if it involves a pair of bodies, with one body being a member in the set of controlled bodies $B_{\text{Controlled}}$ and the other being a member in the set of manipulatable bodies B_{Manip} (see Section 2.3 for the definitions of these sets). All other pairwise collisions are treated normally.

Algorithm 18: `SetupCollisionMatrix`

Input: State: x, Collision Matrix: M.
`// Let` m_{ij} `denote the element at the` i`-th row and` j`-th column of` M`.`
for $i \leftarrow 1$ **to** n **do**
  for $j \leftarrow i$ **to** n **do**
    if $i = j$ **then**
      $m_{ij} = 0$;
    else
      if ($x.t > t_{\text{LOD}}$ **and**
      (($r_i \in B_{\text{Controlled}}$ **and** $r_j \in B_{\text{Manip}}$) **or** ($r_j \in B_{\text{Controlled}}$ **and** $r_i \in B_{\text{Manip}}$)))
      then
        $m_{ij} \leftarrow 0$;
        $m_{ji} \leftarrow 0$;
      else
        $m_{ij} \leftarrow 1$;
        $m_{ji} \leftarrow 1$;
return M;

Figure 9.1 shows an illustrative example of VLOD planning. The controlled robot body (R) has to navigate from its current state to the goal. The domain contains four poorly predictable foreign-controlled moving bodies (labeled 1-4) and a static obstacle. Assuming a LOD horizon $t_{\text{LOD}} = 2$ seconds, the first body is treated with full detail during a predicted collision that occurred before $t = 2$, thus requiring planning a path around the body. The

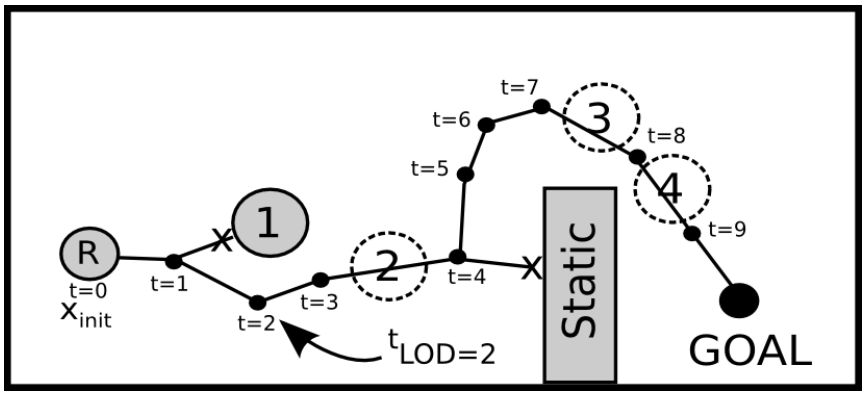


Figure 9.1: An illustration of VLOD planning.

static obstacle is fully predictable and relevant to the global path topology, thus it is always treated as an obstacle (even when $t > t_{\text{LOD}}$). The other moving bodies (2-4) only make contact with the search tree beyond the horizon t_{LOD}, thus they are ignored in this planning iteration.

It should be noted, that all of the modifications that were performed to transform the BK-RRT algorithm into its BK-RRT-VLOD version can also be seamlessly applied to transform BK-BGT intro its equivalent BK-BGT-VLOD version. In fact, the VLOD approach itself is agnostic to the planner's node and action selection methods, since VLOD only affects the state transition component of the planning algorithm.

9.2 Experimental Evaluation

We tested Variable Level-Of-Detail planning with the BK-RRT-VLOD algorithm using a simulation framework that is able to model robot execution and domain uncertainty. For the scope of these experiments, the planner's Tactics model was configured to the traditional dynamic RRT navigation (as described in Section 6.1.1). The simulation timestep Δt in all experiments was 1/60 second.

We devised two challenging dynamic robot navigation domains to test the effects of VLOD planning. In each domain, the controlled robot body has to navigate from a starting state to a goal area while avoiding multiple oscillating obstacles. The planner in this case uses a simple deterministic Tactics model of where it expects the obstacles to move. We actively model the domain's execution uncertainty by controlling the amount of random divergence of the obstacle's actual motion paths from the planner's prediction model. Additionally, a small fixed amount of execution uncertainty is always present, due to the internal non-

determinism of the physics engine (see Section 2.2.2).

The controlled variables of our experiments are the LOD time horizon t_{LOD}, the replanning interval t_{replan}, and the domain uncertainty. Note that, although we are using a fixed replanning interval, the replanning loop is configured to prematurely abort an execution and trigger an immediate replan if a physical collision occurs between the controlled body and another manipulatable body during execution.

Figure 9.2: A search tree in the Hallway domain.

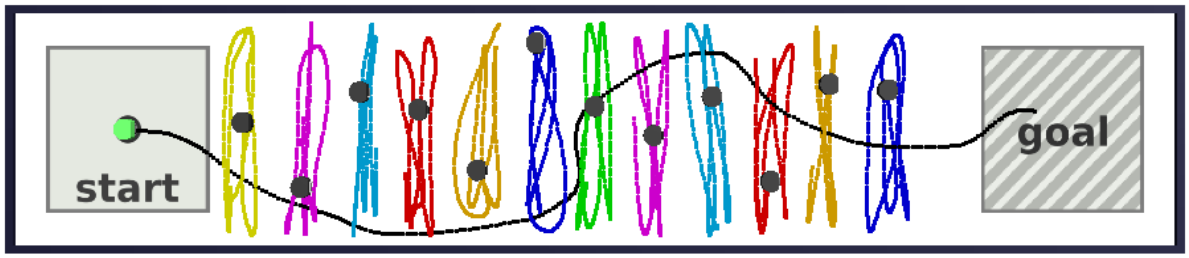


Figure 9.3: A solution trajectory in the Hallway domain, showing executions of obstacles under high uncertainty.

In the "Hallway" domain, the main challenge is to execute a safe trajectory through a dense field of 12 rapidly moving and not fully predictable obstacle robots. Figure 9.2 shows the search tree (black) as generated by the BK-RRT planner with full level of detail ($t_{\text{LOD}} = \infty$) during its first replan iteration. The planner's linearly predicted trajectories of the moving obstacles are indicated by the vertical paths. Figure 9.3 shows what individual obstacle trajectories actually look like during execution under maximum domain uncertainty (uncertainty value of 1.0).

The "Maze" domain (see Figure 9.4) is even more challenging from a navigation standpoint. In its layout, the domain contains several "horseshoe"-shaped walls, consequently requiring a deep search all the way to the goal state because a finite-horizon search would get the robot stuck in a local search minimum. Similar to the Hallway domain, the Maze domain contains several fast-moving foreign-controlled obstacles along the way, further increasing

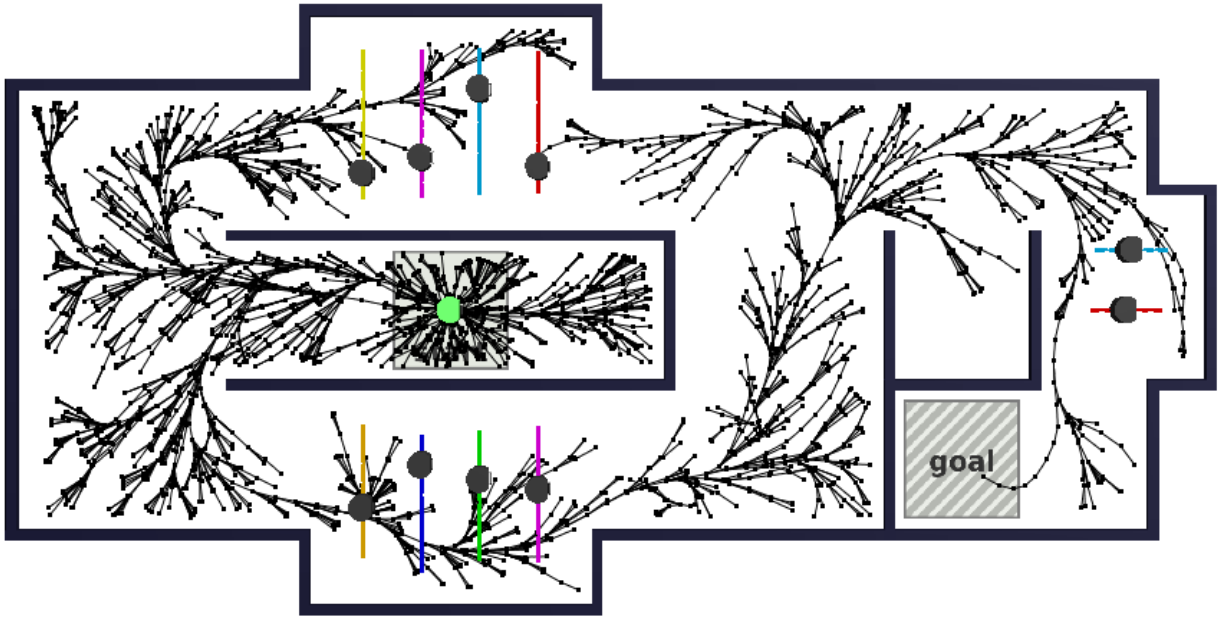


Figure 9.4: A search tree in the Maze domain.

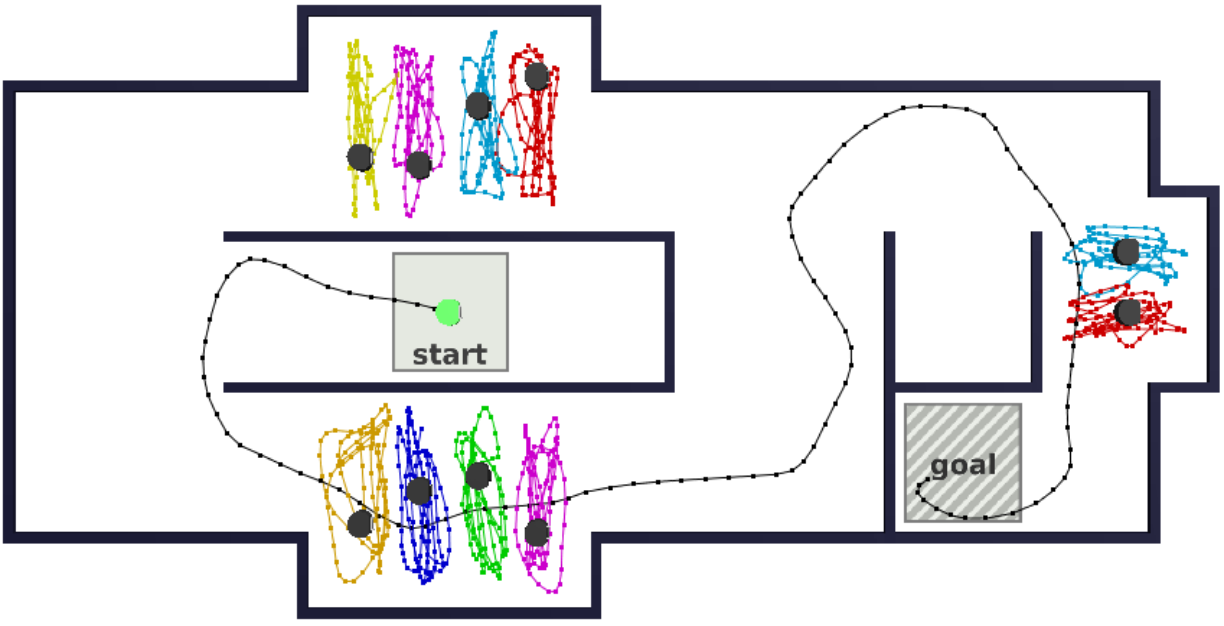


Figure 9.5: A solution trajectory in the Maze domain, showing the execution paths of obstacles under high uncertainty.

the difficulty of the planner's search. Again, the planner will use a linear prediction of the obstacles, as shown in Figure 9.4, whereas their actual motions during execution can be significantly different thanks to our uncertainty model (see Figure 9.5).

9.2.1 Performance Metric

When planning in robot navigation environments, the most relevant performance metrics are the rate of physical collisions occurring during execution (i.e., the number of replanning intervals that result in a collision during execution, divided by the total number of replanning intervals required), as well as the amount of total accumulated planning time required to get the robot from its initial state to the goal state. Combining these two variables, we can express the planner's overall performance using a single relative performance comparison metric, defined as

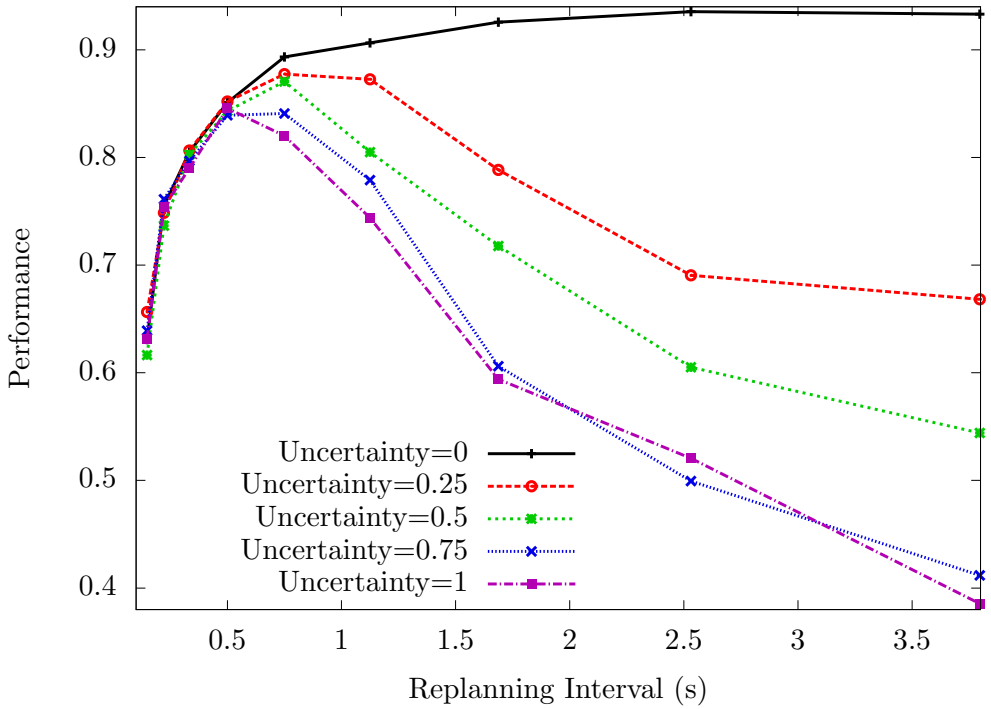
$$\text{performance} = (1 - \text{NormCollisions})\ (1 - \text{NormTime}),$$

where NormCollisions and NormTime are both values ranging from 0-1, normalized over an entire experiment. An ideal planning strategy would use a minimum amount of cumulative planning time and generate a minimum number of collisions, thus generating a maximum performance value.

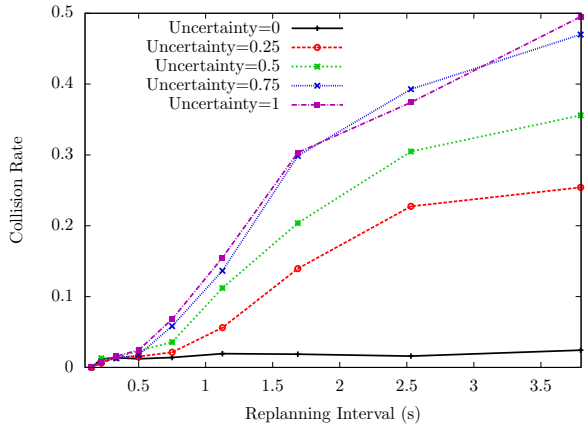
9.2.2 The Merit of Replanning in Uncertain Domains

Because VLOD planning only makes sense in a replanning environment, we first investigate the general relationship between domain uncertainty and replanning interval in our navigation planning experimental setup. The graphs in Figure 9.6 show the effects of varying replanning intervals and domain uncertainties on overall planning performance (Figure 9.6(a)), and on its two defining components, namely collision rate (Figure 9.6(b)) and total accumulated planning time (Figure 9.6(c)). These particular results were computed in the Hallway domain, with full detail ($t_{\text{LOD}} = \infty$). Each data-point represents a mean of 120 simulated trials, thus totaling 5400 simulated trials for the generation of the graphs (with most trials consisting of many replanning iterations). Each trial begins with the controlled body in the start location with randomly initialized opponent robots (positioned somewhere within their pre-constrained areas), and consists of repeated replanning and execution until the robot reaches the goal area.

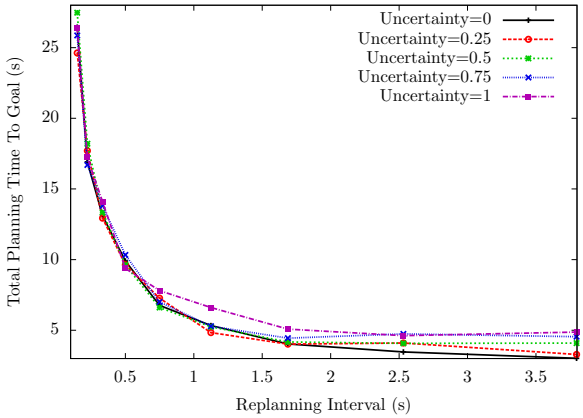
The performance graph in Figure 9.6(a) clearly shows that the optimal replanning interval (i.e., the replanning interval leading to a maximum performance value), depends strongly on the amount of uncertainty present in the domain. For domains with an uncertainty



(a) Impact of replanning interval and domain uncertainty on planning performance. Hallway domain, VLOD planning disabled ($t_{\text{LOD}} = \infty$).



(b) Impact of replanning interval and domain uncertainty on collision rate. Hallway domain, VLOD planning disabled ($t_{\text{LOD}} = \infty$).



(c) Impact of replanning interval and domain uncertainty on total planning time (accumulated over all replanning steps until goal was reached in execution). Hallway domain, VLOD planning disabled ($t_{\text{LOD}} = \infty$).

Figure 9.6: Analysis of the dependence between domain uncertainty and the optimal replanning interval in the Hallway domain.

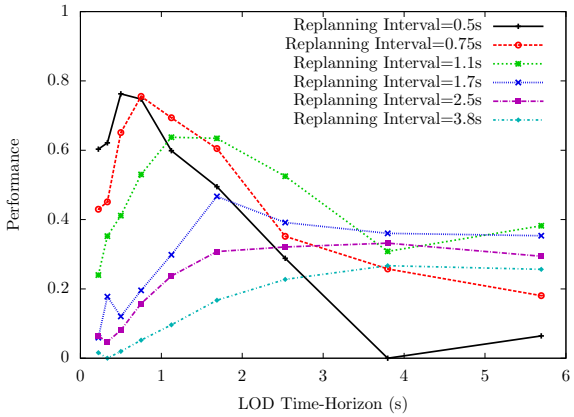
value of zero, the collision rate (see Figure 9.6(b)) remains unaffected, even for extremely long replanning intervals. This is to be expected, as the simulated execution will behave nearly identical to the predicted solution by the planner (where the only remaining difference is caused by the engine's natural non-determinism, as described in Section 2.2.2). Because shorter replanning intervals will lead to a rapid increase in overall planning time (see Figure 9.6(c)), maximum planning performance for domains without uncertainty is achieved by using longer replanning intervals. However, for domains with increased uncertainty, choosing a longer replanning interval clearly leads to higher collision rates. In fact, looking solely at collision rates, one would definitely want to choose the smallest replanning interval possible in order to minimize the rate of collisions during execution. The overall performance metric (Figure 9.6(a)) clearly shows this trade-off between collision rate and planning time. For domains with non-zero uncertainty, peak performance is achieved by selecting a replanning interval that is able to significantly reduce the rate of collisions while at the same time keeping planning times reasonable.

9.2.3 VLOD Performance Analysis

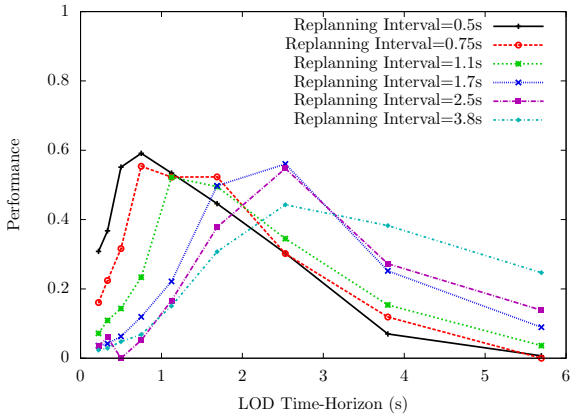
Having shown the general merit of replanning in uncertain physical navigation environments, we now investigate whether our VLOD planning approach is able to further improve planning performance.

We analyzed VLOD planning performance for varying t_{LOD} and replan intervals, both in the Hallway and the Maze domains under simulated execution uncertainty. Figure 9.7 shows the results. Each data-point in the graphs represents the mean over 120 trials, this time resulting in a total number of 6480 trials per domain. Note, that traditional planning methods effectively assume a t_{LOD} value of infinity, planning at maximum detail all the time. As the average solution length in our domains was typically in the range from 3 to 5 seconds, the maximum t_{LOD} in the graph (5.7 seconds) effectively mirrors the performance of traditional methods and acts as our performance baseline.

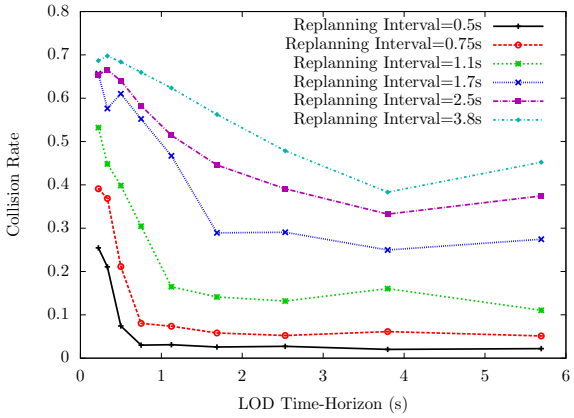
Figures 9.7(a) and 9.7(b) show the performance values for the Hallway and Maze domains respectively. In both cases, it is clear to see that VLOD planning has a positive impact on performance for all tested replanning intervals. Generally, we find the pattern that the VLOD planner achieves its highest performance with t_{LOD} values that are slightly greater than the corresponding replanning interval. This result makes sense, as using a t_{LOD} value lower than the replanning interval would mean that the robot will execute partial solutions that have not been planned with the maximum level of detail, thus likely to collide with obstacles. This reasoning is verified if we look at the corresponding collision rates shown in Figures 9.7(c) and 9.7(d). Here, selecting a t_{LOD} value lower than the replanning interval results in an agressive growth of collision rates.



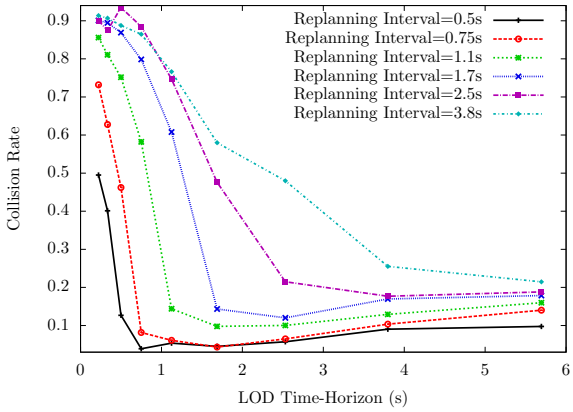
(a) Impact of t_{LOD} and replanning on performance. Hallway domain, Uncertainty=0.75.



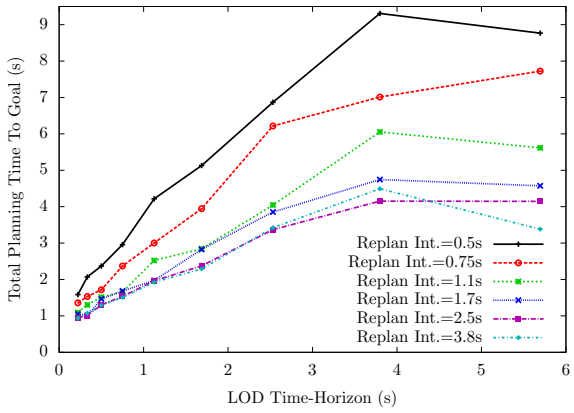
(b) Impact of t_{LOD} and replanning on performance. Maze domain, Uncertainty=0.5.



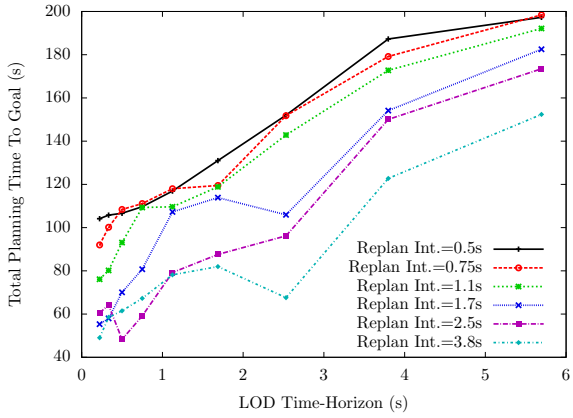
(c) Impact of t_{LOD} and replanning on collisions. Hallway domain, Uncertainty=0.75.



(d) Impact of t_{LOD} and replanning on collisions. Maze domain, Uncertainty=0.5.



(e) Impact of t_{LOD} and replanning on total planning time. Hallway domain, Uncertainty=0.75.



(f) Impact of t_{LOD} and replanning on total planning time. Maze domain, Uncertainty=0.5.

Figure 9.7: VLOD performance analysis for the Hallway (a,c,e) and Maze (b,d,f) domains.

The true benefit of VLOD planning, however, becomes clear when looking at the total accumulated planning time in Figures 9.7(e) and 9.7(f). For example, in the Hallway domain using a replanning interval of 0.5, we see an approximately 70% decrease in planning time, when reducing t_{LOD} from its maximum of 5.7s down to 0.5s. The reason for why the VLOD approach is able to perform so much faster, lies within the fact that a lower value of t_{LOD} allows the planner's search to find simpler solutions that ignore obstacle interactions in the future. In the Maze domains, we see a similar trend of shorter planning times with smaller values of t_{LOD}, but the effect is less pronounced. The reasoning for this behavior lies in the fact that the Maze domain remains significantly difficult to solve even when foreign-controlled bodies are ignored due to a low value of t_{LOD}. Although ignoring these dynamic interactions makes the search evidently easier, the planner still needs to find a trajectory that leads around the maze of static wall bodies without getting stuck in local minima.

Conclusively, achieving optimal planning performance depends on multiple factors. First, as we discussed already in Section 9.2.2, a replanning interval appropriate for the domain's level of uncertainty must be chosen. Given the replanning interval, maximum performance is then achieved by selecting a value for t_{LOD} that minimizes planning time and collision rate. Our overall performance metric tries to capture this trade-off, proposing a t_{LOD} value that not only reduces collisions, but that's also low on computational expense. Depending on one's particular needs and computational power available, one could weigh this metric differently to put special emphasis on either safety or computational time.

9.3 Chapter Summary

In this chapter, we have introduced VLOD planning for physics-based domains with poorly predictable bodies. We have tested its performance on two domains, using a rich simulated model. We have analyzed the impact of the LOD horizon on planning performance over different replanning intervals. Our overall results clearly show that VLOD planning is able to significantly cut down computational cost at little to no expense to collision safety.

Chapter 10

Related Work

In this chapter we discuss related work to this thesis. We cover existing techniques in motion planning, including approaches to the problems of robot navigation, behavioral computer animation, body manipulation, and replanning. We furthermore cover existing techniques for tactical robot control, especially in the robot soccer environment, and we look at existing approaches to the robot minigolf problem.

10.1 Discrete Motion Planning

A traditional approach for robot navigation planning in continuous domains is to discretize the underlying continuous state space into a finite set of searchable states. In 2D navigation planning, the state space can be discretized into a grid of equally-sized square cells, and the action space into transitions between grid cells. Given such a discretization, it is possible to apply traditional graph search algorithms to find a solution by traversing the grid [58]. Grid cells in this case are treated as vertices and transitions are treated as edges. Edge weights represent the cost to transition from one cell to another. Classical examples of graph search algorithms used for grid-based path planning are Dijkstra's algorithm [29] and A* [28]. Dijkstra's algorithm is both complete and optimal, because it is guaranteed to find the shortest existing path in the grid. A* improves upon Dijkstra's algorithm by utilizing the cost of the path from the initial state to a node plus an estimate of the cost from a node to the goal, as a heuristic for determining the order of node traversal during the search. Depending on the domain, this technique can yield significant improvements in computational cost over Dijkstra's algorithm. A* is guaranteed to be complete and optimal if its heuristic is *admissible* (i.e., the heuristic function never underestimates the distance to the goal). One extension to A* is the D* algorithm [82] which allows for more efficient

replanning in the case that edge costs have changed from the ones used in a previous planning query. It does so by re-using dependency information of the previous search and updating only the weights of affected edges. D* is complete and optimal assuming an admissible heuristic similar to A* [82].

Although these graph search algorithms carry the feature of completeness, they are only complete within the particular resolution of the grid. Since the grid is a discretization of a continuous state space, such grid-based algorithms are called *resolution-complete* [58] when being used in inherently continuous domains. Furthermore, due to their graph-based nature, grid-based approaches are not very scalable to higher dimensional state or action spaces, as each additional dimension would result in an exponential increase of numbers of grid-cells and edges respectively. Finally, due to their discretization requirements, grid-based models are not well suited to model continuous robot kinematics and dynamics.

10.2 Sampling-Based Motion Planning

Sampling-based planning approaches intend to overcome the need for discretization and the scalability problems encountered by grid-based planners under high grid resolutions and dimensionality. Rather than attempting to discretize the continuous space and then completely explore a potentially intractable large set of states, sampling-based approaches rely on randomization to pick samples directly from the underlying continuous space [62]. While such algorithms are not *complete*, many of them are considered to be *probabilistically complete* [58], implying that given an infinite amount of time, they would be complete, as their sampling-mechanism would cover all of the search space. However, for finite time applications this means that there is no guarantee that the algorithm will find a solution if it exists, nor will it be able to determine that no solution exists. Nevertheless, sampling-based planning algorithms are very popular and have been used in many robotics applications.

A pioneering sampling-based planning algorithm is Probabilistic Roadmaps (PRM) [44]. Here, random states and collision-free connections between these states are sampled from the state space. States and their connections are added to a graph structure, also known as a roadmap, which can then be searched using traditional graph-based search algorithms [59]. Lazy-PRM [10] is an enhancement to the PRM algorithm which postpones the collision checking component of the algorithm until the query stage. Depending on the domain, it has been shown that Lazy-PRM can result in significant speed-ups over standard PRM [10]. However, collision checks are not the only possible bottleneck in a PRM-based planner. In challenging domains, and especially when kinematics and dynamics are involved, establishing and executing the connection between two vertices of the roadmap can become a non-trivial task and may require using a local planner [38, 91].

Rapidly-Exploring Random Trees (RRT) are another sampling-based planning technique for continuous spaces [57]. RRT uses repeated random sampling to grow a search tree through the state space with the objective that one of the tree's leaves will eventually reach the goal-state (see Section 5.3.3 for a visual example). Similarly to PRM, the tree's nodes are generated by randomly sampling from the continuous underlying state space whereas the edges represent state transitions from a parent to a child node. The advantage of the RRT algorithm is that the resulting trees tend to rapidly cover the reachable space. Furthermore, basic RRT has been shown to be probabilistically complete [57], meaning that given an infinite amount of time, it will find a solution if it exists. Several variants of RRT exist to meet the specific challenges of different robot navigation tasks [15, 47, 67, 86], including navigation under kinodynamic constraints [60].

One significant limitation of RRT-based approaches however, is the requirement of a distance metric which is able to express the distance between two states. Depending on the particular kinematics and dynamics of the domain, obtaining a reliable distance metric can in fact become a computationally intensive and non-trivial process [46]. Furthermore, much of RRT's performance depends on the selection of its underlying sampling distribution, a process that is also highly domain-dependent [97].

Other types of randomized tree-based planners attempt to work around the requirement of a distance metric by employing different node selection schemes. PDST [48] attempts to maintain well-balanced coverage through the configuration space by using a subdivision scheme on its distribution of samples.

In this thesis, we introduce two novel sampling-based planning algorithms, BK-RRT and BK-BGT, that both rely on Skills and Tactics to purposefully constrain the set of searchable actions, allowing a more informed search than traditional Tactics-free approaches. BK-RRT is an extension of basic RRT and requires a distance metric, whereas BK-BGT uses a novel node selection method that does not require a distance metric.

10.3 Efficient Replanning and VLOD Planning

Uncertain environments require the use of replanning to handle the divergence of the world's true state from the planner's predictions. This thesis introduces Variable Level-Of-Detail (VLOD) planning to plan more efficiently in domains that require frequent replanning due to uncertainty.

There exist several approaches that aim to increase the replanning performance of sampling-based planning algorithms. ERRT [14] is an extension to RRT which allows for fast and

stable replanning through the introduction of a waypoint cache. In ERRT, samples of a previously found path are stored and used as a goal-bias during following iterations, thus probabilistically biasing the search towards previous results. In most instances, this ERRT approach results in faster planning times and less oscillations between planning iterations. ERRT does not explicitly model dynamics, but can be combined with lower level trajectory control [15, 43] and a dynamic window method [12, 15, 33] to prevent the robot from entering a state of *inevitable collision* [34], allowing safe and collision-free navigation planning in a fast-paced dynamic environment. Other related approaches to increase RRT's replanning abilities are DRRT [32] and Multipartite RRT [96]. Instead of biasing the search towards previous samples, these algorithms store entire portions of previous search trees to be re-used during search. In dynamic environments, replanning performance can be further enhanced by retaining past planning information and combining it with a greedy exploration strategy [9].

Instead of caching previous planning results, another approach for increasing replanning performance is to reduce the computational complexity of each planning iteration directly by relaxing the planning problem itself. A common method used in robot motion planning is to introduce a layered planning architecture, where a global plan is computed by a planner that uses a higher-level abstraction of the world (e.g., graph-based), which is computationally efficient, but less accurate and typically unaware of lower-level dynamics. The global plan produced by this planner is then handed to a lower-level, local planner (or other control system) that performs a near-term finite-horizon search with the goal of reaching the next waypoint of the global planner's solution. Several variations of such dual-layer planning approaches exist, differing mostly in the concrete planning methods used for each layer. Existing combinations of global and local planning include grid-based distance-transforms on top of potential fields [88], PRM on top of potential fields [91], MDPs on top of RRT [21], D* on top of a state-lattice [75], and D* on top of a custom random sampling-based dynamics planner [45]. One fundamental weakness shared by all of these dual-layer approaches is the fact that the global planner ignores certain details, such as dynamics, to perform its search more quickly. Hence, the global planner cannot guarantee that the local planner will be able to find a valid solution to reach the next waypoint of the global plan. For example, when controlling a vehicle with dynamic constraints, such as an airplane, a grid-based global planner might produce a sequence of waypoints that lead to the goal state, but that are not actually performable by the aircraft (e.g., because of overly tight turns), thus preventing the local planner from finding a valid solution.

In this thesis, we introduce Variable Level-Of-Detail (VLOD) planning as a method for increasing replanning efficiency in uncertain multi-body environments. VLOD does not focus on caching previous planning results, but instead attempts to reduce planning complexity directly by ignoring certain types of multi-body collisions that occur far in the planning future and will therefore only be relevant at a later replanning iteration. VLOD planning

circumvents the global planning inaccuracy problems of layered planning approaches by relying on a single fully dynamic planner that generates global plans leading all the way to the goal state, only ignoring a certain subset of multi-body interactions that are assumed to be locally solvable.

10.4 Animation Planning

All of the related planning work mentioned so far aims primarily at solving the collision-free navigation problem of finding a sequence of actions that leads from an initial state to some target state. We are, however, interested in planning problems beyond navigation; in particular problems with additional constraints on the intermediate states of the solution sequence (see Section 2.4). Such problems are common in the field of computer graphics, where the goal is to create an animation involving one or more simulated bodies, that will not only result in a particular outcome (i.e., the goal state), but also achieve a particular desired visual behavior during its intermediate states.

Several animation approaches relax the constraints of the problem by relying on the concept of *physical plausibility* [7] rather than physical correctness. This relaxation is achieved by introducing additional control parameters such as the slight perturbation of collision normals [7]. Markov Chain Monte Carlo (MCMC) can be used to enhance this technique in order to compute solutions to pre-specified constraint problems [22]. However, the approach suffers from long computational times, even for relatively low-dimensional problems [22]. Furthermore, relaxing the problem through physical plausibility is not helpful for robotics planning applications which require physical accuracy.

Other approaches focus on user-interaction as a means for solving difficult constraint problems. For example, one approach uses an interactive interface which allows the user to manipulate objects at any point during the simulation [76]. This system uses random sampling and gradient descent to support the user in the search for constraint-satisfying solutions. Such interactive systems can be enhanced significantly by allowing the user to perform spatial queries to rapidly cut down the number of possible solutions, being able to generate desired animations for scenes involving complex multi-body dynamics [85]. Another recent work demonstrates how sampling-based planning techniques can be combined with user-provided constraints to solve animation and navigation planning of deformable bodies [69]. The inherent downside of these semi-interactive techniques is clearly that problem-specific user-involvement is required in order to find valid solutions. Although relying on human knowledge might be acceptable for certain computer animation tasks (such as the creation of animated movies), it becomes unacceptable for real-time autonomous applications, such as robotics.

There also exists a body of work which employs motion planning and control techniques to solve constrained computer animation problems. One approach uses a Finite State Machine (FSM) as a behavioral modeling tool to describe the possible motion sequences of humanoid characters [53]. The authors use an A*-based algorithm to compute motion paths for 100 animated humanoids navigating an environment with moving obstacles. Unlike our sampling-based Skills however, the behaviors in the FSM are fully deterministic, thus leaving the choice of behavior as the only search parameter. Furthermore, the behaviors consist of pre-determined action sequences, unlike our Skills that can perform state-dependent, intelligent action generation during the search. A recent performance improvement to this approach uses precomputed search trees to speed-up the planning to real-time performance [54].

Another approach on animating humanoid characters introduces task-specific controllers and a framework for switching between them by learning the appropriate state transition preconditions via a SVM [31]. The result is a stable reactive controller that is applicable under a variety of conditions. However, the approach itself does not perform any planning, but instead relies solely on learning a strong reactive control strategy.

Other work covers the problem of animating human character motions when interacting in complex environments [79, 90], including the physically plausible animation of object manipulation. In terms of planning, however, these approaches focus solely on navigation planning to, with, and around objects, ignoring the actual physics required for body manipulation (such as grasping, or the purposeful exertion of collisions), effectively preventing these approaches to be applicable for physical robot manipulation tasks without further enhancements.

Several modern approaches rely on the search of prerecorded motion databases or motion graphs to animate human characters in problems with goal and intermediate constraints [20, 78, 84]. These databases contain pre-recorded action sequences, but use novel techniques to allow the generation of intermediate sequences, either through the means of learning a statistical model [20], controllers [84], or interpolation [78]. However, all of these approaches focus on the task of computer animation, and it is not clear whether they are applicable on physical robots without specifically addressing the problems of prediction uncertainty caused by model inaccuracy. Furthermore, all of these approaches focus solely on navigation planning, and do not address manipulation planning.

In this thesis, we introduce sampling-based Skills and Tactics as a method for efficiently constraining the set of searchable actions during planning. Although the core focus of this thesis is robot motion planning, we also show that our planning methods can be efficiently applied in challenging computer animation domains involving multiple interacting rigid bodies with intermediate constraints on their animation sequence (i.e., the many-dice domain and the pool domain). Furthermore, thanks to the state-dependent decision making

capabilities of our non-deterministic Skills and Tactics, our planner is able to quickly generate novel and interesting animation sequences in the simulated robot soccer domain which requires the goal-driven manipulation of non-actuated bodies in an adversarial environment.

10.5 Manipulation Planning

The previously mentioned approaches focus on solving problems with goals or intermediate constraints on the controllable body. The planning approaches contributed in this thesis also solve problems that require the purposeful manipulation of passive bodies, e.g., a ball in robot soccer and in robot minigolf.

Much existing work in the area of manipulation planning focuses on the problem of controlling a robotic arm with an end-effector to move a passive body from one configuration to another. Positioning the end-effector around the body and transporting the body through an obstacle-ridden environment is in essence a special case of navigation planning. Hence, experimental domains of sampling-based planners frequently include simulated examples of end-effector positioning and navigation [47]. However, such pure navigation planning approaches oversimplify the manipulation problem by ignoring the task of actually grasping the body. To become more applicable on real-world manipulators, much work focuses on specifically modeling grasping [68] and explicitly integrating it as a necessary component of the manipulation planning sequence [80, 83].

However, the problem of collision-free end-effector positioning and grasping only describes a subset of possible manipulation problems. A different type of manipulation problems are so-called *dexterous* manipulation tasks [73], where the body to be manipulated cannot be temporarily "fused" with the end-effector through the means of grasping, but instead can only be manipulated through the means of targetted collisions. As we have discussed in Section 2.4, such dexterous manipulation can become a difficult problem, because it is not possible to directly apply navigation controls to the manipulatable body, but instead requires higher level knowledge about how to best exert forces through the means of collisions.

To perform dexterous manipulation, some work focuses on analytically modeling specific types of possible collisions in order to optimally compute the required actions to purposefully manipulate the body [63, 65, 73]. A general limitation of such analytic solutions is that their mathematical definition depends strongly on the specific types and shapes of the bodies involved, therefore making them task-dependent, and not trivially applicable to unencountered or larger scale problems.

Nevertheless, task-specific analytic approaches to dexterous manipulation have been successfully applied in several real-world robot manipulation domains. For example, it has been demonstrated that task-specific models can be used to solve challenging control problems, such as juggling a ball [1, 2]. A more recent example is the development of task-specific analytic simulation and control models to implement a pool-playing robot manipulator, focusing particularly on the collision model between two spheres [41, 61].

In this thesis, we introduce algorithms that rely on general physics-based simulation techniques to model rigid body interactions. Instead of devising complex task-specific mathematical models to generate analytic manipulation solutions, we rely on a combination of forward planning and intelligent action-sampling via Skills and Tactics. This thesis furthermore shows that our Tactics-based algorithms are general enough to be applicable to a vast variety of domains, and are able to find solutions quickly, even in challenging multi-body manipulation problems (e.g., robot soccer and robot minigolf).

10.6 Robot Tactics and Control Architectures

In addition to addressing planning problems with additional constraints (Section 10.4) and manipulation challenges (Section 10.5), this thesis also addresses the problem of efficiently planning in domains that contain foreign-controlled adversaries (e.g., robot soccer). Planning goals for such domains go beyond the objective of safely navigating, and instead focus on out-playing the opponent, typically requiring higher level tactical decision making.

Because we are interested in solving tasks with higher level goals, it is important to also consider existing work which is not solely focused on motion planning. Cognitive architectures such as SOAR [52] are frameworks which aim to model various facets of intelligent reasoning. Traditionally, they rely on symbolic representations and have put little focus on representing the low-level continuous physics or rigid body dynamics of a domain. Nevertheless, higher level symbolic reasoning approaches can be applied to control characters in continuous real-time environments [51]. One particularly interesting insight for our work is that the ability to model and anticipate an opponent's movement can be a helpful addition for planning in adversarial domains [50].

Besides such symbolic cognitive architectures, there also exists a vast body of layered control architectures directly aimed at multi-robot behavioral control problems [8, 13, 25, 26, 35, 56, 74]. One such approach, that our work is partially based on, is called *Skills, Tactics, and Plays (STP)* [13]. In STP, a robot's behavior is modeled as a *Tactic*, a finite state machine of several *Skills*. The Tactic and each of its Skills can perform state-dependent decision making. For example, the Tactic determines when to perform state

transitions between Skills based on external events. Similarly, a Skill acts as an informed real-time controller for a robot.

The STP model allows the design of elaborate reactive behaviors to tackle complex tactical domains, such as real-time multi-robot soccer [13]. However, in the traditional STP model, motion planning is treated as a lower level function that is invoked by individual Skills and that solely focuses on the safe navigation problem, without modeling ball interactions, and without being aware of the actual strategic goals of the Skill and higher level Tactic that invoked the planner [14]. Therefore, higher level tactical decisions are performed independently of lower level motion planning, making it challenging to implement behaviors that are able to perform simultaneous ball dribbling and navigation, that is both tactically sound and physically stable. Other robot soccer teams use similar control paradigms, implementing motion planning only as a means for safe navigation, but not for tactical decision making or ball manipulation [25, 56].

Instead of relying on deterministic and reactive Tactics that invoke a separate and tactically uninformed navigation planner, the work in this thesis uses Skills and Tactics as an action sampling model within a physics-based planner. In order to allow the planner to search over different possible perturbations of tactical executions, we extend the traditional Skills and Tactics model to become non-deterministic. Using sampling-based Skills and Tactics, in combination with a rich physical multi-body model of the domain, our planner is able to effectively find novel and stable control solutions for challenging and adversarial robot soccer ball dribbling scenarios.

10.7 Robot Minigolf

This thesis contributes an autonomous minigolf-playing robot. So far, there exist only few attempts in creating golf-playing robots. One approach uses an overhead-mounted robot arm that is able to putt a golf ball around a table and into one of multiple holes with hardcoded positions [42]. Putting is also used as an evaluation metric for improved teleoperation of a robotic arm [70]. Another approach aims to play a small-scale version of traditional golf by autonomously putting a ball into a hole using a small differential-drive robot with a fuzzy reactive controller [39]. However, none of these existing experiments in robot golf feature any kind of physics-based prediction model and, consequently, none of them are able to plan and perform sequences of purposeful collisions, e.g., bouncing the ball off a wall, and avoiding moving obstacles, which are typical challenges in a game of minigolf.

In this thesis, we introduce a minigolf-playing robot that is able to autonomously solve

complex novel minigolf course configurations using physics-based planning. The physics-based planning model allows the robot to solve courses that require purposeful collisions involving multiple types of rigid bodies.

10.8 Model Parameter Optimization

Planning is a process that requires estimation of the future states of the world before execution takes place. Such estimation can be viewed as the result of a simulation of the real execution, using models of the real world. Simulation accuracy depends strongly on the model's parameters.

Traditionally, much work in model parameter optimization focuses on the purposes of reactively acting well, and not for planning. For example, a variety of previous work employs parameter optimization to improve locomotion control accuracy in robot tasks. One example is the use of a genetic algorithm to fine-tune the gait parameters of legged robots [23]. Note, however, that this approach does not aim to improve the accuracy of a simulated model, but of the walking algorithm's parameters itself, thus requiring use of the physical robot during optimization.

Another approach also attempts to improve the overall walking performance of legged robots, but does so by optimizing simulation model parameters [55]. However, the approach requires massive computations on a 60 node computer cluster to deliver results in a reasonable amount of time. Furthermore, due to the nature of the physics parameters involved, our approach is able to use the more deterministic and faster Nelder-Mead Simplex optimization method [49, 71], instead of an evolutionary approach [3].

In this thesis, we present a method for optimizing the parameters of physics-based planning models, focusing specifically on achieving accurate simulations of physical multi-body interactions, such as collisions. Our approach relies on complete episodes of real world executions as ground-truth and automatically attempts to find a set of parameters which lead to simulations that best match the pre-recorded ground-truth sequences. We test our approach in the robot minigolf environment.

Chapter 11

Conclusion and Future Work

This chapter summarizes the contributions of this thesis and discusses promising directions for future work.

11.1 Contributions

This thesis makes the following contributions:

- **A Formalization of Physics-Based Motion Planning in Dynamic Multi-Body Environments**
  To successfully plan in physically complex environments, a planner requires a model that is sufficiently detailed to accurately predict the dynamics of the real world. This thesis introduces physics-based motion planning, contributing a formal domain-independent planning model based on rigid body dynamics, defining the state space (i.e., the bodies' mutable states at a point in time) and the domain (i.e., the bodies' immutable parameters and the action space). State transitions are performed by invoking a black-box physics engine which applies the action's forces and torques to a particular state and forward simulates the rigid body system by some timestep. Using the physics-based planning model, this thesis shows that it is possible to represent a rich variety of interesting and novel motion planning problems.

  This thesis furthermore contributes an analysis of the different planning characteristics and challenges that can be encountered when using the physics-based planning model. In particular, this thesis introduces a formal classification hierachy of different body types, and discusses how planning problem difficulty depends strongly on

the properties of body controllability and predictability.

- **Non-Deterministic Skills and Tactics as Action Sampling Model**
  To make search in the physics-based control space more computationally feasible, this thesis introduces non-deterministic *Skills* and *Tactics* as a method to reduce the size of the searchable action space. These sampling-based Skills and Tactics make it possible to encode a variable amount of domain-dependent knowledge (e.g. state-dependent action selection) into the planning stage, making search feasible even for challenging problems, such as the purposeful manipulation of passive bodies.

- **Two Efficient Tactics-Driven Planning Algorithms: BK-RRT and BK-BGT**
  This thesis contributes two physics-based planning algorithms that both rely on non-deterministic Skills and Tactics as action sampling model, namely *Behavioral Kinodynamic Rapidly-Exploring Random Trees (BK-RRT)* and *Behavioral Kinodynamic Balanced Growth Trees (BK-BGT)*. BK-RRT's node selection methodology is based on traditional RRT and therefore requires the definition of a global sampling space and a distance metric. BK-BGT uses our newly introduced Balanced Growth Trees node selection method, not requiring a global sampling space or distance metric, and yielding a less informed, but significantly faster and more scalable tree growth than BK-RRT.

  To further increase planning efficiency, this thesis contributes the *busy flag* as a method for preventing unnecessary state duplications that can be caused by temporary deterministic action generation of Skills. Additionally, the thesis contributes the *RollBack* function to remove stale branches from the search tree, reducing the memory requirements during search.

  Finally, this thesis also introduces variations of the core BK-RRT and BK-BGT algorithms, namely a *hybrid planning* scheme that probabilistically switches between RRT and BGT node selection methods during planning, and an *anytime planning* version that is able to provide partial solutions if interrupted at any point during planning.

- **Evaluation and Comparison of Planning Algorithms**
  This thesis contributes an extensive performance evaluation of the BK-RRT, BK-BGT, and hybrid algorithms in five different simulated domains, namely *navigation*, *pool table*, *robot soccer*, *robot minigolf*, and *many-dice*. The results suggest that algorithm dominance is dependent on the domain and on the particular Skills and Tactics, with BK-BGT exceeding in domains that are strongly Tactically constrained, and BK-RRT performing better in traditional navigation settings.

- **Variable Level-Of-Detail Planning**
  This thesis introduces *Variable Level-Of-Detail (VLOD)* planning as a method for

reducing overall planning time in uncertain multi-body execution environments that require replanning. We introduce the VLOD time horizon t_{LOD} and analyze its dependence on the replanning interval, which in turn is chosen depending on a domain's uncertainty. We use two example domains to demonstrate that VLOD planning is able to significantly reduce cumulative planning time with little to no expense to collision safety.

- **A Minigolf-Playing Robot**
  This thesis contributes a minigolf-playing robot that uses physics-based planning to autonomously solve complex course configurations. We integrate our planning algorithms into the CMDragons robot system and use a global vision system to allow the free configuration of novel minigolf courses. Although some previous attempts in robot golf exist [39, 42, 70], we believe that our approach is the world's first minigolf-playing robot to solve courses that require purposefully bouncing the ball off obstacles, as it is frequently required in real-world minigolf courses. Furthermore, we also show that our approach can successfully solve courses that contain moving obstacles.

- **Automated Parameter Optimization Based on Real Episodes for Physics-Based Models**
  A physics-based planner's prediction accuracy depends strongly on the various model parameter values that describe the physical properties of the domain's rigid bodies. This thesis introduces methods for increasing the model's accuracy by automatically optimizing parameter values and comparing the simulated execution to a previously collected set of ground-truth sequences. We test our approach in the robot minigolf domain and show that the optimized set of parameters ultimately leads to significantly better execution success rates.

- **A Ball-Dribbling Robot Soccer Behavior, using Physics-Based Planning**
  This thesis contributes the *PhysicsDribble* behavior which relies on our physics-based anytime BK-BGT planning algorithm with sampling-based Skills and Tactics to compute its actions. We integrate the behavior into the CMDragons robot soccer system and we show that it is able to significantly out-perform a traditional state-of-the-art reactive robot soccer control behavior. We furthermore present visual results from RoboCup 2009, where we tested the PhysicsDribble behavior under real competition conditions, allowing our robot to effectively out-dribble the opponent robots, and leading to a successful goal-shot for our team.

11.2 Future Work

Following are several possible directions for future work:

- **Learning of Skills and Tactics**
  The planning algorithms presented in this thesis rely on the definition of Skills and Tactics for efficient action sampling. Assembling a Tactic, defining its transition probabilities, and selecting the Skills' internal constants and sampling distributions requires a certain degree of domain knowledge and human expertise. A next logical step for future work would be to develop techniques which can *automatically* learn a Tactics model that is optimized for a particular domain.

  One possible approach would be to employ a parameter optimization algorithm similar to the one used for the physics model optimization approach in this thesis. To learn Skills and Tactics, however, the algorithm's evaluation function would need to evaluate the overall performance of the planner, while adjusting the parameters of the Skills and Tactics itself (i.e. the sampling distributions, transition probabilities, and possibly even the actual set of Skills itself). In order to prevent such an approach from over-fitting to a particular problem instance, one would need to ensure that the planner performance is measured over a broad enough random set of problem instances that are likely to be encountered within the domain. Such an approach should automatically yield a Skills and Tactics configuration that is strongly optimized to perform well in its particular target domain.

  A succeeding step would be to adjust Skills and Tactics parameters *online*, using continuous monitoring of planning and execution success, *while* the robot is operating in its environment. Such an online-optimization approach would allow a robot to become a truly adaptive problem solver, autonomously adjusting and improving its planning strategy to perform best, even in changing real-world environments.

- **Multi-Robot Coordination**
  The work in this thesis focuses primarily on single-robot motion planning. An interesting question is how to perform physics-based motion planning with *multiple* robots. Our current approach extends and embeds the Skills and Tactics components of the traditional STP model into the planning stage, leaving the question open as to whether and how the *Plays* component of STP could be integrated into the planner in order to allow search over different possible multi-robot role assignments. Another question is how to plan in domains that require multiple robots to coordinate their actions (e.g., two robots that collaboratively need to manipulate a passive body). Presumably, one could introduce methods to allow information sharing and action synchronization between multiple Tactics, possibly similar to the

approaches that are used in the reactive STP model. Such coordination issues would be particularly challenging if each robot would run their own planner instance, thus requiring inter-planner communication.

- **Application on High-Dimensional Action Spaces**
  In this thesis, we integrate and evaluate our physics-based planning methods on the CMDragons robot platform to successfully plan in the robot soccer and robot minigolf domains. For future work, it would be interesting to apply our approaches to a larger variety of robots, focusing particularly on robots with many degrees of freedom, such as humanoids. How to best apply Skills and Tactics to control the high-dimensional action space of a humanoid robot is still an open research question. One possible approach might be to encode lower-level control and planning approaches, such as walking and footstep planning, as shared lower-level building blocks (i.e. sub-Skills) that can be configured and invoked by sampling-based Skills.

- **Faster Planning**
  The only true remaining restriction from applying Tactics-driven, physics-based planning in arbitrarily complex real-world robot domains are its computational requirements. Although the emerging visual results and behaviors in the simulated domains (i.e. the 1-vs-3 robot soccer) are impressive, they also took several seconds to compute. In order to implement the *PhysicsDribble* robot soccer behavior in the real-time CMDragons system, the Tactics model had to be significantly reduced in complexity to still work effectively even with very small search tree sizes.

  Therefore, an important direction for future work should be to further increase planning speed. Algorithmically, it should be possible to parallelize the planning algorithm, for example by handing off different sub-branches of the search tree to different threads. Especially with the current rise of multi-core architectures and massively parallel general purpose GPU processing, the potential for overall planning speed increase through parallelization is vast. Assuming these computational trends will continue, the future should allow deeper planning with increasingly elaborate Tactics models, ultimately enabling more and more intelligent and creative decision making by robots.

Appendix A

The CMDragons Multi-Robot System

The robot soccer and robot minigolf experiments in this thesis are implemented using the *CMDragons* multi-robot system. CMDragons is Carnegie Mellon's RoboCup Small Size League (SSL) robot soccer team. The CMDragons system is the result of a collaborative effort and builds upon the research and development contributions of its current and past team members, dating back as far as 1997 [18, 87]. Special acknowledgment goes to James Bruce, for developing most of the current CMDragons software framework, and Mike Licitra for developing the current CMDragons robots.

In this appendix, we briefly describe the RoboCup Small Size robot soccer league. We then introduce the CMDragons robot hardware and its capabilities. Finally, we describe the overall architecture and individual components of its control software.

A.1 The RoboCup Small Size League (SSL)

RoboCup is an international initiative with the primary goal to foster research in robotics and AI through the means of friendly robot soccer competitions. RoboCup features multiple robot soccer leagues, each with their own unique challenges.

In the Small Size League (SSL), teams play fully autonomous 5 vs. 5 robot soccer matches with an orange golf ball on a field of approximately 6m×4m (see Figure A.1). Each team is allowed to freely develop their own robot hardware, with the only constraints being the robot's size (maximum diameter of 18cm and maximum height of 15cm) and its ball

manipulation capabilities (maximum enclosure of the ball is limited to 20%, only 1 degree of freedom can be removed from the ball during dribbling, and kick speed is limited to 10m/s). Robots can be radio-controlled through off-board computation, as long as the entire system is fully autonomous. For robot and ball localization, the RoboCup Small Size League allows the use of an off-board global vision system with cameras mounted above the soccer field.

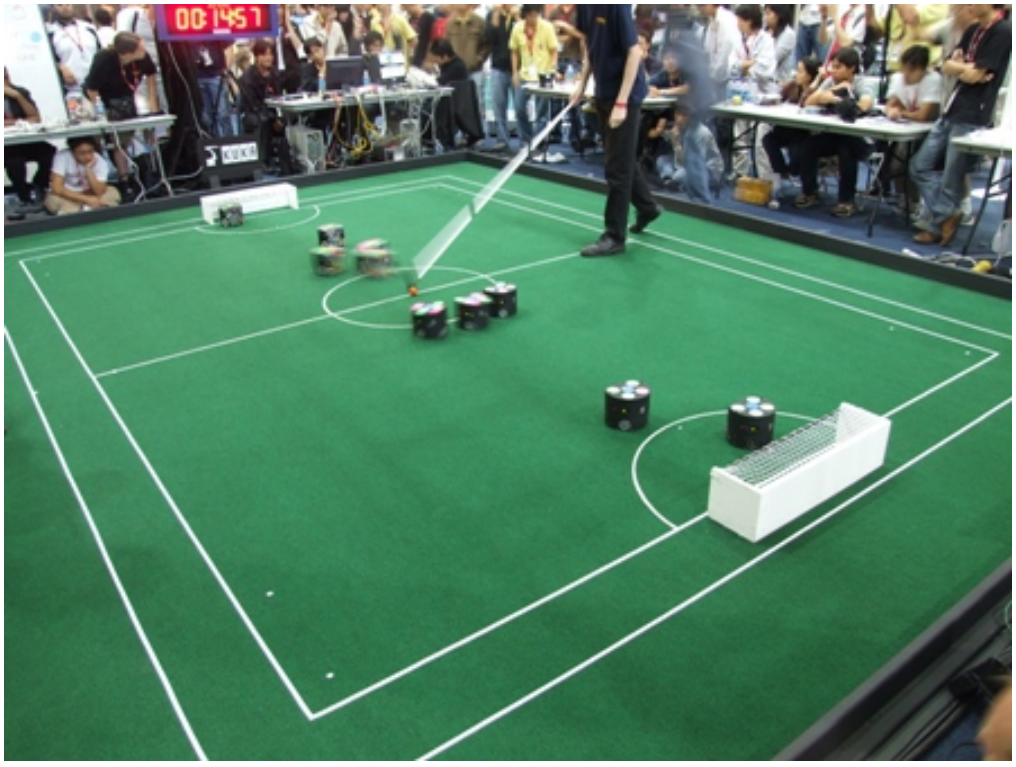


Figure A.1: A game in the RoboCup Small Size League.

The Small Size League is the fastest-paced league at RoboCup, with the stronger teams driving around 2.5 m/s and kicking at the maximum allowed 10 m/s, requiring extremely fast decision making and high accuracy. Challenges in the league include multi-agent team strategy, safe navigation at high velocities, and purposeful ball manipulation. To be competitive in the SSL, teams must excel at hardware engineering, algorithmic design, and software development.

A.2 Robot Hardware

The CMDragons system consists of seven custom-made homogeneous robots (with a maximum of five used in a game). Figure A.2 shows multiple CMDragons robots, one without its protective cover. The core of the current robot hardware has been developed in 2006, with minor upgrades and maintenance modifications in recent years. The CMDragons robot hardware is still highly competitive and allows the robot team to perform close to optimal within the tolerances of the league's rules.

A.2.1 Robots

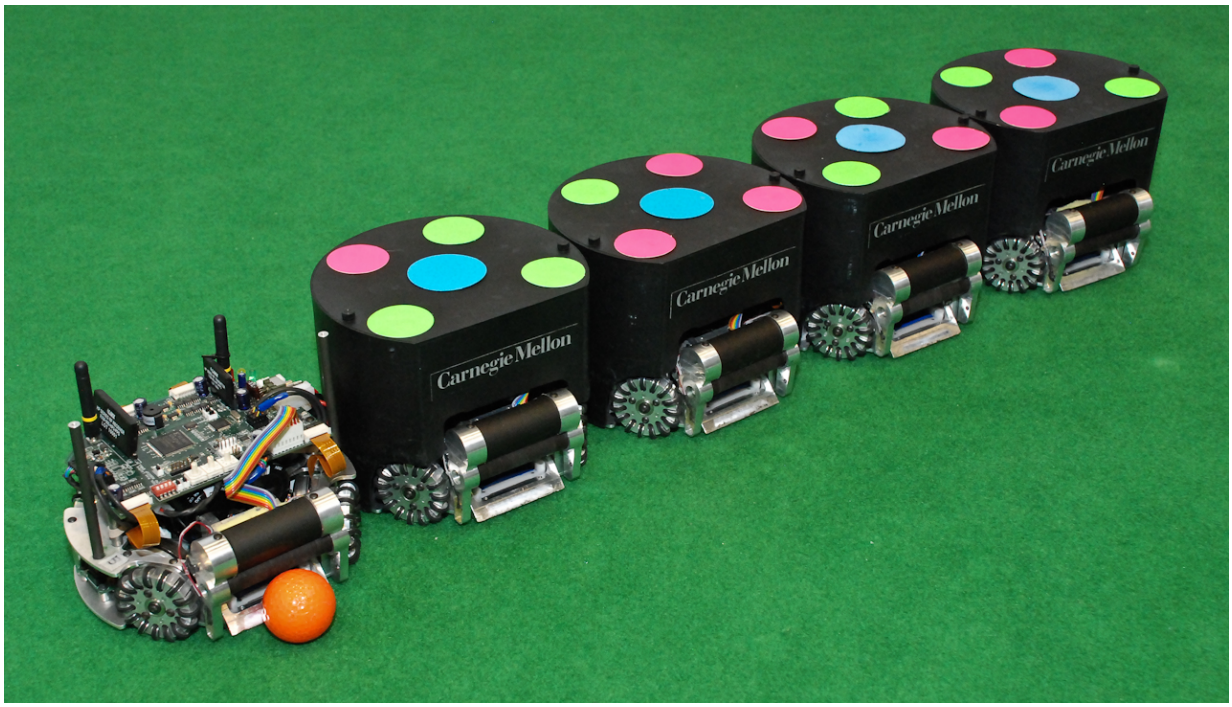


Figure A.2: Five CMDragons robots, one shown without its protective cover (robots developed and constructed by Mike Licitra).

Each robot features a 4-wheel omni-directional drive system, propelled by four 30 Watt brushless motors, each equipped with an optical encoder, enabling accurate odometry measurements. Each wheel is custom-made and is composed of 15 outer miniature wheels, all freely rolling orthogonally to the main wheel's axis of rotation.

Each CMDragons robot carries two kicking mechanisms, for flat kicks and upwards "chip"

kicks respectively. Both kicking mechanisms consist of custom-built solenoids driven by a shared bank of capacitors, charged to approximately 200 Volts. The flat kicker is able to kick the ball with up to 15m/s, but its strength is fully controllable to allow kicking at slower velocities. The chip-kicker accelerates the ball indirectly by hitting a metal "paddle" mounted at the front-bottom of the robot, which in turn is able to strike the ball at an approximately 45° upwards angle. The chip kicker is capable of propelling the ball up to 4.5m before hitting the ground, but can also be controlled to kick shorter distances. In order to assure that the kicker is only triggered if a ball is truly within kicking range, each robot features an infrared sensor and emitter mounted to the edges of the kicker opening, letting kicks only occur if the infrared beam is interrupted by the ball.

For ball manipulation, the robot features a so-called "dribbler": a rubber-coated, motorized bar that is able to exert backspin on the ball. This dribbler-bar is mounted on a servo-retractable hinged damper, allowing improved pass-reception and non-obstructed chip-kicking. One significant hardware improvement introduced in 2010 (thanks to CM-Dragons team-member Joydeep Biswas) is a novel dribbler assembly that encloses the robot's infrared sensor and the dribbler's motor, thus "shielding" all valuable and fragile exterior components from collisions with opponent robots.

The robot features two electronics boards. The "kicker-board" is mounted directly on the robot's base-plate and solely handles the high-voltage charge and discharge of the kickers' capacitors. The robot's main-board is mounted above the core robot assembly and performs motor control, local sensing, and communication. On-board processing is performed by an ARM7 processor and a Xilinx Spartan FPGA. For communication, the robot features a Linx HP3 transmitter and receiver that operate in the 902-927 MHz frequency range. Although technically able to perform bi-directional communication, the system relies mostly on one-way communication from the off-board computer for control only.

A.3 Software

CMDragons features a modular software infrastructure (see Figure 7.2) consisting of the four components *Vision*, *Server*, *Soccer*, and *GUI*, that are interconnected via UDP networking. The entire system has been implemented in C++ and currently runs on a Linux platform. The Server module receives the raw positions and orientations of all robots and the ball from the Vision module at 60Hz intervals. The server's Tracker then post-processes this raw data using a Kalman filter [89], providing a relatively accurate estimate of the positions and velocities of the robots and ball. These state estimates are then integrated into the Soccer module's world model. The Soccer module implements all the intelligent

robot soccer decision making, including multi-robot strategy, and single-robot control. The output of the Soccer module is a set of low level motion and actuation commands that are sent back to the Server, which in turn communicates them to the robots via the wireless radio.

The modularity of the CMDragons software architecture is beneficial for multiple reasons. First, it allows running each of the individual components on separate computers, which can be extremely valuable during development and testing, as multiple team-members can work on separate work-stations and can remotely connect their Soccer or GUI programs to the server computer. Another advantage is that it allows transparent swapping of individual modules without affecting the rest of the system. For example, it is possible to replace the *Server* module with a *Simulator* module (further explained in Section A.3.4) that provides a semantically identical network interface to the Soccer and GUI programs.

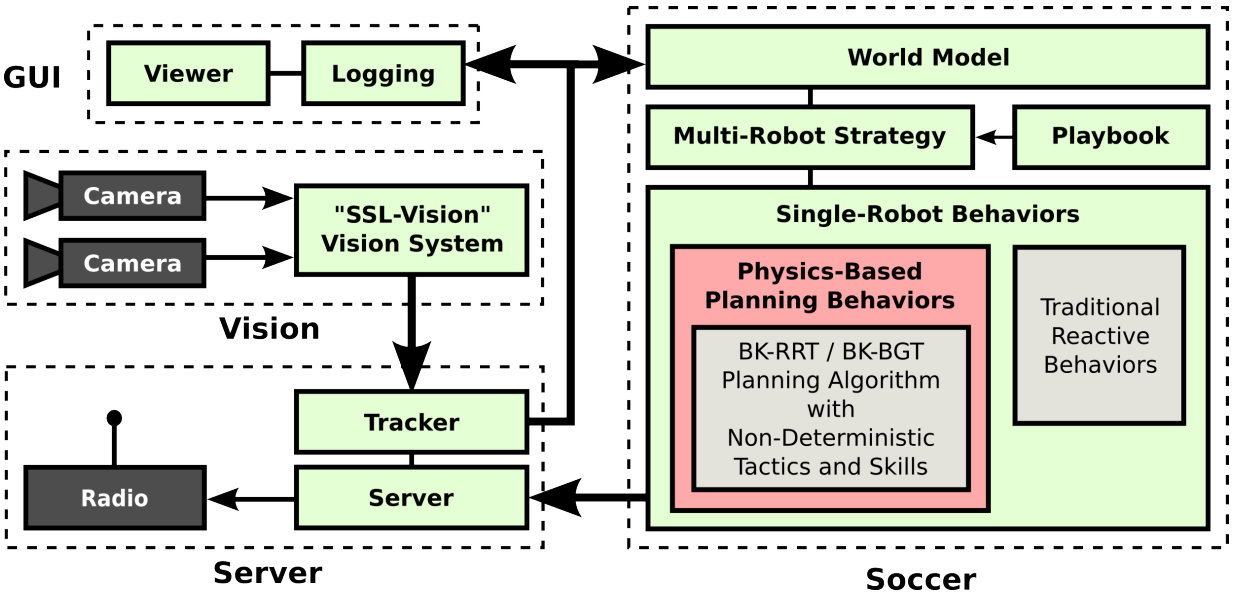


Figure A.3: The CMDragons system infrastructure, including the integration of physics-based planning.

A.3.1 Vision

In past years, each team in the RoboCup Small Size League was allowed to mount their own set of cameras above the field and run their own vision system to perform robot and ball localization. This, however, lead to many logistic and organizational problems during competition, and furthermore required new-coming teams to spend significant effort on "re-inventing" approaches to the already adequately solved vision problem. As a solution, the

Small Size League committees decided to pursue the concept of a shared vision system, i.e., a single set of cameras per field, connected to a single vision server computer, running an official vision software. This lead to the development of *SSL-Vision*, the shared open-source vision system for the RoboCup Small Size League [94]. I was SSL-Vision's lead designer and continue to be one of the project's lead developers and project administrators. Beginning with RoboCup 2010, the use of the shared vision system will be mandatory for all teams, and teams are no longer allowed to mount their own cameras.

The SSL-Vision software architecture and image processing stack is described in Appendix B. Within the CMDragons lab environment, SSL-Vision runs on a desktop computer that contains two separate 1394b host controllers, each connected to a respective AVT Stingray F46 color camera. Each camera is configured to capture at 780x580 resolution in YUV422 at 60Hz. The vision system's output consists of the location and orientation for each robot, as well as the location of the ball, all in real-world field coordinates.

A.3.2 Soccer

The Soccer component relies on the Skills, Tactics, and Plays (STP) [13] model to perform its intelligent decision making. Within the STP architecture, plays are responsible for coordinating multi-robot team strategy, whereas Skills and Tactics compute the actions for individual robots (although Tactics can also coordinate their actions with each other, as we will elaborate on further below).

Plays

The multi-robot strategy layer uses a *Playbook*, consisting of multiple *Plays*, each describing a set of individual "roles" to be distributed among the available robots. The play-system works in a singleton centralized fashion: exactly one play is active at any point in time, assigning the roles for all robots. The playbook itself is stored in a configuration file, allowing modifications at run-time without system recompilation [11].

Plays are selected based on a set of pre-conditions. For example, a "defensive" play will only be assigned if its precondition is met that the opponent currently has ball possession. If multiple plays are simultaneously applicable, then one of them is chosen randomly. To bias this random play selection process, the playbook allows the definition of custom weights for each play.

Once selected, a play will assign the individual single-robot roles to the actual robots. The assignment occurs dynamically: each robot role has a cost function that is able to

heuristically estimate the cost of assignment to a particular robot, given the robot's current physical state in the world. For example, an "attacker" role should preferably be assigned to the robot that is currently closest to the ball. Thus, its cost metric simply returns the distance between the potential robot's current location and the ball's location. The role assignment process seeks global optimality, that is, it selects a mapping from roles to individual robots that will minimize the sum of all role's individual cost functions. Because the team is limited to only 5 robots in a game, the play system can exhaustively compute the cost values for all possible mappings and is therefore guaranteed to find and assign the optimal assignment mapping.

Skills and Tactics

Besides the novel physics-based planning behaviors, such as the *PhysicsDribble* behavior (see Chapter 7), all other single-robot behaviors in the CMDragons system are implemented as traditional Skills and Tactics, as defined by the STP model [13]. Within the STP model, both Skills and Tactics act as single-robot controllers, taking as input the world's state, and producing as output a control for the robot to execute. The difference between Skills and Tactics is that Skills implement low-level control tasks that are invoked by Tactics, which perform higher level single-robot behaviors. Furthermore, only Tactics implement the role assignment cost function described in Section A.3.2 and therefore only Tactics can be directly assigned to robots using the Play system. Examples of Tactics in the CMDragons system are "attacker", "defender", "goalie", and "supporter". Internally, Tactics may contain any kind of control code, but within the CMDragons system, Tactics typically employ a finite state machine to transition between Skills. For example, the "Attacker" Tactic transitions between the Skills "get close to ball", "steal ball", and "kick" (amongst others), where the particular transitions depend on the current world state. Skills are typically treated as fundamental control building-blocks in the CMDragons system, and it is common that a single Skill is used by multiple different Tactics, thus reducing the need for re-implementation of frequent tasks. Furthermore, all Skills and Tactics can rely on a shared library of "helper functions" that are able to perform frequent control and math computations. One particular such shared helper function is the navigation planner of the CMDragons system.

Navigation Planning

The navigation planner in the CMDragons system is based on ERRT [14, 15, 19], and is able to compute paths from a current location to a target location. Unlike our physics-based planning approaches presented in this thesis, the CMDragons ERRT planner does

not simulate dynamics and therefore produces velocity-free paths that can be potentially dangerous to execute directly. To achieve safety, the CMDragons planner uses a post-processing step based on the *dynamic window* method [12, 33] to ensure that the velocities which are sent to the robot will not result in a collision. While such a strategy will not yield time-optimal control, it does guarantee safety and furthermore allows planning at much faster rates than fully dynamic planning. This ERRT planner solves only the problem of navigation, and is completely unaware of multi-body interactions, or of the higher level tactical goals of its calling Skill or Tactic. These properties make it difficult to construct behaviors that need to rely on purposeful ball-manipulation and on navigation planning simultaneously. This limitation was a major source of motivation for the development of the physics-based Tactics-aware planning approaches presented in this thesis. Now, the CMDragons system includes the physics-based planner and the *PhysicsDribble* behavior (see Chapter 7) which is actively used during games.

State Evaluation

Many of the CMDragons Tactics need to constantly make difficult strategic choices from a vast array of possible options. One particularly challenging example is the problem of where to best position a robot, when running a "supporter" behavior. A supporter's role is to assist the robot that currently handles the ball (the "attacker"), by being available for passes, but also by not interfering with the attacker's actions. To make such strategic decisions, many CMDragons Tactics rely on heuristics that are able to evaluate the quality of potential states. The Tactic then computes the heuristic quality for many such potential states and chooses the best one. The actual heuristics typically consist of a weighted set of individual evaluators, that each represent one particular quality metric. In the case of the supporter-Tactic, the evaluation consists of the following factors:

- **Angle Toward Opponent's Goal-Box**
  The supporter robot prefers positions that allow a direct shot onto the opponent's goal, allowing direct deflections from a pass-reception to a goal-shot.

- **Clear Path for Ball Reception**
  The supporter robot prefers positions that create an un-obstructed corridor for the ball to travel from its current position to the supporter's dribbler.

- **Distance From Opponents**
  The supporter robot prefers positions that maximize the distance from opponent robots. This metric is particularly useful to outrun opponent's that attempt to "mark" the supportive robot.

- **Distance Toward Attacker Robot**
  The supporter robot prefers positions that are within good pass reception distance. Being too far from the opponent requires long passes that are less likely to succeed, whereas being too close can interfere with the attacker's actions.

- **Distance From Robot's Current Physical Location**
  The supporter robot prefers positions that are close to the robot's current position, thus minimizing travel time and preventing unneccesary oscillations.

The actual weights for each of these evaluators are currently determined manually via experimental optimization. Optimizing these parameters automatically by using supervised learning techniques is a major future goal for the CMDragons team.

Multi-Robot Coordination

In the CMDragons system, there is a clear distinction between multi-robot role assignment (i.e., Plays), and single-robot control (i.e., Skills and Tactics). However, throughout the course of a robot soccer game, there are many instances where a subset of robots need to jointly coordinate their controls. One example are passes, where the receiving robot needs to be aware of the intended target location of the passing robot's kick, preferably *before* the kick occurs. To coordinate such multi-robot events, the CMDragons system provides the ability for Tactics to share important information about their internal state in a *shared state* data structure that is accessible to all robots. This shared state data structure essentially implements a publish/subscribe communication model between robots. Because each robot has a shared state, all shared states are updated in lock-step. For example, at a particular frame number n, each robot is allowed to read the shared states of all other robots which were set at frame $n-1$. Similarly, each robot is only allowed to modify their shared state information for frame n, while frame n is occurring, and this state will not be made available to the other robots until the next frame $n+1$. Using this lock-step scheme successfully prevents potentially dangerous race-conditions during communication, while only incurring a single frame latency.

A.3.3 GUI

Using the Skills, Tactics, and Plays infrastructure, it is possible to model complex multi-robot behaviors. However, such complexity can also make it challenging to analyze and debug the many components and parameters of the system. To overcome such challenges, CMDragons features a logging infrastructure that is able to record complete sequences

of world states during games. Besides containing the world state (which consists of the ball's and all the robots' state parameters), the system also records the internal state of the Play-selection system and the current hierarchy of selected Tactics. Additionally, each Tactic is able to write further debugging information into the log, thus providing detailed information about the Tactics' internal reasoning at any given point in time.

To make use of such rich data, CMDragons features a *Viewer* program that allows easy browsing through logs. Figure A.4 shows a screenshot of this viewer. The internal state of the Play system and Tactics is shown as a browsable hierarchy of text (on the right). The world state is drawn onto the virtual soccer field, and allows toggling of several visualization options. Furthermore, the viewer can display the current velocity of the robots or balls as a graph (bottom).

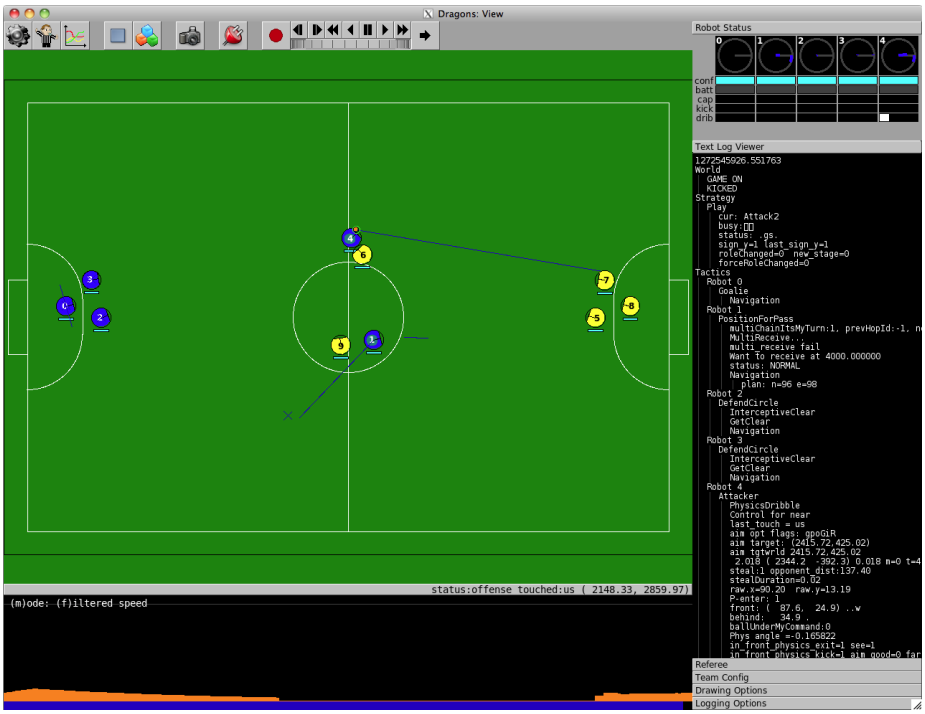


Figure A.4: A screenshot of the CMDragons Viewer.

A.3.4 Simulator

Being able to accurately simulate robot behaviors is an integral part of CMDragon's development cycle. Simulation allows rapid testing of new code without the need to impose drain on the robotic hardware. Additionally, it allows testing of scenarios that cannot be tested with the limited number of physical robots, such as full five-on-five RoboCup games.

For this purpose, CMDragons features a *Simulator* module that can be used instead of the Server module. To the remaining infrastructure (i.e. Soccer and GUI), the Simulator behaves identically to the Server module: it produces state observations that are integrated into the Soccer module's world model, and it receives robot motion commands from the Soccer module that are then executed on the simulated robots. In order to accurately simulate real-world robot interactions, the Simulator module uses the same physics engine and model that we introduced for our physics-based planner. In particular, the robot shape and dribbling model is identical to the one presented in Chapter 7. Figure A.5 shows an example of the simulator's 3D rigid body representation of the soccer domain.

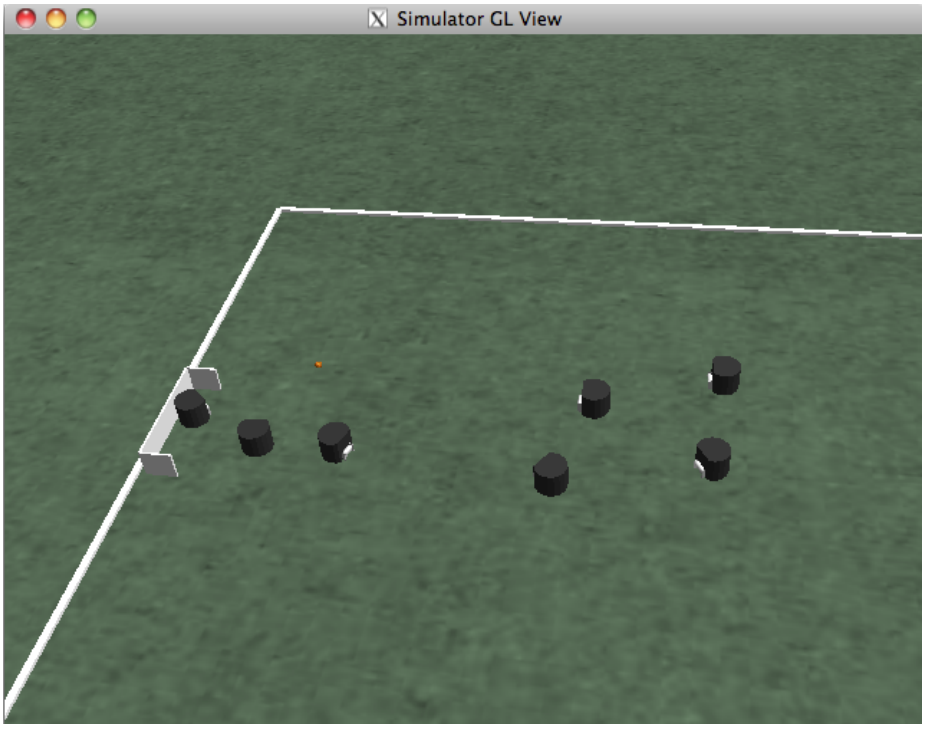


Figure A.5: A visualization of the Simulator's internal state.

A.4 Appendix Summary

In this appendix, we have briefly presented the CMDragons architecture, including the robot hardware, and off-board control software. Members of the past and current CMDragons teams, who we acknowledge, are Joydeep Biswas, James Bruce, Mike Licitra, Gabriel Levi, Philip Wasserman, Michael Bowling, Brett Browning, Sorin Achim, Kwun Han, and Peter Stone. In the following appendix, we further explain the SSL-Vision soft-

ware that CMDragons and other Small Size League Teams rely upon for robot and ball localization.

Appendix B

The SSL-Vision Shared Vision System

In Appendix A, we have outlined the CMDragons RoboCup Small Size League (SSL) robot soccer system. A significant component contributing to CMDragons' performance is its global vision system, delivering accurate localization of the robots and the ball in real-time. This appendix explains the details of "SSL-Vision", the open source vision system for the RoboCup Small Size League that is used by CMDragons and that I significantly contributed to design and develop.

In previous years, the RoboCup Small Size League rules allowed every team to mount their own cameras above or next to the soccer field. To adequately cover the entire soccer field, teams typically use two cameras, one above each half. With up to six teams sharing a soccer field, this configuration results in several organizational problems. Each team requires a significant amount of time to mount and calibrate their cameras. During this calibration time, the field is unusable for actual robot soccer matches or robot testing (see Figure B.1). Furthermore, due to the setup overhead, teams are tied to their fields, and are unable to play testing matches against teams from other fields (a typical RoboCup competition has 4 fields). Finally, letting each team provide their own cameras also prevents scalability of the league, to e.g., a larger field that would require 4 cameras for full coverage.

The requirement that each team provides their own vision system also makes it difficult for new teams to enter the league. Many beginner teams spend significant time and resources on re-implementing and re-inventing solutions to the vision and localization problem which has already been solved with sufficient accuracy and performance by many other teams. In fact, among the strong teams in the Small Size League and even other RoboCup leagues, the state-of-the-art vision approaches seem to have converged to very similar color-based

segmentation methods [4, 16, 40, 64].

Figure B.1: Teams mounting their vision equipment before a RoboCup competition.

To overcome these limitations, the SSL committees decided to migrate to a shared vision system, i.e., to a single set of cameras per field that are connected to an image processing server which is broadcasting localization output to the participating teams. The software for this server needs to be flexible, i.e., scalable for future changes and open to new approaches, as well as competitive, i.e., performant and precise, to not constrain the current performance of the top teams. This system, named SSL-Vision, is developed by a group of volunteers from the SSL, with my role being the project's lead designer, and lead developer. After having been successfully tested on a voluntary basis at RoboCup 2009, SSL-Vision will be the only global sensing platform allowed in the Small Size League at RoboCup 2010.

The remainder of this appendix is organized as follows: Section B.1 describes the overall architecture of the system. We then present the system's approaches for image processing and camera calibration in Section B.2. Finally, we conclude with a summary, and a short discussion of the system's implications for RoboCup and other robotics research in Section B.3.

B.1 Framework

SSL-Vision is intended to be used by all Small Size League teams, with a variety of camera configurations and processing hardware. As such, configurability and robustness are key

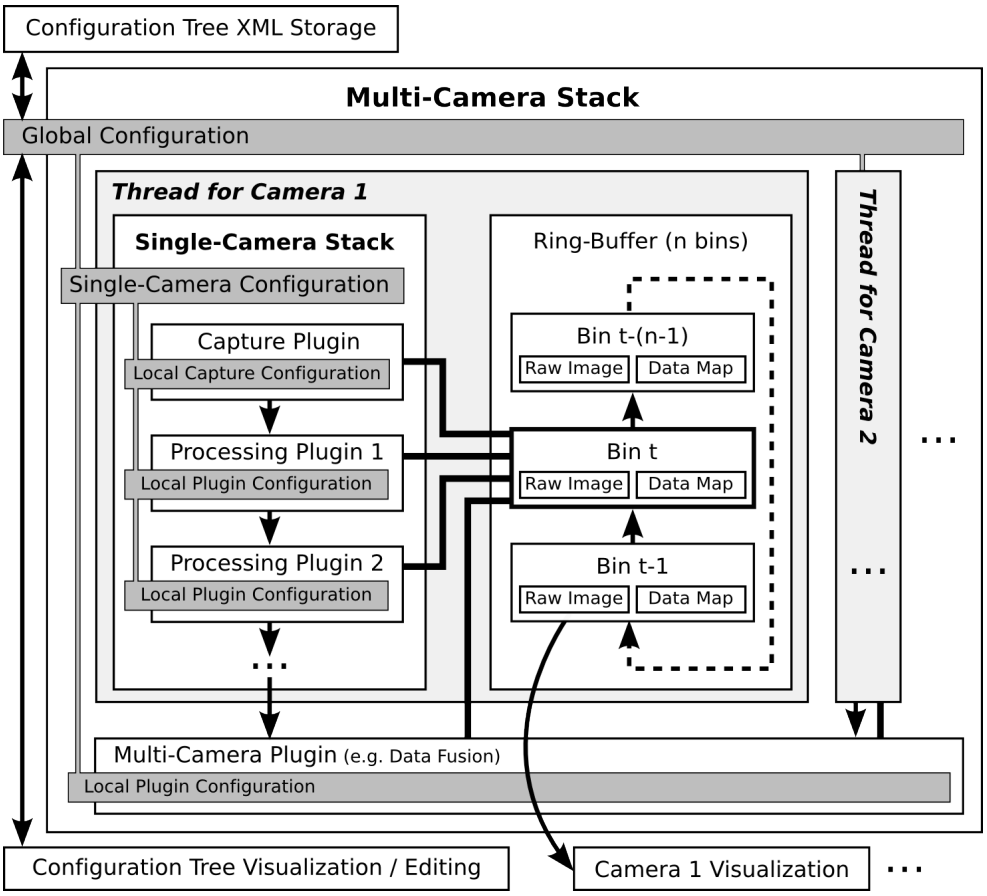


Figure B.2: The extendible, multi-threaded processing architecture of SSL-Vision.

design goals for its framework architecture. Additionally, the project's collaborative openness and its emphasis on research all need to be reflected in its framework architecture through an extendable, manageable, and scalable design.

One major design goal for the framework is to support concurrent image processing of multiple cameras in a single seamless application. Furthermore, the application should integrate all necessary vision functionality, such as configuration, visualization, and actual processing. To achieve better scalability on modern multi-core and hyper-threaded architectures, the application uses a multi-threaded approach. The application's main thread is responsible for the Graphical User Interface (GUI), including all visualizations, and configuration dialogs. Additionally, each individual camera's vision processing is implemented in a separate thread, thus allowing truly parallel multi-camera capture and processing. The application is implemented in C++ and makes heavy use of the Qt toolkit [72], to allow for efficient, platform-independent development.

Fig. B.2 shows an overview of the framework architecture. The entire system's desired processing flow is encoded in a *multi-camera stack* which fully defines how many cameras are used for capturing, and what particular processing should be performed. The system has been designed so that developers can create different stacks for different robotics application scenarios. By default, the system will load a particular multi-camera stack, labeled the "RoboCup Small Size Dual Camera Stack" which we will elaborate on in the following section. However, the key point is that the SSL-Vision framework provides support for choosing any arbitrarily complex, user-defined stack at start-up, and as such becomes very extendible and even attractive for applications that go beyond robot soccer.

Internally, a multi-camera stack consists of several threads, each representing the processing flow of a corresponding capture device. Each thread's capturing and processing flow is modeled as a *single-camera stack*, consisting of multiple *plugins* which are executed in order. The first plugin in any single-camera stack implements the image capturing task. All capture plugins implement the same C++ capture interface, thus allowing true interchangeability and extendibility of capture methods. The framework furthermore supports unique, independent configuration of each single-camera stack, therefore enabling capture in heterogeneous multi-camera setups. Currently, the system features a capture plugin supporting IEEE 1394 / DCAM cameras, including higher bandwidth Firewire 800 / 1394B ones. Configuration and visualization of all standard DCAM parameters (such as white balance, exposure, or shutter speed) is provided through the GUI at run-time, thus eliminating the need for third-party DCAM parameter configuration tools. The system furthermore features another capture plugin supporting capturing from still image and video files, allowing development on machines which do not have actual capture hardware. Additional capture plugins for Gigabit Ethernet (GigE) Vision as well as Video4Linux are under construction as well.

B.1.1 The Capture Loop

A capture plugin produces an output image at some resolution, in some color-space. For further processing, this image data is stored in a *ring-buffer* which is internally organized as a cyclical linked-list where each item represents a *bin*, as is depicted in Fig. B.2. On each capture iteration, the single-camera stack is assigned a bin where it will store the captured image and any additional data resulting from processing this image. As the stack is being executed, each of its plugins is sequentially called, and each of them is able to have full read and write access to the data available in the current bin. Each bin contains a *data map*, which is a hash-map that is able to store arbitrary data under a meaningful label. This data map allows a plugin to "publish" its processing results, thus making them available to be read by any of the succeeding plugins in the stack.

The purpose of the ring-buffer is to allow the application's visualization thread to access the finished processing results while the capture thread is allowed to already move on to the next bin, in order to work on the latest video frame. This architecture has the great advantage of not artificially delaying any image processing for the purpose of visualization. Furthermore, this ring-buffered, multi-threaded approach makes it possible to prioritize the execution schedule of the capture threads over the GUI thread, thus minimizing the impact of visualization on processing latency. Of course it is also possible to completely disable all visualizations in the GUI for maximum processing performance.

In some processing scenarios it is necessary to synchronize the processing results of multiple camera threads after all the single-stack plugins have finished executing. This is done through optional *multi-camera plugins*. A typical example would be a plugin which performs the data fusion of all the threads' object detection results and then sends the fused data out to a network.

B.1.2 Parameter Configuration

Configurability and ease of use are both important goals of the SSL-Vision framework. To achieve this, all configuration parameters of the system are represented in a unified way through a variable management system called *VarTypes* [92]. The VarTypes system allows the organization of parameters of arbitrarily complex types while providing thread-safe read/write access, hierarchical organization, real-time introspection/editing, and XML-based data storage.

Fig. B.2 shows the hierarchical nature of the system's configuration. Each plugin in the SSL-Vision framework is able to carry its own set of configuration parameters. Each single-camera stack unifies these local configurations and may additionally contain some stack-wide configuration parameters. Finally, the multi-camera stack unifies all single-camera stack configurations and furthermore contains all global configuration settings. This entire configuration tree can then be seamlessly stored as XML. More importantly, it is displayed as a data-tree during runtime and allows real-time editing of the data. Fig. B.3 shows a snapshot of the data-tree's visualization.

B.2 RoboCup SSL Image Processing Stack

The system's default multi-camera stack implements a processing flow for solving the vision task encountered in the RoboCup Small Size League. In the Small Size League, teams typically choose a dual-camera overhead vision setup. The robots on the playing

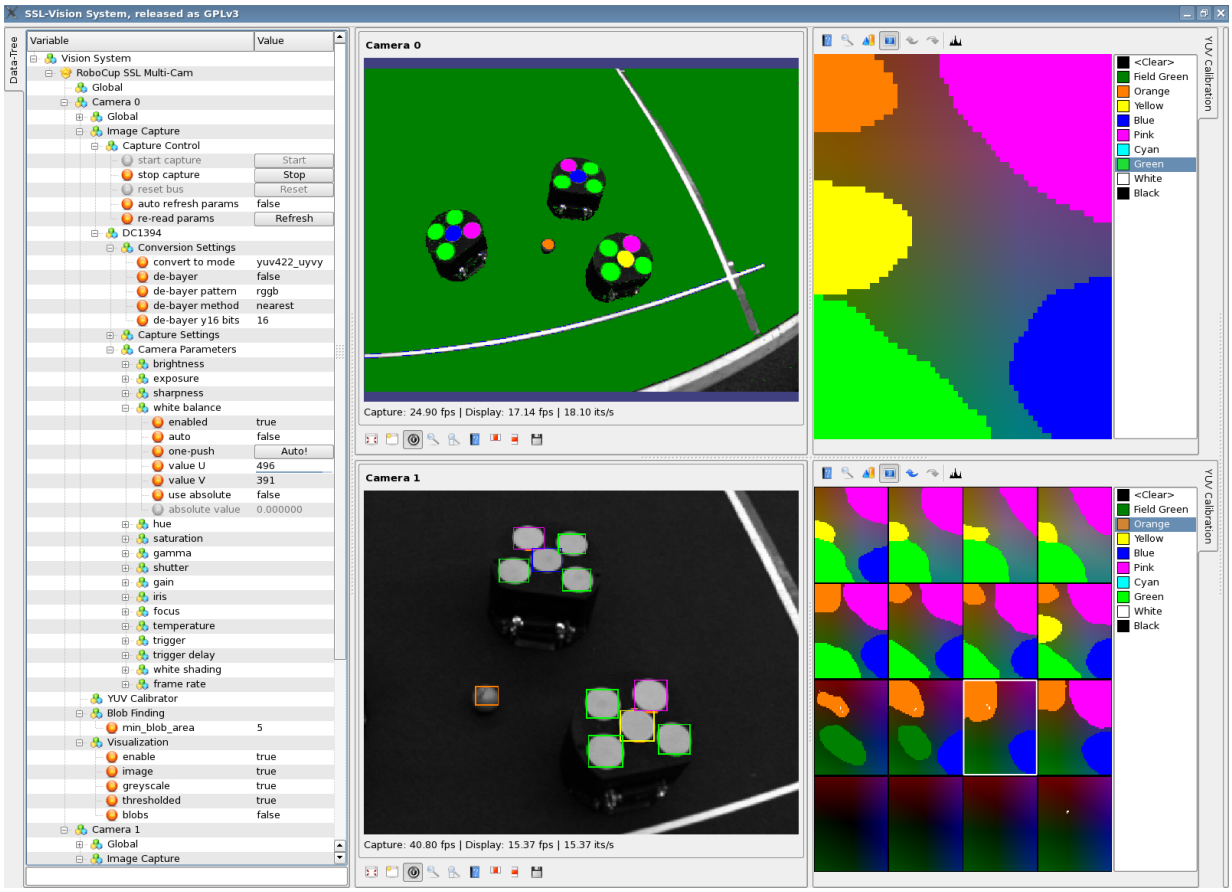


Figure B.3: Screenshot of SSL-Vision, showing the parameter configuration tree (left), live-visualizations of the two cameras (center), and views of their respective color thresholding YUV LUTs (right).

field are uniquely identifiable and locatable based on colored markers. Each robot carries a team-identifying marker in the center as well as a unique arrangement of additional colored markers in order to provide the robot's unique ID and orientation. In the past, each team was able to determine their own arrangement and selection of these additional markers. However, with the introduction of the SSL-Vision system, it is planned to unify the marker layout among all teams for simplification purposes.

The processing stack for this Small Size League domain follows a typical multi-stage approach as it has been proven successful by several teams in the past. The particular single-camera stack consists of the following plugins which we explain in detail in the forthcoming sections: image capture, color thresholding, runlength encoding, region extraction and sorting, conversion from pixel coordinates to real-world coordinates, pattern

detection and filtering, and delivery of detection results via network.

B.2.1 CMVision-Based Color Segmentation

The color segmentation plugins of this stack, namely color thresholding, runlength encoding, region extraction and region sorting, have all been implemented by porting the core algorithms of the existing CMVision library to the new SSL-Vision plugin architecture [17]. To perform the color thresholding, CMVision assumes the existence of a lookup table (LUT) which maps from the input image's 3D color space (by default YUV), to a unique color label which is able to represent any of the marker colors, the ball color, as well as any other desired colors. The color thresholding algorithm then sequentially iterates through all the pixels of the image and uses this LUT to convert each pixel from its original color space to its corresponding color label. To ease the calibration of this LUT, the SSL-Vision system features a fully integrated GUI which is able to not only visualize the 3D LUT through various views, but which also allows to directly pick calibration measurements and histograms from the incoming video stream. Fig. B.3 shows two example renderings of this LUT. After thresholding the image, the next plugin performs a line-by-line runlength encoding on the thresholded image which is used to speed up the region extraction process. The region extraction plugin then uses CMVision's tree-based union find algorithm to traverse the runlength-encoded version of the image and efficiently merge neighboring runs of similar colors. The plugin then computes the bounding boxes and centroids of all merged regions and finally sorts them by color and size.

B.2.2 Camera Calibration

In order to extract meaningful information about the location of objects on the field, SSL-Vision requires a calibration for defining the relationship between the pixel coordinates of the image plane and the actual real-world coordinates on the soccer field. In past years, teams have employed a variety of calibration techniques, often requiring the use of calibration patterns, adding additional logistic constraints to the competition.

SSL-Vision introduces a calibration procedure that does not require any calibration patterns. Instead, the procedure relies solely on the image delivered by the cameras and on the field's known dimensions. Because SSL-Vision uses two independent vision stacks, each camera uses an independent calibration configuration. The SSL-Vision calibration model was developed and implemented by members of the B-Smart SSL team, and the mathematical details of the calibration model are described in [94]. Generally, the calibration method consists of two steps:

1. Using SSL-Vision's real-time GUI, the user selects the four corner points of the half soccer-field in the camera image. These manually selected points act as constraints to construct a rough initial calibration.

2. Using the initial calibration estimate obtained from the manual point selection, the system automatically detects the position of the field lines in the image by using edge detection. The system then uses these extracted locations to further refine its geometric calibration by performing a least-squares fit of the calibration model's parameters, finding the best set of parameters that will consistently map the detected edge locations to their known reference world-coordinates.

Once calibrated, the model is able to convert image coordinates to field coordinates, and vice versa (by using the model's inverse function).

B.2.3 Pattern Detection

After all regions have been extracted from the input image and all their real-world coordinates have been computed, the processing flow continues with the execution of the pattern recognition plugin. The purpose of this plugin is to extract the identities, locations, and orientations of all the robots, as well as the location of the ball. The internal pattern detection algorithm was adopted from the CMDragons vision system and is described in detail in a previous paper [16].

Although this pattern detection algorithm can be configured to detect patterns with arbitrary arrangements of 2D colored markers, the Small Size committees are intending to mandate a standard league-wide pattern layout with the transition to SSL-Vision, for simplification purposes.

B.2.4 System Integration and Performance

After the pattern detection plugin has finished executing, its results are delivered to participating teams via UDP Multicast. Data packets are encoded using Google Protocol Buffers [36], and contain positions, orientations, and confidences of all detected objects, as well as additional meta-data, such as a timestamp and frame-number. Furthermore, SSL-Vision is able to send geometry data (such as camera pose) to clients, if required. To simplify these data delivery tasks, SSL-Vision provides a minimalistic C++ sample client which teams can use to automatically receive and deserialize all the extracted positions and orientations of the robots and the ball. Currently, SSL-Vision does not perform any

Plugin	Time
Image capture	1.1 ms
Color thresholding (CPU)	3.6 ms
Runlength encoding	0.7 ms
Region extraction and sorting	0.2 ms
Coordinate conversion	< 0.1 ms
Pattern detection	< 0.1 ms
Other processing overhead	0.4 ms
Total frame processing	< 6.2 ms

Table B.1: Single frame processing times for the plugins of the default RoboCup stack.

"sensor fusion", and instead will deliver the results from both cameras independently, leaving the fusion task to the individual teams. Similarly, SSL-Vision does not perform any motion tracking or smoothing. This is due to the fact that robot tracking typically assumes knowledge about the actual motion commands sent to the robots, and is therefore best left to the teams.

Table B.1 shows a break-down of processing times required for a single frame of a progressive YUV422 video stream of 780×580 pixel resolution. These numbers represent rounded averages over 12 consecutive frames taken in a randomly configured RoboCup environment, and were obtained on an Athlon 64 X2 4800+ processor.

B.3 Appendix Summary

In this appendix, we have introduced SSL-Vision, the shared vision system for the RoboCup Small Size League and for the CMDragons multi-robot system. We have presented the system's open software architecture and have described its approaches for image processing, camera calibration, and pattern detection. We strongly believe that the system will positively affect the Small Size League by reducing organizational problems and by allowing teams to re-focus their research efforts towards elaborate multi-agent systems and control issues. Because SSL-Vision is a community project, everybody is invited to contribute. Therefore, SSL-Vision's entire codebase is released as open-source [95].

Finally, it should be strongly emphasized that SSL-Vision's architecture is not at all limited to only solving the task of robot soccer vision. Instead, the system should be recognized as a framework which is flexible and versatile enough to be employed for almost any imaginable real-time image processing task. While, by default, the system provides the

stacks and plugins aimed at the RoboCup domain, we are also eagerly anticipating the use and extension of this system for applications which go beyond robot soccer.

Appendix C

Example Skills

This appendix presents the internal control structures and algorithms for several selected Skills that are used as part of the Tactics from the experiments in Chapter 6.

C.1 Sampling-Based Wait Skill

Sampling Distributions D**:**
- Range of wait time (min_wait, max_wait)

Function $f(C, D, V, x, a)$:
  if $t_{\text{exit}} \notin V$ **then**
    $V.t_{\text{exit}} \leftarrow x.t +$ `SampleFromRange(min_wait, max_wait)`;
  if $x.t \geq V.t_{\text{exit}}$ **then**
    busy $\leftarrow$ *false*;
  else
    busy $\leftarrow$ *true*;
  return $\langle a, V,$ busy$\rangle$;

Algorithm 19: "Wait Sampled Time" Skill

The "Sampling-Based Wait Skill" (see Algorithm 19) is used in the minigolf domain where the controlled body is required to wait for a sampling-based amount of time before starting to move. This Skill is relatively simple: during its first call, it samples a waiting period from some predefined sampling range and stores the time when it should stop waiting t_{exit} to V. On each call it then returns busy = true as long as t_{exit} has not been reached yet in

the world state's time $x.t$. Remember, that setting busy = true will prevent any outgoing state-transition from occurring in its parent Tactic. Once t_{exit} has been reached, it will return busy = false, thus allowing a state-transition to another Skill to occur.

C.2 Sampling-Based Kick Skill

The "Sampling-Based Kick Skill" is used throughout the robot soccer domain and also for "putting" in the robot minigolf domain. Its purpose is to sample a kick-target and a kick-strength from predefined distributions, and produce a control action that performs such a kick. Algorithm 20 shows how these actions are computed. Essentially, the algorithm drives the controlled body towards the ball, keeping it ball_dist away from it (in our case, ball_dist is the sum of the robot's radius and the ball's radius). The Skill marks itself as busy until the ball has actually been kicked.

C.3 Biased Fall Down Skill

The "Biased Fall Down Skill" is used in the many-dice domain. Its purpose is to bias the motion of a freely falling body towards a particular predefined target location. Algorithm 21 shows how the Skill achieves this. On first execution, it samples the strength of bias toward its target location. It then generates the appropriate forces to be applied to the body. The Skill marks itself as busy until a collision with another rigid body occurs.

Constants C:

- Index of controlled rigid body: i
- Index of ball rigid body: j
- Distance of player to ball for kicking: ball_dist

Sampling Distributions D:

- Kick force range: min_kickforce, max_kickforce
- Bounding box of kick target: goal_bbox

Function $f(C, D, V, x, a)$:

```
if kickforce, kicktarget ∉ V then
    V.kickforce ← SampleFromRange(min_kickforce, max_kickforce);
    V.kicktarget ← SampleFromBBox(goal_bbox);
aim_heading ← V.kicktarget − x.r̂_j.α;
aim_heading ← aim_heading / ‖aim_heading‖;
aim_location ← x.r̂_j.α − (ball_dist aim_heading);
// Trapezoidal motion control functions [15] compute the necessary
   forces/torques to turn/drive towards the aim heading/location:
a.â_i.torque ← TurnToHeading(x.r̂_i, aim_heading);
a.â_i.force ← DriveToLocation(x.r̂_i, aim_location);
if BallIsInFront(x.r̂_i, x.r̂_j,ball_dist) and ReachedHeading(x.r̂_i, aim_heading)
and ReachedLocation(x.r̂_i, aim_location) then
    // Apply kick to ball:
    a.â_j.force ← V.kickforce aim_heading;
    busy ← false;
else
    busy ← true;
return ⟨a, V, busy⟩;
```

Algorithm 20: "Sampled Kick" Skill

Constants C**:**

- Index of controlled rigid body: i
- Body Parameters: $\langle \bar{r}_1, \ldots, \bar{r}_n \rangle$
- Target location: target

Sampling Distributions D**:**

- Acceleration bias range: min_bias, max_bias

Function $f(C, D, V, x, a)$:

if bias $\notin V$ **then**
  V.bias $\leftarrow$ `RndValFromRange(`min_bias, max_bias`)`;

`// The following assumes that the y-axis is the axis of gravity.`
`Furthermore, gravity is implicitly added by the physics engine`
`after all Skills have returned their actions.`

Δtarget $\leftarrow$ target $- x.\hat{r}_i.\alpha$;
$a.\hat{a}_i.force \leftarrow \bar{r}_i.\phi_{\text{Mass}}\ V$.bias $[\Delta$target.x, 0, Δtarget.z$]$;
$a.\hat{a}_i.torque \leftarrow [0, 0, 0]$;
`// Check if a collision is occurring between` r_i `and any other`
`bodies:`
if `CollisionOccurring(`$i, x, \langle \bar{r}_1, \ldots, \bar{r}_n, \rangle$`)` **then**
  busy $\leftarrow$ *false*;
else
  busy $\leftarrow$ *true*;
return $\langle a, V,$ busy$\rangle$;

Algorithm 21: "Biased Fall Down" Skill

Appendix D

List of Notation

α	Position	12
β	Orientation	12
γ	Linear velocity	12
λ	Collision body index	15
μ	BGT growth control parameter	58
Π	Tactic transition probability function	40
σ	Index of currently active Skill	40
σ'	Index of successor Skill	40
σ_{init}	Index of initial Skill	40
τ	Tactic	40
Φ	Set of parameters	110
ϕ	Body parameter	13
ω	Angular velocity	12
A	Action space	14
a	Action	14
$\hat{a}$	Sub-action	14
B	Body classification set	19
b	Busy flag	36
$b_\wedge$	Global busy flag	45
C	Set of constants	36
D	Set of sampling distributions	36
d	Domain	14
e	Physics engine's simulation function	15
f	Skill operator function	36
G	Gravity force vector	14
g	Tactic transition function	40

h	Number of Tactics	45
k	Number of Skills	40
L	List of collisions	15
l	Collision	15
M	Collision matrix	124
m	Collision matrix entry	124
n	Number of rigid bodies	13
p	Probability	40
r	Rigid body	12
$\hat{r}$	Mutable body state	12
$\bar{r}$	Immutable body parameters	13
S	Set of Skills	40
s	Skill	40
t	Time	13
Δt	State transition time-step	15
$\bar{t}$	Internal simulation time-step	16
t_{LOD}	LOD horizon	122
t_{replan}	Replanning interval	123
V	Set of variables	36
X	State space	13
X_{goal}	Set of goal states	16
x	State	13
x'	Successor state	15
x_{init}	Initial state	16
x_{goal}	Goal state	11
Y	RRT sampling space	49
y	RRT random sample	49
z	Number of planning iterations	49

Appendix E

List of Videos

This appendix provides a list of supplemental videos that demonstrate the application of our physics-based planning algorithms. The videos are available to download and view at: `http://szickler.net/thesis/`.

- **Examples of Tactics-Driven Physics-Based Planning in Multiple Challenging Domains**
  This video contains planning examples from the simulated robot minigolf, robot soccer, pool table, and many-dice domains (see Chapter 6).
- **Comparison of Physics-Based Planning and Traditional Reactive Control in a One-vs-One Robot Soccer Domain**
  This video contains example dribbling sequences from the one-vs-one robot soccer experiment described in Chapter 7.
- **Dribbling via Physics-Based Planning at RoboCup 2009**
  This video shows a goal scored by our team CMDragons at RoboCup 2009 using the PhysicsDribble behavior described in Chapter 7.
- **Autonomous Physics-Based Planning in a Freely Configurable Robot Minigolf Domain**
  This video demonstrates the application of physics-based planning in the robot minigolf domain (see Chapters 7 and 8).
- **Physics-Based Planning Examples in the Robot Minigolf Domain**
  This video shows additional challenging planning examples from the robot minigolf domain (see Chapters 7 and 8).

Bibliography

[1] E.W. Aboaf, S.M. Drucker, and C.G. Atkeson. Task-level robot learning: Juggling a tennis ball more accurately. In *Proceedings of IEEE Interational Conference on Robotics and Automation*, pages 14–19, 1989. 10.5

[2] C.H. An, C.G. Atkeson, and J.M. Hollerbach. *Model-based control of a robot manipulator*. MIT press Cambridge, MA, 1988. 10.5

[3] T. Bäck and H.P. Schwefel. An overview of evolutionary algorithms for parameter optimization. *Evolutionary computation*, 1(1):1–23, 1993. 8.1, 10.8

[4] T. Bandlow, M. Klupsch, R. Hanek, and T. Schmitt. Fast image segmentation, object recognition and localization in a RoboCup scenario. *RoboCup-99: Robot Soccer World Cup III*, pages 111–128, 2000. B

[5] D. Baraff. Analytical methods for dynamic simulation of non-penetrating rigid bodies. *ACM SIGGRAPH Computer Graphics*, 23(3):223–232, 1989. 2.1

[6] D. Baraff. Physically Based Modeling: Rigid Body Simulation. *SIGGRAPH Course Notes, ACM SIGGRAPH*, 2001. 2.1, 2.1.3

[7] R. Barzel, J.F. Hughes, and D.N. Wood. Plausible motion simulation for computer graphics animation. *Computer Animation and Simulation*, pages 184–197, 1996. 10.4

[8] S. Behnke and R. Rojas. A Hierarchy of Reactive Behaviors Handles Complexity. *Balancing Reactivity and Social Deliberation in Multi-Agent Systems: From Robocup to Real-World Applications*, 2001. 4.1, 10.6

[9] K.E. Bekris and L.E. Kavraki. Greedy but safe replanning under kinodynamic constraints. In *IEEE Int. Conference on Robotics and Automation (ICRA)*, pages 704–710, 2007. 10.3

[10] R. Bohlin and L. Kavraki. Path planning using lazy PRM. *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, 1, 2000. 10.2

[11] M. Bowling, B. Browning, and M. Veloso. Plays as effective multiagent plans enabling opponent-adaptive play selection. In *Proceedings of International Conference on Automated Planning and Scheduling (ICAPS04)*, volume 1, 2004. A.3.2

[12] O. Brock and O. Khatib. High-speed navigation using the global dynamic window approach. In *1999 IEEE International Conference on Robotics and Automation, 1999. Proceedings*, volume 1, 1999. 10.3, A.3.2

[13] B. Browning, J. Bruce, M. Bowling, and M. Veloso. Stp: Skills, tactics and plays for multi-robot control in adversarial environments. *IEEE Journal of Control and Systems Engineering*, 219:33–52, 2005. 4.1, 7.1.1, 10.6, A.3.2, A.3.2

[14] J. Bruce and M. Veloso. Real-time randomized path planning for robot navigation. In *Proceedings of IROS-2002*, Switzerland, October 2002. 10.3, 10.6, A.3.2

[15] J. Bruce and M. Veloso. Safe Multi-Robot Navigation within Dynamics Constraints. *Proceedings of the IEEE, Special Issue on Multi-Robot Systems*, 2006. 2, 10.2, 10.3, A.3.2, 20

[16] J. Bruce and M. Veloso. Fast and accurate vision-based pattern detection and identification. In *Proceedings of the 2003 IEEE International Conference on Robotics and Automation (ICRA '03)*, 2003. B, B.2.3

[17] J. Bruce, T. Balch, and M. Veloso. Fast and inexpensive color image segmentation for interactive robots. In *Proceedings of the 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS '00)*, volume 3, pages 2061–2066, 2000. B.2.1

[18] J. Bruce, S. Zickler, M. Licitra, and M. Veloso. CMDragons 2007 Team Description. Technical report, Tech Report CMU-CS-07-173, Carnegie Mellon University, School of Computer Science, 2007. 4.1, 7.1, A

[19] J.R. Bruce. *Real-Time Motion Planning and Safe Navigation in Dynamic Multi-Robot Environments*. PhD thesis, Carnegie Mellon, Computer Science Department, 2006. A.3.2

[20] J. Chai and J.K. Hodgins. Constraint-based motion optimization using a statistical dynamic model. In *ACM SIGGRAPH 2007*, page 8. ACM, 2007. 10.4

[21] S. Chakravorty and R. Saha. Hierarchical motion planning under uncertainty. In *Decision and Control, 2007 46th IEEE Conference on*, pages 3667–3672, 2007. 10.3

[22] S. Chenney and D. Forsyth. Sampling plausible solutions to multi-body constraint problems. *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 219–228, 2000. 10.4

[23] S. Chernova and M. Veloso. An evolutionary approach to gait learning for four-legged robots. In *IEEE/RSJ International Conference on Intelligent Robots and Systems, 2004. (IROS 2004)*, volume 3, pages 2562–2567, 2004. 10.8

[24] NVIDIA Corporation. NVIDIA PhysX. Computer Software. Available at `http://developer.nvidia.com/object/physx.html`, 2010. 2.1.3, 2.2.1, 6.1

[25] R. D'Andrea. The Cornell RoboCup Robot Soccer Team: 1999-2003. *New York, NY: Birkhauser Boston, Inc, 2005.*, pages 793–804, 2005. 4.1, 10.6

[26] A. D'Angelo, E. Menegatti, and E. Pagello. How a cooperative behavior can emerge from a robot team. *Distributed Autonomous Robotic Systems 6*, pages 75–84, 2007. 10.6

[27] T. Dean and M. Boddy. An analysis of time-dependent planning. In *Proceedings of the seventh national conference on artificial intelligence*, pages 49–54, 1988. 7.2.2

[28] R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality af A*. *Journal of the ACM (JACM)*, 32(3):505–536, 1985. 10.1

[29] E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. 10.1

[30] B. Donald, P. Xavier, J. Canny, and J. Reif. Kinodynamic motion planning. *Journal of the ACM (JACM)*, 40(5):1048–1066, 1993. 2.1

[31] P. Faloutsos, M. van de Panne, and D. Terzopoulos. Composable controllers for physics-based character animation. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 251–260. ACM New York, NY, USA, 2001. 10.4

[32] D. Ferguson, N. Kalra, and A. Stentz. Replanning with RRTs. *Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on*, pages 1243–1248, 2006. 10.3

[33] D. Fox, W. Burgard, and S. Thrun. The dynamic window approach to collision avoidance. *IEEE Robotics & Automation Magazine*, 4(1):23–33, 1997. 10.3, A.3.2

[34] T. Fraichard and H. Asama. Inevitable collision states. A step towards safer robots? In *Intelligent Robots and Systems, 2003.(IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*, volume 1, 2003. 10.3

[35] B.P. Gerkey and M.J. Mataric. A formal analysis and taxonomy of task allocation in multi-robot systems. *The International Journal of Robotics Research*, 23(9):939, 2004. 10.6

[36] Google Inc. Protocol Buffers. Computer Software. Available at `http://code.google.com/p/protobuf/`, 2009. B.2.4

[37] E. Horvitz. Reasoning about beliefs and actions under computational resource constraints. *Int. J. Approx. Reasoning*, 2(3):337–338, 1988. 7.2.2

[38] P. Isto. Constructing probabilistic roadmaps with powerful local planning and path optimization. In *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 2323–2328, 2002. 10.2

[39] N. Jabson, K. Leong, S. Licarte, G. Oblepias, E. Palomado, and E. Dadios. The autonomous golf playing micro robot: With global vision and fuzzy logic controller. *International Journal on Smart Sensing and Intelligent Systems*, 1(4):824–841, 12 2008. 10.7, 11.1

[40] M. Jamzad, BS Sadjad, VS Mirrokni, M. Kazemi, H. Chitsaz, A. Heydarnoori, MT Hajiaghai, and E. Chiniforooshan. A fast vision system for middle size robots in robocup. *Robocup 2001: Robot soccer world cup V*, pages 159–203, 2002. B

[41] Y.B. Jia, M. Mason, and M. Erdmann. A State Transition Diagram for Simultaneous Collisions with Application in Billiard Shooting. *Algorithmic Foundation of Robotics VIII*, pages 135–150, 2008. 10.5

[42] M. Jouaneh and P. Carnevale. Watch out, Tiger Woods! [golf playing robot]. *IEEE Robotics & Automation Magazine*, 10(2):56–60, 2003. 10.7, 11.1

[43] T. Kalmár-Nagy, R. D'Andrea, and P. Ganguly. Near-optimal dynamic trajectory generation and control of an omnidirectional vehicle. *Robotics and Autonomous Systems*, 46(1):47–64, 2004. 10.3

[44] L. Kavraki, P. Svestka, J. Latombe, and M. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *Robotics and Automation, IEEE Transactions on*, 12(4):566–580, 1996. 10.2

[45] R. Knepper, S. Srinivasa, and M. Mason. Hierarchical planning architectures for mobile manipulation tasks in indoor environments. In *Proceedings of ICRA 2010*, May 2010. 10.3

[46] J. Kuffner. Effective sampling and distance metrics for 3D rigid body path planning. In *IEEE International Conference on Robotics and Automation*, volume 4, pages 3993–3998, 2004. 10.2

[47] J. Kuffner and S. LaValle. RRT-connect: An efficient approach to single-query path planning. *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, 2, 2000. 5.5.1, 10.2, 10.5

[48] A.M. Ladd and L.E. Kavraki. Motion planning in the presence of drift, underactuation and discrete system changes. *Proceedings of Robotics: Science and Systems*, 2005. 10.2

[49] J.C. Lagarias, J.A. Reeds, M.H. Wright, and P.E. Wright. Convergence properties of the Nelder-Mead simplex method in low dimensions. *SIAM Journal on Optimization*, 9(1):112–147, 1999. 8.1, 10.8

[50] J.E. Laird. It knows what you're going to do: adding anticipation to a Quakebot. In *Proceedings of the fifth international conference on Autonomous agents*, pages 385–392. ACM New York, NY, USA, 2001. 10.6

[51] J.E. Laird. Using a Computer Game to Develop Advanced AI. *COMPUTER*, pages 70–75, 2001. 10.6

[52] J.E. Laird, A. Newell, and P.S. Rosenbloom. SOAR: an architecture for general intelligence. *Artificial Intelligence*, 33(1):1–64, 1987. 10.6

[53] M. Lau and J.J. Kuffner. Behavior planning for character animation. *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 271–280, 2005. 10.4

[54] M. Lau and J.J. Kuffner. Precomputed search trees: planning for interactive goal-driven animation. *Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 299–308, 2006. 10.4

[55] T. Laue and M. Hebbel. Automatic Parameter Optimization for a Dynamic Robot Simulation. In *RoboCup 2008: Robot Soccer World Cup XII*, volume 5399, pages 121–132. Springer, 2009. 10.8

[56] T. Laue and T. Rofer. A Behavior Architecture for Autonomous Mobile Robots Based on Potential Fields. *8th International Workshop on RoboCup*, 2004. 4.1, 10.6

[57] S.M. LaValle. Rapidly-exploring random trees: A new tool for path planning. *Computer Science Dept, Iowa State University, Tech. Rep. TR*, pages 98–11, 1998. 5.3, 5.3.3, 5.4, 6.1.1, 10.2

[58] S.M. LaValle. *Planning algorithms*. Cambridge Univ Pr, 2006. 5.5.1, 10.1, 10.2

[59] S.M. LaValle and M.S. Branicky. On the relationship between classical grid search and probabilistic roadmaps. *Algorithmic Foundations of Robotics V*, pages 59–76, 2003. 10.2

[60] S.M. LaValle and J.J. Kuffner Jr. Randomized Kinodynamic Planning. *The International Journal of Robotics Research*, 20(5):378, 2001. 5.3.3, 10.2

[61] W. Leckie and M. Greenspan. An event-based pool physics simulator. *Advances in Computer Games*, pages 247–262, 2005. 10.5

[62] S.R. Lindemann and S.M. LaValle. Current issues in sampling-based motion planning. *Robotics Research*, pages 36–54, 2005. 10.2

[63] K.M. Lynch and M.T. Mason. Stable pushing: Mechanics, controllability, and planning. *The International Journal of Robotics Research*, 15(6):533, 1996. 10.5

[64] L.A. Martinez-Gomez and A. Weitzenfeld. Real Time Vision System for a Small Size League Team. In *Proc. 1st IEEE-RAS Latin American Robotics Symposium, ITAM, Mexico City, October*, pages 28–29, 2004. B

[65] M.T. Mason. *Mechanics of robotic manipulation*. The MIT Press, 2001. 10.5

[66] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998. 6.1

[67] N.A. Melchior, J. Kwak, and R. Simmons. Particle RRT for Path Planning in very rough terrain. In *NASA Science Technology Conference 2007 (NSTC 2007)*, 2007. 10.2

[68] D.J. Montana. The kinematics of contact and grasp. *The International Journal of Robotics Research*, 7(3):17, 1988. 10.5

[69] W. Moss, M. Lin, and D. Manocha. Constraint-based Motion Synthesis for Deformable Models. *Computer Animation and Virtual Worlds*, 19:421–431, July 2008. 10.4

[70] S.R. Munasinghe, Ju-Jang Lee, T. Usui, M. Nakamura, and N. Egashira. Telerobotic mini-golf: system design for enhanced teleoperator performance. In *Robotics and Automation, 2004. Proceedings. ICRA '04. 2004 IEEE International Conference on*, volume 2, pages 1688–1693, 5 2004. 10.7, 11.1

[71] J. Nelder and R. Mead. A simplex method for function minimization. *The computer journal*, 7(4):308, 1965. 8.1, 10.8

[72] Nokia Corporation. The Qt Toolkit. Computer Software. Available at `http://www.qtsoftware.com/`, 2009. B.1

[73] A.M. Okamura, N. Smaby, and M.R. Cutkosky. An overview of dexterous manipulation. In *IEEE International Conference on Robotics and Automation*, volume 1, pages 255–262. IEEE; 1999, 2000. 10.5

[74] LE Parker. ALLIANCE: an architecture for fault tolerant multirobot cooperation. *Robotics and Automation, IEEE Transactions on*, 14(2):220–240, 1998. 10.6

[75] M. Pivtoraiko and A. Kelly. Differentially constrained motion replanning using state lattices with graduated fidelity. In *IEEE/RSJ International Conference on Intelligent Robots and Systems, 2008. IROS 2008*, pages 2611–2616, 2008. 10.3

[76] J. Popović, S.M. Seitz, M. Erdmann, Z. Popović, and A. Witkin. Interactive manipulation of rigid body simulations. *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 209–217, 2000. 10.4

[77] RoboCup Small Size League. Official Web Site. `http://small-size.informatik.uni-bremen.de`, 2009. 3.1.2

[78] A. Safonova and J.K. Hodgins. Construction and optimal search of interpolated motion graphs. In *ACM SIGGRAPH 2007*, page 106. ACM, 2007. 10.4

[79] A. Shapiro, Ma. Kallmann, and P. Faloutsos. Interactive motion correction and object manipulation. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 137–144, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-628-8. doi: http://doi.acm.org/10.1145/1230100.1230124. 10.4

[80] T. Simeon, J.P. Laumond, J. Cortes, and A. Sahbani. Manipulation planning with probabilistic roadmaps. *The International Journal of Robotics Research*, 23(7-8):729, 2004. 10.5

[81] R. Smith et al. Open Dynamics Engine. Computer Software. Available at `http://www.ode.org`, 2008. 2.1.3

[82] A. Stentz. Optimal and efficient path planning for unknown and dynamic environments. Technical Report CMU-RI-TR-93-20, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, August 1993. 10.1

[83] M. Stilman, J.U. Schamburek, J. Kuffner, and T. Asfour. Manipulation planning among movable obstacles. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 3327–3332. Citeseer, 2007. 10.5

[84] A. Treuille, Y. Lee, and Z. Popović. Near-optimal character animation with continuous control. In *ACM SIGGRAPH 2007*, page 7. ACM, 2007. 10.4

[85] C.D. Twigg and D.L. James. Many-worlds browsing for control of multibody dynamics. *ACM Transactions on Graphics (TOG)*, 26(3), 2007. 5.5.2, 10.4

[86] N. Vahrenkamp, C. Scheurer, T. Asfour, J. Kuffner, and R. Dillmann. Adaptive motion planning for humanoid robots. In *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, pages 2127–2132, 2008. 10.2

[87] M. Veloso, P. Stone, and K. Han. The CMUnited-97 robotic soccer team: Perception and multiagent control. In *Proceedings of the second international conference on Autonomous agents*, pages 78–85, 1998. 7.1, A

[88] L.C. Wang, L.S. Yong, and M.H. Ang Jr. Hybrid of global path planning and local navigation implemented on a mobile robot in indoor environment. In *Proceedings of the IEEE International Symposium on Intelligent Control*, pages 821–826, 2002. 10.3

[89] G. Welch and G. Bishop. An introduction to the Kalman filter. *University of North Carolina at Chapel Hill, Chapel Hill, NC*, 1995. 7.1.1, A.3

[90] K. Yamane, J.J. Kuffner, and J.K. Hodgins. Synthesizing animations of human manipulation tasks. *ACM Transactions on Graphics (TOG)*, 23(3):532–539, 2004. 10.4

[91] Y. Yang and O. Brock. Elastic roadmaps: Globally task-consistent motion for autonomous mobile manipulation in dynamic environments. *Proceedings of Robotics: Science and Systems II*, 2006. 10.2, 10.3

[92] S. Zickler. The VarTypes System. Computer Software. Available at `http://code.google.com/p/vartypes/`, 2009. B.1.2

[93] S. Zickler and M. Veloso. Efficient physics-based planning: sampling search via non-deterministic tactics and skills. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 1*, pages 27–33, 2009. 2.1

[94] S. Zickler, T. Laue, O. Birbach, M. Wongphati, and M. Veloso. SSL-Vision: The Shared Vision System for the RoboCup Small Size League. *RoboCup 2009: Robot Soccer World Cup XIII*, pages 425–436, 2009. A.3.1, B.2.2

[95] S. Zickler, T. Laue, et al. SSL-Vision. Computer Software. Available at `http://code.google.com/p/ssl-vision/`, 2009. B.3

[96] M. Zucker, J. Kuffner, and M. Branicky. Multipartite rrts for rapid replanning in dynamic environments. *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 1603–1609, 2007. 10.3

[97] M. Zucker, J. Kuffner, and J.A. Bagnell. Adaptive workspace biasing for sampling-based planners. In *IEEE International Conference on Robotics and Automation. New York: IEEE Press*, pages 3757–3762, 2008. 10.2