Space-Time Planning with Parameterized Locomotion Controllers

SERGEY LEVINE
Stanford University
YONGJOON LEE
University of Washington
VLADLEN KOLTUN
Stanford University
and
ZORAN POPOVIĆ
University of Washington

We present a technique for efficiently synthesizing animations for characters traversing complex dynamic environments. Our method uses parameterized locomotion controllers that correspond to specific motion skills, such as jumping or obstacle avoidance. The controllers are created from motion capture data with reinforcement learning. A space-time planner determines the sequence in which controllers must be executed to reach a goal location, and admits a variety of cost functions to produce paths that exhibit different behaviors. By planning in space and time, the planner can discover paths through dynamically changing environments, even if no path exists in any static snapshot. By using parameterized controllers able to handle navigational tasks, the planner can operate efficiently at a high level, leading to interactive replanning rates.

Categories and Subject Descriptors: I.3.6 [**Computer Graphics**]: Three-Dimensional Graphics and Realism—*Animation*

General Terms: Algorithms

Additional Key Words and Phrases: Human animation, data-driven animation, optimal control, motion planning

ACM Reference Format:

Levine, S., Lee, Y., Koltun, V., and Popović, Z. 2011. Space-time planning with parameterized locomotion controllers. ACM Trans. Graph. 30, 3, Article 23 (May 2011), 11 pages.
DOI = 10.1145/1966394.1966402
http://doi.acm.org/10.1145/1966394.1966402

1. INTRODUCTION

Navigation through complex dynamic environments is a common problem in games and virtual worlds, and poor decision-making on the part of computer-controlled agents (such as nonplayer characters) is a frequent source of frustration for users. In recent years, a number of methods have been proposed that address the problem of path planning and animation in large environments [Choi et al. 2003; Sung et al. 2005; Lau and Kuffner 2006; Sud et al. 2007]. Yet no method has been proposed that can efficiently and gracefully handle the highly dynamic scenes that are common in games, and no animation method has been proposed that takes future obstacle motion into account during global planning. On the other hand, considerable progress has been made in recent years on constructing optimal and near-optimal kinematic controllers from motion capture data [Treuille et al. 2007; McCann and Pollard 2007; Lo and Zwicker 2008; Lee et al. 2009]. These controllers sequence motion clips to produce high-quality animations, but are limited to simple environments described with a small set of parameters.

We present a method that combines path planning in space and time with parameterized controllers to produce graceful animations for characters traversing highly dynamic environments. Our planning algorithm selects controllers that, when concatenated in sequence, generate an animation for a character traversing the dynamically changing environment. Intuitively, the controllers represent intelligent motion skills, such as obstacle avoidance or jumping, which are composed to produce complex paths. The controller library is modular, and controllers can be added or removed to yield characters that possess a wider or narrower variety of motion skills.

The use of parameterized controllers has three key advantages. First, since controllers are high-level constructs that can negotiate

This work was supported in part by NSF grant CCF-0641402, an NSF Graduate Research Fellowship, the UW Animation Research Labs, the UW Center for Game Science, Microsoft, Intel, Adobe, and Pixar.
Authors' addresses: S. Levine, Department of Computer Science, Stanford University, Stanford, CA; email: svlevine@cs.stanford.edu; Y. Lee, Department of Computer Science, University of Washington, Seattle, WA; email: yongjoon@cs.washington.edu; V. Koltun, Department of Computer Science, Stanford University, Stanford, CA; email: vladlen@cs.stanford.edu; Z. Popović, Department of Computer Science, University of Washington, Seattle, WA; email: zoran@cs.washington.edu.

DOI 10.1145/1966394.1966402
http://doi.acm.org/10.1145/1966394.1966402

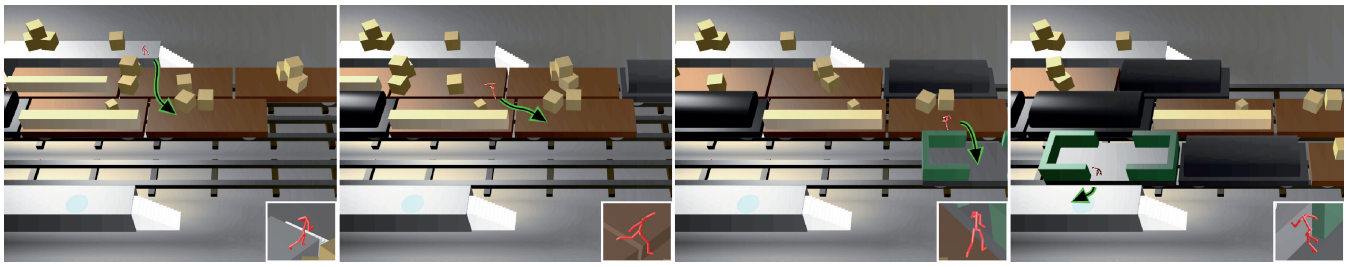


Fig. 1. Our technique animates characters traversing complex dynamic environments. Here, the character jumps on moving railroad cars to reach the opposite platform.

local obstacles, the global planner need not consider as many environmental details as standard sampling-based methods, allowing for efficient planning in large environments. Second, the parameterized controllers can synthesize finely tuned animations that would not be possible with a rigid graph structure. Finally, since the controllers expose only their low-dimensional task parameters to the space-time planner, the time complexity of the planner is independent of the number of motion clips used, in contrast to planners that operate directly on motion graphs. Because of this, the controllers can use a rich database of parameterized motion clips to construct fluid, graceful animations without adversely impacting performance. In order to fit the low-dimensional task parameters of the controllers to an arbitrary polygonal environment, we employ a parameter adaptation algorithm that samples relevant points in the world and fits the controller's internal obstacle representation to them.

The planner assembles the sequence of controllers using a space-time planning algorithm. The space-time planning formulation enables the method to handle even highly dynamic scenes, in which the character must not only avoid collisions with moving objects, but can also use them to more efficiently reach the goal. This allows our method to create much more intelligent behavior than previous methods that plan on a static snapshot of the world.

A static snapshot of the world is often insufficient to generate a feasible (much less optimal) trajectory in a highly dynamic environment, and the planner must consider how the world will change over the course of the planned path. Our method plans in both space and time, over a set of landmarks sampled on supporting surfaces in the world. Since space-time planning is performed on a much larger search space than spatial planning, standard search algorithms such as A^* are not efficient enough to plan and replan in real time. We therefore introduce a novel optimization to the A^* algorithm to make planning tractable for real-time applications.

The planner schedules a sequence of controllers that is optimal for the sampled landmarks. It is able to handle a variety of cost functions to achieve paths that exhibit desired behaviors, such as minimal traversal time, preference for specific actions, or preference for avoiding parts of the environment. This cost function can differ significantly from the one used by individual controllers, since it need not account for the details of individual tasks (such as completing a jump). This allows us to generate paths with different cost functions using the same set of controllers. Although the controllers are unaware of the planner's cost function, the optimal planning algorithm automatically selects those controllers that are most capable of minimizing the desired cost, resulting in a trajectory that is optimal up to the capabilities of the controllers.

While controllers can efficiently execute high-level locomotion tasks, they introduce additional complexity into the planning process. Previous methods have demonstrated efficient space-time planning for moving agents using backward planning from the goal, which allows the path to be continuously replanned by reusing information from previous planning episodes [van den Berg et al. 2006]. Unfortunately, backward planning is not suitable for controller-based navigation in dynamic environments. Since backward planning begins at the goal and plans towards the start, the planner must target not just a position in space-time, but also the current pose of the character, which has much higher dimensionality. Backward planning also requires a method for inverting the policy used by the controllers to select an optimal sequence of clips for a given set of task parameters. Finally, backward planning methods lose the ability to reuse information from previous episodes when the target moves unpredictably. Instead of using a backward planning algorithm and continuously replanning, we introduce a novel culling rule that allows A^*-based space-time search to run at interactive rates, while preserving optimality and completeness. This enables our approach to generate plans in real time, and even replan in real time with unpredictably moving targets.

2. RELATED WORK

Sampling-based planners were initially developed for robot path planning [Kavraki et al. 1994], and were subsequently adopted for path planning of virtual characters [Kamphuis et al. 2004]. Recently, several methods have been described that combine animation with path planning to create agile agents that can use motion capture to navigate complex environments. Choi et al. [2003] propose a method that samples potential footstep locations in the world and generates animations with footstep constraints, but it is not efficient enough to run in real time. Lau and Kuffner [2005] propose a method that searches over animation sequences generated by a finite state machine, in order to generate animations that traverse complex environments. A follow-up work [Lau and Kuffner 2006] describes a precomputed acceleration structure that enables the method to be used in real-time applications. However, these methods plan with individual animation clips rather than parameterized controllers. This limits the number of motion clips that the character can utilize, thus constraining the fluidity of the motion. In contrast, the number of motion clips used by individual controllers in our algorithm has no effect on the performance of the planner.

A number of methods have also been proposed that use probabilistic roadmaps to plan paths for agents, which are then animated using a separate animation module. Such methods are generally very fast and are often used for crowd animation [Sung et al. 2005; Sud et al. 2007; van den Berg et al. 2008], but usually assume that the scene is entirely or largely static.

Several of the previously mentioned methods are able to handle dynamic environments by quickly replanning an agent's path when its surroundings change [Sung et al. 2005; Lau and Kuffner 2006; Sud et al. 2007], or by using a local planner that avoids small

moving obstacles, such as other agents [van den Berg et al. 2008]. However, no technique has been proposed in computer graphics for taking dynamic obstacles into account during *global* planning. In robotics, global planning among predictably moving obstacles has been addressed with velocity obstacles, which constrain the plan to velocities that avoid collisions [Fiorini and Shiller 1998], as well as with space-time planning algorithms that plan in a joint space-time configuration space [Fraichard 1999; Zucker et al. 2007].

Early roadmap-based space-time planning methods directly sampled roadmap nodes from the four-dimensional space-time volume [Hsu et al. 2002]. More recent methods construct the roadmap on-the-fly, by first sampling a static spatial probabilistic roadmap, and then lazily creating space-time nodes as the spatial nodes are reached [van den Berg and Overmars 2005; van den Berg et al. 2006]. Since path planning in space-time may require the same spatial node to be revisited multiple times, a number of techniques have been proposed to speed up the planning process. Van den Berg and Overmars [2006] propose a method for culling previously visited nodes in the special case when the planner aims to minimize travel time, but no such method has been proposed for reducing the number of nodes with arbitrary cost functions. Space-time planning with more complex costs has been addressed with anytime replanning methods that trade off optimality for performance [van den Berg et al. 2006]. Our algorithm preserves optimality (within the sampling resolution and the capabilities of the controllers) and handles a variety of cost functions. Efficiency is retained with a novel culling technique that significantly reduces the number of nodes that must be explored.

To animate a character using motion capture, motion clips must be reassembled to follow a desired trajectory. Motion graphs accomplish precisely this [Arikan and Forsyth 2002; Kovar et al. 2002; Lee et al. 2002], and a number of recent developments have extended motion graphs to produce motions not present in the original database by interpolation [Safonova and Hodgins 2007] and warping [Sung et al. 2005]. Motion graphs have been extended for use in interactive applications through the use of controllers that are trained with reinforcement learning to select or interpolate clips for executing simple tasks [McCann and Pollard 2007; Treuille et al. 2007; Lo and Zwicker 2008]. By precomputing optimal ways to achieve specific tasks, these controllers remove the need to search the graph at runtime at the expense of being constrained to a low-dimensional representation of the environment (such as maneuvering around a single obstacle of varying size, or jumping onto a single platform of varying height). In order to traverse a complex environment, different controllers must be used in sequence. Coros et al. [2009] proposed a reinforcement learning method for sequencing physics-based controllers for performing simple tasks. Unfortunately, precomputed policies cannot sequence controllers in arbitrarily complex environments. Instead, we make use of planning at the high level to deal with complex, dynamic worlds, while precomputed controllers are used to execute individual tasks. Recently, Lee et al. [2009] proposed a technique for smoothly switching between kinematic controllers, which we employ to ensure that the transitions between our controllers are seamless.

3. OVERVIEW

The proposed method operates on an arbitrary polygonal environment in which objects move along known trajectories. The trajectories need not be periodic or finite, but the planner must be able to query the state of the world at any time. In a real-time, unpredictable application, this state can be obtained from the best known guess of how the environment will evolve. Formally, we assume that we have access to functions $\mathcal{O}, \mathcal{U} : \mathcal{X} \times \mathbb{R} \rightarrow \{0, 1\}$. $\mathcal{O}(x, t)$ tells us whether character configuration x at time t is in collision with an obstacle, and $\mathcal{U}(x, t)$ tells us if the character is not supported by a valid horizontal surface when in configuration x at time t. We specify the character's configuration with a tuple $x = (p, r, c, \alpha)$, where p and r are the character's position and rotation, c is the current parameterized motion clip, and α is the fraction of the clip that has elapsed. The method also takes as input a set of parameterized locomotion controllers $\pi_1, \ldots, \pi_n$. Each controller π_i is an optimal policy for executing a locomotion task (such as running or jumping), operating on the state-space $\mathcal{C} \times \Theta_i$, where $\mathcal{C}$ is the set of available parameterized motion capture clips, and Θ_i is the set of task parameters for the controller. The task parameter is a compact description of the controller's task and local environment. For example, it might specify the desired movement direction, or the location where a jump must begin.

The user specifies the character's starting configuration x_s and time t_s, as well as the desired goal position p_g. The user also specifies a cost function $\Delta(x, t, \pi)$. This cost function can be parameterized by any combination of configuration, time, and controller, and returns a cost *per second*. The total cost of a path can then be obtained by integrating over the cost function with respect to time. The output of the algorithm is a sequence of tuples $(\pi_1, \theta_1, \tau_1), \ldots, (\pi_m, \theta_m, \tau_m)$. By executing controller π_1 with parameters $\theta_1 \in \Theta_1$ for τ_1 seconds, followed by π_2 for τ_2 seconds, etc., the character will traverse the changing environment from the starting configuration x_s and time t_s to the goal location p_g. The algorithm is optimal up to sampling error (landmark placement and temporal sampling of waiting times) with respect to the cost function Δ, under the additional assumption that there exists a "waiting" controller that has lower cost per second than any other action. Varying the set of controllers or the cost function can produce a wide range of character behaviors, as demonstrated in Section 7.2 and the accompanying video.

In a preprocessing stage, the method samples a set of landmarks $\mathcal{P}$ on all potential supporting surfaces in the environment. These landmarks $p \in \mathcal{P}$ are stored relative to the surface on which they are sampled, so that $p(t)$ returns the location of the landmark at time t. Thus the landmarks "follow" the surface on which they were sampled. For a query (x_s, t_s, p_g), where x_s is the initial configuration of the character with position p_s, the algorithm associates the start and goal positions p_s and p_g with their supporting surfaces and adds them to $\mathcal{P}$. A space-time planner is then used to find an optimal path from the starting configuration to the goal using a variant of A^* search. The planner computes a path over nodes of the form (x, p, t), where $p \in \mathcal{P}$ is a landmark and $x \in \mathcal{X}$ is a configuration that is located within some tolerance of $p(t)$. When considering a new landmark p', the planner determines if a local path (or "edge") exists from an existing node (x, p, t) by executing a parameter adaptation algorithm that uses the functions $\mathcal{O}$ and $\mathcal{U}$ to find the parameters θ that allow a controller π to travel between the two landmarks, if possible. In the remainder of the article, we will use the terms "source" and "target" to denote the source and target landmarks for these local paths, saving "start" and "goal" for the global start and goal positions p_s and p_g.

Since the space of possible space-time nodes is very large, we introduce a culling rule that uses the previously stated waiting cost assumption, together with recently developed transition controllers [Lee et al. 2009], to plan at interactive speeds while preserving optimality. This makes our approach suitable for interactive applications, as demonstrated in Section 7.3, where we adapt the algorithm for pursuing an unpredictably moving target. Since the plan is constructed in both space and time, it takes into account the motion of obstacles and supporting surfaces,

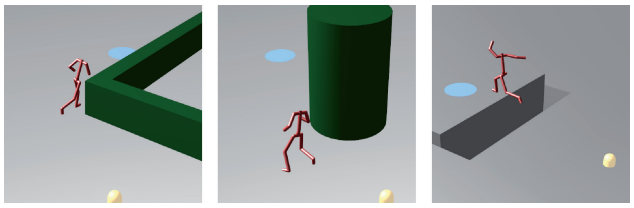


Fig. 2. Locomotion controllers used in our implementation. From left to right: a controller for running around a corner, avoiding a cylindrical obstacle, and jumping over a ditch at various heights.

allowing the character to avoid collisions and make use of moving platforms.

4. PARAMETERIZED LOCOMOTION CONTROLLERS

To animate a character traversing a complex scene, a motion graph can be "unrolled" into the environment, and the branch that reaches the target at lowest cost can be used as the synthesized path. This approach has been shown to produce compelling animations, but suffers long planning times that make it unsuitable for interactive applications [Safonova and Hodgins 2007]. Instead, we observe that the character will usually traverse the environment by performing a sequence of simple locomotion tasks, such as jumping or avoiding obstacles. By precomputing controllers for these tasks with reinforcement learning, we can compute the path much faster while retaining the high quality of animation driven by motion capture. Instead of planning an optimal sequence of motion clips, the planning algorithm need only plan an optimal sequence of controllers and adapt the low-dimensional task parameters of these controllers to the local environment.

4.1 Control through Reinforcement Learning

Our controllers are trained with reinforcement learning to reach a target in the presence of obstacles, by using a sequence of parameterized clips derived from motion capture data. The controllers are created using methods proposed by Treuille et al. [2007], Lo and Zwicker [2008], and Lee et al. [2009]. The controller is trained to maximize a reward function R that indicates the desirability of an action a at a state s. The reward function ensures that the controller accomplishes the desired task and penalizes unrealistic transitions, and is defined individually for each controller. The state is a tuple $s = (c, \theta)$, where c is the current parameterized clip and $\theta \in \Theta$ is a vector of task parameters. An action is specified by a pair $a = (c, \omega)$ that identifies the next parameterized clip and its parameters. Parameterized clips improve a controller's agility by allowing it to continuously vary some aspect of the motion, such as the precise heading after a running turn. As proposed by Lee et al. [2009], our parameters only influence the portion of the clip before the blending phase, allowing us to avoid adding the clip parameters to the state space. The parameters define the controller's internal obstacle representation, such as the position of a ditch the character must jump over.

Value iteration is used to compute a value function V for each controller, which gives the discounted infinite sum of the reward R. The value function enables us to select the long-term optimal action efficiently at runtime for current state s according to the policy $\pi(s) = \text{argmax}_a(R(s,a) + \gamma V(s'))$, where s' is the next state induced by the action a and γ is the discount factor. Figure 2 shows the three controllers used in our implementation, and their parameterizations are given in Appendix A.

4.2 Adaptation to Arbitrary Environments

In order to use a controller π to travel from a space-time node (x, p, t) to a target landmark p' through an arbitrary polygonal environment, we must be able to select appropriate task parameters θ. This is a challenging task, because the controller's simple internal abstraction of the environment may not precisely fit the local region. To find appropriate parameters, we iteratively test candidate parameters $\tilde{\theta}$ by simulating the resulting trajectories and, if these trajectories fail to arrive at the target p', treat the resulting failure locations as samples for adapting the controller's obstacle representation. A controller can fail to arrive at the target p' because the character either becomes unsupported or comes into collision; that is, for some time t and configuration x during the controller's trajectory, we have $\mathcal{O}(x,t) = 1$ or $\mathcal{U}(x,t) = 1$.[1] The failing configuration x is then added to a set of failure samples $\mathcal{F}$. An initial set of sample points is obtained by checking for collisions and loss of foot-support along rays to the target. Several rays are tested in slightly different directions, since most obstacle representations (such as cylinders) cannot be unambiguously fitted to a single point.

For each controller π, we construct a function $\phi_\pi(\mathcal{F})$ that fits the controller's internal obstacle representation to the set of failure samples $\mathcal{F}$ and returns the corresponding task parameters $\tilde{\theta}$. Starting with the initial failure set $\mathcal{F}$ obtained from the ray tests, we iteratively evaluate the controller starting at (x, p, t) with parameters $\phi_\pi(\mathcal{F})$, update the failure samples $\mathcal{F}$, and repeat. The algorithm terminates either when the controller successfully reaches the target landmark p', or when the internal obstacle representation cannot be successfully fitted to $\mathcal{F}$. The latter occurs if the algorithm does not succeed after a maximum number of attempts (5 in our prototype), or if the points in $\mathcal{F}$ do not satisfy the assumptions of the controller, for example if the fitted cylinder in a cylinder avoidance controller encloses the target location.

By fitting the obstacle representation to failure points along evaluated trajectories, the adaptation algorithm samples those parts of the environment that are relevant for the desired path. The whole environment is far too complex to fit to the obstacle representation directly, and by selecting only those points that actually cause failures, the controller is able to negotiate regions that are much more complex than its internal representation. This process is more efficient than randomly sampling obstructed points in the local region, since the most relevant failure points are discovered with a guided search from the initial samples. Further details on the implementation of ϕ_π for each of the controllers are discussed in Appendix A. In the remainder of this article, we will use $\Phi_\pi(x, t, p')$ to denote the task parameters for controller π returned by the adaptation algorithm for initial configuration x at time t and target position p'.

4.3 Waiting Transition Controllers

While most controllers employed by the planner are designed to travel between two points, the dynamic environment sometimes requires the character to wait at a specific location. It is not sufficient for a waiting character to simply stand in place; in order to appear natural and to traverse the environment optimally, the waiting character must use the available waiting time to prepare to execute the next action as efficiently as possible. For example, while waiting to jump on to a moving platform, the character must orient itself for the jump. In fact, this "preparation time" should allow the character

[1] In our implementation, $\mathcal{U}(x,t)$ tests whether the current stance foot is supported. Since there is no stance foot in the flight phase of running, or in a jump, these poses do not need to be supported.

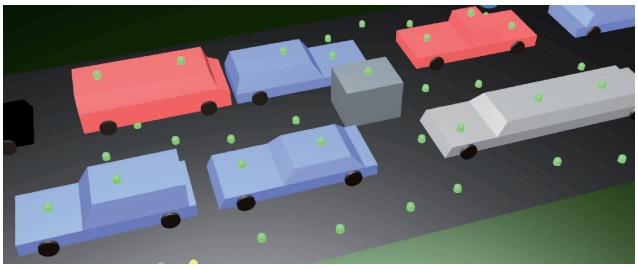


Fig. 3. The planner samples a set of landmarks on supporting surfaces. Since controllers can navigate around obstacles, landmarks need not be sampled densely. Note that landmarks are sampled on all potential supporting surfaces and follow these surfaces as they move.

to execute any subsequent action in the most efficient way possible from the current location.

To achieve this behavior, we formulate the waiting controller as a transition controller. The state space of the transition controller contains all of the clips and parameters of both itself and the next controller [Lee et al. 2009]. Since the only parameter of the waiting controller is the waiting time τ, the state of the waiting controller π_w that transitions into controller π is given by $s = (c, \theta_\pi, \tau)$. The reward function is identical to that of π, which causes the waiting controller to reposition the character to create an optimal transition into the next behavior. As will be discussed in Section 5.2, this property of the waiting controller allows us to introduce a culling rule that enables space-time planning to run at interactive rates.

5. SPACE-TIME PLANNING

Our method concatenates locomotion controllers to create a global plan using a specialized space-time version of the A^* search algorithm. In a preprocessing stage, a set of landmarks $\mathcal{P}$ is sampled from all potential supporting surfaces in the environment; that is, all surfaces that may support the character at some time t. In our implementation, this is also followed by a retraction step to avoid placing samples too close to edges or obstacles [Geraerts and Overmars 2004], but a number of other sampling approaches could be used [Kamphuis et al. 2004; van den Berg et al. 2005; Geraerts and Overmars 2006; van den Berg and Overmars 2006]. The start and goal landmarks p_s and p_g are also added to $\mathcal{P}$. Each $p \in \mathcal{P}$ is associated with its supporting surface (which can move over time), so that $p(t)$ gives the position of the landmark at time t. Figure 3 shows a set of example landmarks.

The planning itself is performed over space-time planning nodes (x, p, t), where $x \in \mathcal{X}$ is the configuration of the character, $p \in \mathcal{P}$ is the landmark, and t is the time at which the landmark is reached. Since each planning node is associated with the time t at which it is reached, the planner can use knowledge about how the environment will change over time to construct a path that takes these changes into account. The space-time graph is not generated explicitly. Instead, a node (x, p, t) is created when the landmark p can be successfully reached from an existing node. To make planning efficient in this high-dimensional environment, we introduce a culling rule $\mathcal{W}(x, p, t)$ that rejects those nodes that can be reached at lower cost from another path by waiting. In our experiments, we found this culling rule to be crucial for achieving interactive performance, providing a 2–50× speedup over conventional A^*, as discussed in Section 7.2. To handle nodes that must be revisited at a later time because they become blocked by obstacles, we also introduce *gap markers* that prevent the node from being culled when it is revisited.

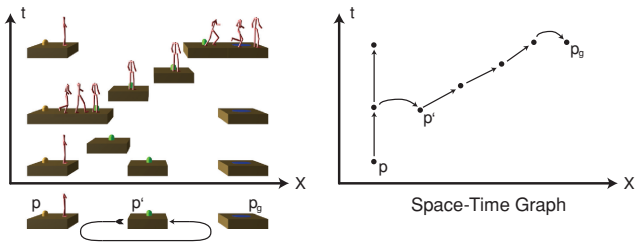


Fig. 4. The character can travel to p_g by stepping on the moving platform node p', but this path is not apparent without considering the motion of the platform. In space-time planning, the world is unrolled along the time axis, revealing paths that may not be present in a static snapshot.

Unlike previous space-time planning methods, we do not rely on a specific discretization of time.

The optimality and completeness of the proposed algorithm follows directly from the optimality and completeness of A^* search. For A^* to be optimal, nodes must be explored in order of increasing f_{score}, and the f_{score} must give a lower bound on the true cost of a path from the start to the goal through that node. Our algorithm preserves these invariants. All paths that are culled are more expensive than some other path leading to the same node, and therefore cannot possibly be part of an optimal solution. Since the proposed heuristic is also admissible and consistent, our space-time search algorithm retains the optimality of A^* search.

Although the use of controllers allows planning with sparser samples, evaluating an edge between two nodes to determine its cost requires the adaptation of controller parameters for navigating the region between the nodes, as discussed in Section 4.2. In experiments, parameter adaptation took 2 ms per edge, making it a relatively time-consuming process when each path requires hundreds of planning nodes to be explored. The planner must therefore minimize the number of trajectory evaluations.

In order to generate a range of behaviors, the planner must also be able to produce optimal paths that satisfy different cost functions $\Delta(x, t, \pi)$. For example, the character might prefer to traverse an environment in minimal time, with minimal expenditure of energy, or in a way that avoids specific areas. Our space-time planning algorithm is able to maintain optimality for any cost function that assigns a lower cost per second to the waiting controller than to any other action. This assumption is necessary to ensure that we can cull nodes that can be reached at lower cost by waiting, as discussed in Section 5.2.

5.1 Successor Exploration

Our planning algorithm is based on A^* search. In A^*, explored nodes are added to an open set and sorted by f_{score}, defined as $f_{\text{score}} = g_{\text{score}} + h_{\text{score}}$, where g_{score} is the cost of reaching the node from the start and h_{score} is an *admissible heuristic* that gives a lower bound on the cost to travel from the node to the goal. The f_{score} of a node represents a lower bound on the cost of a path from the start to the goal through that node. At each iteration, the node with the lowest f_{score} is removed from the open set, and all of its successors are explored and added to the set. Planning terminates when the goal node is removed from the open set.

We define a planning node in the space-time search as a tuple (x, p, t), where x is the configuration of the character with location at the landmark p, and t is the time at which the landmark was reached. Figure 4 shows a simple one-dimensional world and the matching space-time graph, with the $\mathcal{X}$ dimension omitted for

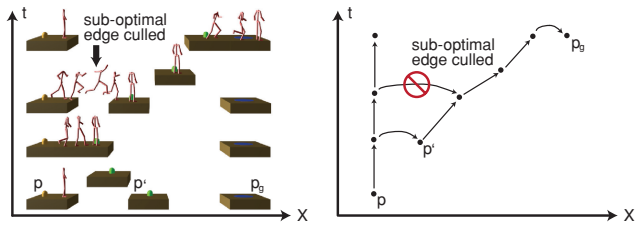


Fig. 5. We cull planning nodes that can be reached at lower cost by waiting from another node. Here, the character can reach the platform by stepping onto it and waiting, making the later jump action redundant.

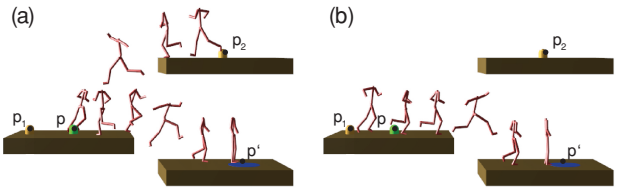


Fig. 6. The character's configuration x on reaching p affects the cost of the edge from p to p'. When arriving at p from p_2 in (a), the character must turn around to jump to p'. When arriving from p_1 in (b), the jump can be performed much faster.

simplicity. The space-time graph itself is not constructed explicitly. Instead, connections are considered between the node (x, p, t) and all landmarks p' that are within some threshold distance of $p(t)$ at time t. For each controller π and each candidate target p', the parameter adaptation algorithm from Section 4.2 is used to find the parameters $\theta_\pi = \Phi_\pi(x, t, p')$ and check if p' is reachable. If it is, a new planning node (x', p', t') is added to the open set. Let $x_{\pi,\theta_\pi}(\tau)$ be the configuration of the character at time τ after invoking the controller π with parameters θ_π, starting from the current node's configuration x. The controller terminates when the position of $x_{\pi,\theta_\pi}(\tau)$ is close to p' (1 meter in our implementation), with t_ϵ denoting the corresponding value of τ. We then set $x' = x_{\pi,\theta_\pi}(t_\epsilon)$ and $t' = t + t_\epsilon$. The cost c_ϵ of the new edge is given by $\int_0^{t_\epsilon} \Delta\left(x_{\pi,\theta_\pi}(\tau), \tau + t, \pi\right) d\tau$. We will denote the function that computes c_ϵ, t_ϵ, and x' as $(c_\epsilon, t_\epsilon, x') = g_\epsilon(x, p, t, \pi, p')$.

In addition to constructing nodes for each landmark that can be reached from the current node (x, p, t), a special node (x_w, p, t_w) is constructed to represent a wait. Unlike the controllers that travel to other nodes, the waiting controller π_w can select any value $t_w \geq t$. The choice of t_w constitutes a temporal sampling problem. Intuitively, t_w should be chosen so that the character waits the minimum possible time until the surrounding environment changes in a relevant way, for example, a new outgoing edge from p appears, or an existing edge becomes less expensive to traverse. In our implementation, waiting times t_w are sampled in 1-second increments, and a sample t_w is only accepted if it is possible to reach any successor of (x_w, p, t_w) at a lower cost than before. How this is determined is discussed at the end of Section 5.2. Note that the 1-second sampling increments do not commit our method to a specific discretization of time, as nonwaiting nodes are created at whatever time the target landmark is reached.

5.2 Culling Suboptimal Planning Nodes

A^* search avoids revisiting previously explored nodes by using a closed set. A node is added to the closed set when it is removed from the open set, because at this point no lower-cost path can exist through this node. In the context of the proposed space-time planning algorithm, the closed set provides little benefit, because the same landmark p might be visited at many different points in time, and in many different configurations x. Without an additional culling scheme, such a search space would quickly become intractable. Recall, however, that we assumed that the cost function $\Delta(x, t, \pi)$ assigned a lower cost to the waiting controller π_w than to any other action. We leverage this assumption to make the search tractable by culling paths through landmarks when another path exists that can reach that landmark at lower cost by waiting, as shown in Figure 5.

Formally, the assumption on the cost of waiting ensures that $\Delta(x_1, t_1, \pi_w) \leq \Delta(x_2, t_2, \pi)$ for any x_1, x_2 and t_1, t_2, where π_w is the waiting controller and $\pi \neq \pi_w$. Note that for most cost functions, such as shortest time, least effort, etc., it is perfectly reasonable to expect waiting to be no more "costly" per second than any other activity. For the space-time planning algorithm, this assumption guarantees that, regardless of the actual trajectory taken by the waiting controller at landmark p, the cost per second over the elapsed time would be no greater than if any other controller had been used. Intuitively, it means that it is always less expensive to stay in place (if possible) than to travel to another landmark and come back. We leverage this property to avoid adding provably suboptimal planning nodes to the open set. When a waiting node (x_w, p, t_w) is created, we store the time t_{ws} at which the wait begins and the cost of the path preceding the wait $g_{\text{score}}(x_{ws}, p, t_{ws})$. Before evaluating a new path to landmark p at time t, we first check all waiting nodes at p for which $t_{ws} + t_{min} \leq t$, where t_{min} is discussed in the next paragraph. We compute a bound on the g_{score} of a hypothetical path that waits from t_{ws} to t and, if the cost of this path is lower than the g_{score} of the new path, we do not add the new node (x, p, t) to the open set, since we know that this node can be reached at lower-cost from (x_{ws}, p, t_{ws}) by waiting. We will denote this culling test $\mathcal{W}(x, p, t)$, where $\mathcal{W}(x, p, t) = 1$ denotes that the node passed the test, while $\mathcal{W}(x, p, t) = 0$ denotes that a lower-cost waiting path exists.

One difficulty with this approach is that the configuration x of the character at node (x, p, t) may have an impact on the cost of a subsequent edge, that is, the cost of traveling from (x, p, t) to another landmark p' may depend on x, which in turn depends on how the character arrived at p, as shown in Figure 6. However, consider that, if the character arrives at p early and waits, it can use this waiting time to leisurely prepare to travel to p' at the lowest possible cost. Recall from the discussion of the waiting controller in Section 4.3 that this is precisely the functionality that is provided by the waiting transition controllers, which determine the optimal way to transition into the next controller. We formalize this notion of "preparation time" with the parameter t_{min}, which specifies the minimum time needed for the waiting transition controller to enter the optimal configuration p for an outgoing edge, so as to traverse at the lowest possible cost. Note that, since the character remains near p, this preparation time is generally very short. In our prototype, t_{min} was set to 1 second.

To compute a bound on the g_{score} of a hypothetical waiting path from t_{ws} to t, we use the maximum possible cost of waiting per second, $\Delta_w = \max_{x,t} \Delta(x, t, \pi_w)$. The cost of the wait can then be bounded by $\Delta_w(t - t_{ws})$, allowing us to bound the g_{score} of reaching (x, p, t) by waiting by

$$g_{\text{score}}(x, p, t) \leq g_{\text{score}}(x_{ws}, p, t_{ws}) + \Delta_w(t - t_{ws}).$$

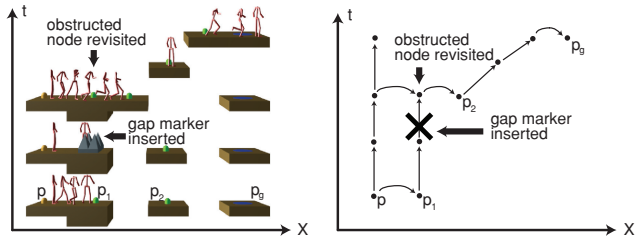


Fig. 7. When a node becomes obstructed due to a moving obstacle, we insert a gap marker to indicate that subsequent visits to this node should not be culled. In this example, a gap marker inserted at p_1 when it becomes obstructed allows the character to revisit it in order to reach the moving platform.

We can then avoid adding to the open set any node (x, p, t) reached from another node (x_p, p_p, t_p) if its g_{score} is higher than this bound. However, since the cost of edge evaluation is high, we can use the assumption that waiting costs no more per second than another action to instead use a lower-bound estimate of the time t at which the node can be reached, allowing us to avoid evaluating the edge between (x_p, p_p, t_p) and (x, p, t), as discussed further in Section 5.4. The same culling rule can also be used when sampling waiting times to determine if a waiting node (x, p, t_w) at a temporal sample t_w can possibly reach any of its successors at a lower cost.

5.3 Obstructed Nodes

The culling rule described in the previous section culls a node (x, p, t) if a wait beginning at (x_{ws}, p, t_{ws}) exists from which the character could arrive at (x, p, t) at lower cost. However, when a landmark p becomes obstructed by an obstacle at time t_b, with $t_{ws} < t_b < t$, the node (x, p, t) should not be culled, because it cannot actually be reached by waiting from (x_{ws}, p, t_{ws}). To solve this problem, we introduce *gap markers*. A gap marker (p, t_b) is created when a waiting edge is evaluated between (x_{ws}, p, t_{ws}) and (x_w, p, t_w) and found to intersect an obstacle at time t_b. When the culling rule is applied to node (x, p, t), only waiting nodes starting at time t_{ws} for which no gap marker exists between t_{ws} and t are considered, since nodes that begin before a gap are unable to reach the current time by waiting, as shown in Figure 7. In the next section, we describe a modification to the planning algorithm that ensures that the culling test is performed only when a node (x, p, t) is removed from the open set, which guarantees that the test is performed on the node with the lowest f_{score} of any open node. In Appendix B, we prove that this guarantees that all gap markers (p, t_b) with $t_b < t$ have already been explored.

5.4 Deferred Edge Evaluation

In order to handle obstructed nodes, we must perform the culling test just on the node with the lowest f_{score} in the open set. Since the evaluation of an edge requires adaptation of controller parameters, and is therefore relatively costly, we would like to only evaluate edges that pass the culling test. We therefore add nodes to the open set without fully evaluating their edges, and only evaluate them once they have the lowest f_{score} in the open set and pass the culling test. To add a node to the open set without complete evaluation, we construct an admissible lower-bound estimate for its g_{score} and the time t at which it is reached from its predecessor (x_p, p_p, t_p) with the controller π_p, denoted $(c_\epsilon, t_\epsilon) = h_\epsilon(p_p, t_p, \pi_p, p)$. The function h_ϵ computes a lower bound on the cost of the edge c_ϵ and its length t_ϵ by using the maximum speed of the controller π_p and its minimum cost per second, $\min_{x,t} \Delta(x, t, \pi_p)$, together with the distance between $p_p(t_p)$ and $p(t_p)$. The time and g_{score} bounds are then given by $t = t_p + t_\epsilon$ and $g_{\text{score}}(\emptyset, p, t) = g_{\text{score}}(x_p, p_p, t_p) + c_\epsilon$. The configuration x at the new node is set to $\emptyset$ to denote that it is unknown.

When a node (x, p, t) has the lowest f_{score} of any open node, we evaluate $\mathcal{W}(x, p, t)$. If $\mathcal{W}(x, p, t) = 0$, the node is removed from consideration. Otherwise, its f_{score} is updated to better reflect its actual cost in one of two ways. As discussed in Section 5.5, the heuristic value of a node can increase as additional nodes are explored, so if other nodes were explored since (x, p, t) was last examined, its h_{score} is recomputed and it is added back to the open set. If its h_{score} is up to date, the node's g_{score} is computed by fully evaluating the edge, the time t is updated to reflect the actual time at which the node is reached, and the node is again added back to the open set. If both the h_{score} and g_{score} are already computed, the node is expanded and new planning nodes are created for all of its successors with each controller. Since the sum $h_{\text{score}} + g_{\text{score}}$ can only increase with each update, the A^* invariant that all unexplored nodes have a higher f_{score} than explored nodes is maintained. Waiting nodes are evaluated immediately to ensure that the gap marker optimality proof in Appendix B holds.

5.5 Computing Admissible Heuristics

The Euclidean distance heuristic that is most commonly used with A^* is not directly applicable to dynamic environments with moving surfaces, because the maximum speed of the character cannot be easily bounded. For example, a fast-moving platform might take the character straight to the goal, yielding nodes whose cost is much smaller than their Euclidean distance to the goal would imply.

We instead compute an admissible heuristic by constructing a static graph that captures the best-case connectivity between the landmarks, with edge lengths corresponding to the minimum possible distance between pairs of landmarks $p_1, p_2 \in \mathcal{P}$. Moving obstacles in games often exhibit periodic motion or move within fixed domains [van den Berg and Overmars 2006], which allows minimum landmark distances to be computed analytically. More generally, landmark positions can be sampled at fixed intervals between the current time and the latest expected planning time, which is the method employed in our prototype. When the time t^* at which the distance between p_1 and p_2 is minimized is known, the minimum cost of any edge between them is given by $h^0_\epsilon(p_1, p_2) = \min_\pi h_\epsilon(p_1, t^*, \pi, p_2)$, the lowest heuristic cost with any controller. In our prototype, the static graph is constructed very quickly, as shown in Section 7.2.

We can obtain an admissible heuristic for a node (x, p, t) by performing a search on the static graph, since for any edge between two planning nodes, the static graph must contain an edge with equal or lower cost. We can further improve the heuristic if we note that a path will not revisit landmarks that are culled by the culling rule in Section 5.2 until after a gap marker. We construct an improved minimum edge cost $h^1_\epsilon(p_1, t, p_2)$ that checks if (p_2, t) passes the culling test. If it does, then we simply set $h^1_\epsilon(p_1, t, p_2) = h^0_\epsilon(p_1, p_2)$. Otherwise, we know that p_2 cannot be reached earlier than t_f, the earliest time at which p_2 is not culled. t_f is either the next gap marker after t, or, if no such gap marker exists, the latest time at which p_2 has been explored. Given $(c_\epsilon, t_\epsilon) = h^0_\epsilon(p_1, p_2)$, we can bound the cost of the edge from p_1 to p_2 by the minimum cost to travel between the two landmarks c_ϵ, plus the minimum cost of passing the remaining time $t_f - t - t_\epsilon$. Since we assumed earlier that waiting has the lowest cost per second of any controller, the

```
1: W(x, p, t)
2: for all waits starting at (x_ws, p, t_ws) such that t_ws + t_min ≤ t do
3:     blocked ← false
4:     for all gap markers (p, t_b) such that t_b ≥ t do
5:         if t_b ≤ t_ws then
6:             blocked ← true
7:         end if
8:     end for
9:     if blocked = false then
10:        if g_score(x, p, t) ≥ g_score(x_ws, p, t_ws) + Δ_w(t − t_ws) then
11:            return 0 (cull)
12:        end if
13:    end if
14: end for
15: return 1 (do not cull)
```

Fig. 8. Culling rule pseudocode. A node (x, p, t) is culled if the same space-time location can be reached at lower cost by waiting.

total lower bound is given by

$$h^1_\epsilon(p_1, t, p_2) = c_\epsilon + (t_f - t - t_\epsilon) \min_{x,t} \Delta(x, t, \pi_w).$$

The full heuristic, denoted $h(p, t)$, is computed by performing a search *on the static graph*, using h^1_ϵ as the edge cost. This *static* search can be performed efficiently using the standard A^* algorithm, with an admissible heuristic constructed by precomputing the cost to the goal from each node according to h^0_ϵ at the beginning of each planning episode (using Dijkstra's algorithm). After each edge traversed during the *static* search, t is either incremented by t_ϵ, or set to t_f if (p_2, t) does not pass the culling test.

Admissibility of the heuristic follows from the fact that the cost of any path through landmarks $p_1, p_2, \ldots, p_n$ starting at time t is equal to or greater than its cost according to h^1_ϵ. If none of the nodes is culled, this follows from the fact that h^0_ϵ is a lower bound on any edge between two landmarks. If p_k is the last node in the path that is culled, then the path from p_1 to p_k must have a cost no lower than the minimum cost of the edges up to p_k, plus the cost of waiting until the earliest time that p_k is not culled, which is exactly the cost given by h^1_ϵ. Since p_k is the last culled node, the cost under h^1_ϵ of the remaining edges is equal to their cost under h^0_ϵ. Therefore, h^1_ϵ gives a lower bound on the cost of the entire path.

6. ALGORITHM SUMMARY

Pseudocode for the culling rule is presented in Figure 8, and the complete planning algorithm is summarized in Figure 9. At each iteration, the node (x, p, t) with the lowest f_{score} is removed from the open set queue. The node is tested against the culling rule $\mathcal{W}(x, p, t)$, which checks for existing waits at the landmark p that can be used to arrive at (x, p, t) at lower cost. If the node is not culled, its f_{score} is updated in one of two ways. If other nodes were explored since this node was last checked, its h_{score} is recomputed. Otherwise, the node's actual g_{score} is computed by evaluating the edge from (x_p, p_p, t_p) to p using the associated controller π_p.

To evaluate an edge and determine its cost, the parameter adaptation algorithm from Section 4.2 is used to find parameters $\Phi_{\pi_p}(x_p, t_p, p)$ for controller π_p that allow it to travel from p_p to p, starting at time t_p. The cost is computed from this trajectory by integrating the current cost function $\Delta(x, t, \pi)$ with respect to time. For example, a function that minimizes travel time would simply set $\Delta(x, t, \pi) = 1$, while a function that minimizes energy might examine the animation to determine its energy expenditure per second.

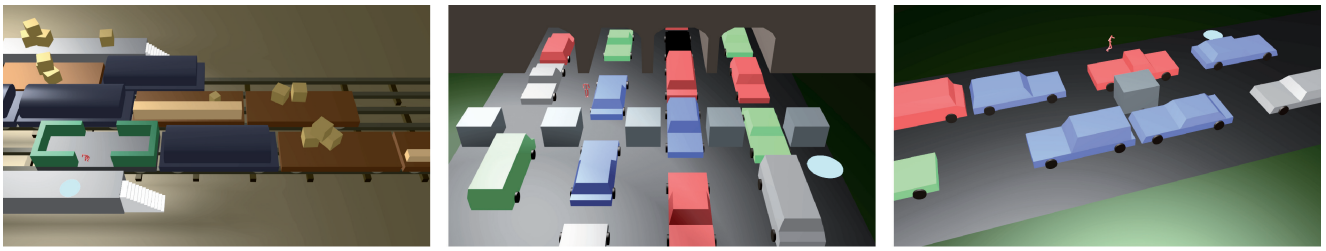
```
1: g_score(x_s, p_s, t_s) ← 0
2: f_score(x_s, p_s, t_s) ← h(p_s, t_s)
3: open ← {(x_s, p_s, t_s)}
4: while open ≠ ∅ do
5:     (x, p, t) ← node in open set with lowest f_score
6:     if W(x, p, t) = 0 then
7:         remove (x, p, t) from open set
8:     else if h_score(x, p, t) not up to date then
9:         h_score(x, p, t) ← h(p, t)
10:        f_score(x, p, t) ← h_score(x, p, t) + g_score(x, p, t)
11:    else if g_score(x, p, t) not evaluated then
12:        (c_ε, t_ε, x) ← g_ε(x_p, p_p, t_p, π_p, p)
13:        t ← t_p + t_ε
14:        g_score(x, p, t) ← g_score(x_p, p_p, t_p) + c_ε
15:        f_score(x, p, t) ← h_score(x, p, t) + g_score(x, p, t)
16:    else
17:        if p = p_g then
18:            return path by using predecessor pointers from (x, p, t)
19:        end if
20:        remove (x, p, t) from open set
21:        for all p′ s.t. p′(t) within L (threshold) units of p(t) do
22:            for all non-waiting controllers π do
23:                (c_ε, t_ε) = h_ε(p, t, π, p′)
24:                t′ ← t + t_ε
25:                g_score(∅, p′, t′) ← g_score(x, p, t) + c_ε
26:                h_score(∅, p′, t′) ← h(p′, t′)
27:                f_score(∅, p′, t′) ← h_score(∅, p′, t′) + g_score(∅, p′, t′)
28:                add (∅, p′, t′) to open set
29:            end for
30:        end for
31:        sample waiting time t_w > t
32:        if character in collision at (p, t_b), t < t_b ≤ t_w then
33:            create gap marker at (p, t_b)
34:        else
35:            g_score(x_w, p, t_w) ← g_score(x, p, t) + g_ε(x, p, t, π_w, p)
36:            h_score(x_w, p, t_w) ← h(p, t_w)
37:            f_score(x_w, p, t_w) ← h_score(x_w, p, t_w) + g_score(x_w, p, t_w)
38:            add (x_w, p, t_w) to open set
39:        end if
40:    end if
41: end while
```

Fig. 9. Planning algorithm pseudocode. (x_p, p_p, t_p) is the predecessor of (x, p, t) and π_p is the controller used to reach (x, p, t).

If the node's f_{score} is already fully evaluated and the node passes the culling test, all of its successors p' are added to the open set with an admissible lower-bound estimate of their f_{score}, as discussed in Section 5.4. A waiting time t_w is also sampled, and the edge to the waiting node (x_w, p, t_w) is evaluated immediately. If the wait fails, a gap marker (p, t_b) is inserted at the time of failure t_b. The planning algorithm is optimal within the sampled landmarks up to the sampling resolution of waiting nodes and the capabilities of the controllers.

7. RESULTS

We tested the presented method on a number of highly dynamic environments that contain both moving obstacles and moving supporting surfaces. The planner was provided with controllers for jumping to different heights, running in the vicinity of a small obstacle, and running around a corner. These controllers are described in detail in Appendix A. In addition to environments where the



(a) train yard (b) toll booth (c) highway

Fig. 10. The method was evaluated on three highly dynamic environments. The train yard environment has no path between the start and goal in a static snapshot of the world, the toll booth environment demonstrates obstructed nodes on the road and dynamic obstacles, and the highway environment is used to demonstrate various cost functions. Traversals of these environments are presented in the accompanying video.

character was given perfect knowledge of the future configuration of obstacles, the planner was evaluated on a real-time task in which the character had to pursue an unpredictably moving target.

7.1 Experimental Setup

The environments presented in our evaluation require the character to constantly interact with dynamic objects. The accompanying video contains animations for each environment discussed in this section, and images of the environments are presented in Figure 10. In the two "train yard" scenes, a static snapshot of the world does not contain a path from the start to the goal, and such a path can only be discovered with space-time planning. The two environments are identical except for the timing of the trains, to demonstrate that the planner is able to select appropriate waiting times in order to time jumps between trains correctly. The "toll booth" environments demonstrate handling of obstructed nodes on the road and the use of moving surfaces to reach otherwise unreachable locations. Two versions of the environment are again presented, with cars moving at different speeds. The "highway" environment is used to demonstrate how the planner can handle a variety of different cost functions. In addition to the minimal time path, we show a path that minimizes movement by penalizing nonwaiting actions, a path that penalizes running on the road, and a path that favors walking over running and jumping. For the walking example, the planner was provided with a controller for running and jumping as well as a controller for walking slowly around an obstacle.

7.2 Performance

Table I presents performance results for the method. Our single-threaded prototype was evaluated on an Intel Core i7 3.07 GHz with 8GB RAM. For each path, we list the total planning time, the portion of that time that was spent generating the static graph for heuristic computation, and the total number of edges evaluated during the search. We also list the time necessary to compute the path without the culling rule presented in Section 5.2, using only the closed set formulation from the standard A^* algorithm, in which a node is skipped only if another node exists at the same position and time (with 0.05-second tolerance) but at lower or equal cost. The results indicate that our algorithm computes paths at interactive rates and that the culling rule produces a speedup of 10–50× on the larger "train yard" and "toll booth" environments, and a speedup of 2–50× on the smaller "highway" environment.

Table I.

Scene	Cost function	planning time (s)	heuristic time (ms)	path length (s)	trajectories evaluated	planning without culling (s)
train yard 1	min time	1.08	16	17.0	502	25.8
train yard 2	min time	1.18	16	24.9	508	52.9
toll booth 1	min time	1.79	78	14.0	934	17.3
toll booth 2	min time	1.93	78	14.4	980	47.8
highway	min time	0.99	47	9.2	557	19.1
highway	min motion	1.37	47	12.9	662	5.6
highway	avoid road	1.31	47	15.6	551	2.8
highway	prefer walk	2.35	47	22.8	1065	107.8

We present total planning times for paths of various lengths, the time needed to compute the heuristic static graph, the total number of trajectories evaluated, and planning times using the standard A^* algorithm without the new proposed culling rule.

7.3 Online Replanning

In addition to the examples in the previous section, we evaluate the method on a real-time task in which the character must pursue an unpredictably moving target in a "platformer video game" environment. For each planning episode, the planner is provided with the target's current position and velocity. The planner selects the expected planning time and plans a path starting at the location and time that the character will occupy after this time passes. Once the planning time has elapsed, the planner provides its current best guess for the correct path. If the search completed successfully, this is the correct path to the target from the current time, based on the target position as estimated at the beginning of the planning process. Otherwise, the node with the lowest f_{score} is removed from the open set and the path to that node is used. The expected planning time was selected as 1.5 times the length of the previous planning episode, in order to allow some margin of error.

In the three replanning examples in the accompanying video, the planner found a path to the goal in the allotted time in 93% of the planning episodes, often with ample time to spare. In 68% of the episodes, the alloted time was less than the time needed to reach the next node, so the planner produced a new path as early as it could possibly be put to use. In an additional 12% of the episodes, the planner completed within a single edge traversal but the alloted planning time was greater, so the new plan was not implemented until after the next edge. When approaching to within a few nodes of the target, planning times fell to around 50 ms, fast enough to be computed in only a few frames for agile chasing behaviors. Only 16% of the planning episodes took over 1 second to complete.

Although the target's position changed unpredictably, the planner still used its knowledge of how other obstacles in the scene would move to plan complex paths that intercepted the target using moving platforms, as can be seen in the accompanying video.

8. DISCUSSION

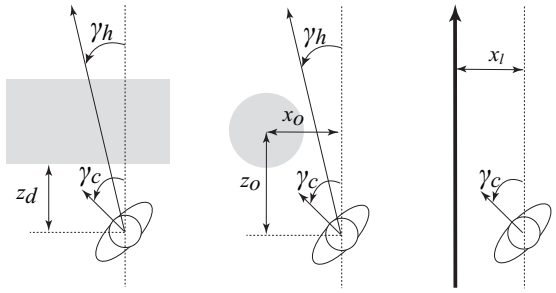
We presented a method for planning animated paths for characters in highly dynamic environments. The proposed method is able to take into account the motion of objects in the world using a space-time planning algorithm, and uses a novel culling rule to make the space-time search fast enough for real-time applications. To our knowledge, this is the first application of space-time planning to character animation. Our planner is also the first to use precomputed locomotion controllers to find local paths between landmarks. We demonstrate that such controllers can be effectively adapted to arbitrary polygonal environments, and that they can be concatenated to construct long paths without loss of motion quality. Our planning algorithm is optimal within the sampled landmarks, up to the sampling of waiting times and the capabilities of the controllers.

While the method is efficient enough to replan in real time, it does not guarantee that a good solution can always be found within a limited amount of time. To adapt the algorithm for widespread use in interactive applications, where numerous characters might need to be simulated, it would be necessary to develop an efficient replanning scheme that reuses information from previous planning episodes. This replanning problem has been addressed in the robotics community with backward planning methods [van den Berg et al. 2006; Likhachev et al. 2005] but, as discussed in Section 1, such methods are not appropriate for controller-driven animation. Future work could address the topic of continuous replanning for controller-driven animation in more detail. In less predictable environments, space-time planning can also be performed with a short horizon, using a static planner to select milestones.

Additionally, while the running time of the planner is independent of the number of motion clips used by the controllers, it is not independent of the number of the controllers themselves. The culling rule serves to remove edges for highly suboptimal controllers, but when several controllers can reach the same node close together in time, all edges must be evaluated. In future work, this limitation can be addressed by intelligently selecting which controllers to use for which edges.

Finally, optimal planning, especially in complex, changing environments, may not always be desirable for virtual agents. To emulate human behavior, some degree of suboptimality may be needed to make the agent appear human. To this end, a modification of the planning algorithm to simulate unobserved or unpredicted obstacle motion, or the use of approximate, short-horizon guesses, may create a method that is better able to emulate human behavior.

The space-time locomotion controller planning algorithm presented in this article is the first of its kind for planning animations for virtual characters. It is able to traverse complex, highly dynamic environments, and generates high-quality animations from a large body of motion capture. This technique can be used to create characters that appear more intelligent and are able to discover intricate paths through virtual environments that even a human observer might not discern. More fundamentally, the use of precomputed controllers for executing simple tasks is an approach to the planning problem that has received little attention. This technique effectively shifts the dimensionality problem from the number of motion clips to the dimensionality of the controller: instead of selecting individual clips, we instead find a good representation for the local obstacles using one of a few fixed low-dimensional representations.



(a) jumping (b) obstacle avoidance (c) corner

Fig. 11. Each controller is parameterized by an obstacle representation and, in the case of jumping and obstacle avoidance, a heading to the target.

In environments that consist of predictable components, such as corners and ditches, this method allows for high-quality planning in substantially fewer dimensions. Further research into the benefits of such "task-centric" planning may yield useful algorithms for both animated characters and robots.

APPENDIXES

A. CONTROLLER PARAMETERS

In this appendix, we describe the obstacle representation for each of the controllers in our prototype (Figure 11), as well as how that representation is fitted to sampled failure points using the algorithm in Section 4.2.

Jumping. The jumping controller jumps over a linear, fixed-width ditch. The obstacle is defined by the orientation of the ditch relative to the character γ_c, the distance to the ditch z_d, and the height difference h_d between the two sides. An additional parameter γ_h provides a continuously updated heading to the target position, ensuring that the controller reaches the target if possible. The ditch is fitted to a set of sampled failure points such that all points lie on the side opposite from the start. The height of the destination is selected to be the height of the target at the time of the last failure.

Obstacle avoidance. The obstacle avoidance controller avoids a cylindrical obstacle with a fixed radius. The cylinder is defined by the position of the character relative to the obstacle (x_O, z_O) and the orientation of the character relative to the obstacle's coordinate frame γ_c. γ_h again provides a continuously updated heading to the target. The obstacle is fitted to the failure samples by enclosing all points within the cylinder such that center is placed as far as possible from the line between the start and the target while still enclosing all samples.

Corner. The corner controller follows a line to navigate around a corner, by first approaching the line at a right angle and then following it to the target. The line is represented by the relative orientation γ_c and the distance to the line x_l. Since the line represents free space rather than obstacle, it is fitted such that all sampled points are at least a fixed clearance away from both the line and the perpendicular segment between the line and the starting point.

B. GAP MARKER OPTIMALITY PROOF

We show that if the node (x, p, t) has the lowest f_{score} of any open node, then all gap markers (p, t_b) with $t_b < t$ have already been explored. This allows us to cull (x, p, t) if it does not pass the culling test in Section 5.2 without loss of optimality.

PROOF. Assume (x, p, t) is culled because of a wait (x_w, p, t_w) beginning at t_{ws}, and that an unexplored gap exists at t_b, with $t_{ws} \leq t_b \leq t$. Since the gap at t_b is unexplored, we have $t_w < t_b \leq t$. By the culling rule, we have that $g_{\text{score}}(x, p, t) \geq g_{\text{score}}(x_{ws}, p_{ws}, t_{ws}) + \Delta_w(t - t_{ws}) \geq g_{\text{score}}(x_w, p, t_w) + \Delta_w(t - t_w)$. Since the heuristic search on the static graph adds to the cost of each culled landmark the cost of waiting until the earliest time t_{clear} it is unexplored or has a gap, given by $\Delta_w(t_{clear} - t)$, as t increases, heuristic values decrease at most Δ_w per second. Therefore, $h_{\text{score}}(x_w, p, t_w) \leq h_{\text{score}}(x, p, t) + \Delta_w(t - t_w)$. Adding the h_{score} and g_{score}, we have $f_{\text{score}}(x_w, p, t_w) \leq f_{\text{score}}(x, p, t)$. If we break ties according to t, since $t_w < t$, the node (x_w, p, t_w) must be expanded before (x, p, t), producing a new wait (x'_w, p, t'_w). By induction, we must therefore have $t_w > t$, and therefore $t_w > t_b$. This means that either $t_{ws} > t_b$, or the wait crosses the gap, which contradicts the assumption that t_b is unexplored. □

REFERENCES

ARIKAN, O. AND FORSYTH, D. A. 2002. Interactive motion generation from examples. In *ACM SIGGRAPH 2002 Papers*. 483–490.

CHOI, M. G., LEE, J., AND SHIN, S. Y. 2003. Planning biped locomotion using motion capture data and probabilistic roadmaps. *ACM Trans. Graph.* 22, 2, 182–203.

COROS, S., BEAUDOIN, P., AND VAN DE PANNE, M. 2009. Robust task-based control policies for physics-based characters. In *ACM SIGGRAPH 2009 Papers.* ACM Press.

FIORINI, P. AND SHILLER, Z. 1998. Motion planning in dynamic environments using velocity obstacles. *Int. J. Robot. Res. 17*, 7, 760–772.

FRAICHARD, T. 1999. Trajectory planning in a dynamic workspace: A 'state-time space' approach. *Adv. Robot. 13*.

GERAERTS, R. AND OVERMARS, M. 2004. Clearance based path optimization for motion planning. In *Proceedings of the IEEE International Conference on Robotics and Automation.* 2386–2392.

GERAERTS, R. AND OVERMARS, M. 2006. Creating high-quality roadmaps for motion planning in virtual environments. In *Proceedings of the IEEE/RSJ'06 International Conference on Intelligent Robots and Systems.* 4355–4361.

HSU, D., KINDEL, R., AND LATOMBE, J.-C. 2002. Randomized kinodynamic motion planning with moving obstacles. *Int. J. Robot. Res. 21*, 3, 233–255.

KAMPHUIS, A., MOOIJEKIND, M., NIEUWENHUISEN, D., AND OVERMARS, M. H. 2004. Automatic construction of roadmaps for path planning in games. In *Proceedings of the International Conference on Computer Games*. 285–292.

KAVRAKI, L., LATOMBE, J.-C., SVESTKA, P., AND OVERMARS, M. 1994. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA'94).* 171.

KOVAR, L., GLEICHER, M., AND PIGHIN, F. 2002. Motion graphs. In *ACM SIGGRAPH 2002 Papers.* 473–482.

LAU, M. AND KUFFNER, J. J. 2005. Behavior planning for character animation. In *Proceedings of the Conference of the Science Council of Asia (SCA'05).* ACM Press, 271–280.

LAU, M. AND KUFFNER, J. J. 2006. Precomputed search trees: Planning for interactive goal-driven animation. In *Proceedings of the Conference of the Science Council of Asia (SCA'06)*. Eurographics Association, 299–308.

LEE, J., CHAI, J., REITSMA, P. S. A., HODGINS, J. K., AND POLLARD, N. S. 2002. Interactive control of avatars animated with human motion data. In *ACM SIGGRAPH 2002 Papers*.

LEE, Y., LEE, S. J., AND POPOVIC, Z. 2009. Compact character controllers. In *ACM SIGGRAPH Asia 2009 Papers 28*, 5, 1–8.

LIKHACHEV, M., FERGUSON, D., GORDON, G., STENTZ, A., AND THRUN, S. 2005. Anytime dynamic a*: An anytime, replanning algorithm. In *Proceedings of the International Conference on Autonomous Planning and Scheduling (ICAPS'05).*

LO, W.-Y. AND ZWICKER, M. 2008. Real-Time planning for parameterized human motion. In *Proceedings of the Conference of the Science Council of Asia (SCA'08).* Eurographics Association, 29–38.

MCCANN, J. AND POLLARD, N. 2007. Responsive characters from motion fragments. In *ACM SIGGRAPH 2007 Papers*. ACM Press, 6.

SAFONOVA, A. AND HODGINS, J. K. 2007. Construction and optimal search of interpolated motion graphs. In *ACM SIGGRAPH 2007 Papers*. ACM Press, 106.

SUD, A., GAYLE, R., ANDERSEN, E., GUY, S., LIN, M., AND MANOCHA, D. 2007. Real-Time navigation of independent agents using adaptive roadmaps. In *Proceedings of the Conference on Virtual Reality Software and Technology (VRST'07).* ACM Press, 99–106.

SUNG, M., KOVAR, L., AND GLEICHER, M. 2005. Fast and accurate goal-directed motion synthesis for crowds. In *Proceedings of the Conference of the Science Council of Asia (SCA'05).* ACM Press, 291–300.

TREUILLE, A., LEE, Y., AND POPOVIĆ, Z. 2007. Near-Optimal character animation with continuous control. In *ACM SIGGRAPH 2007 Papers.* ACM Press.

VAN DEN BERG, J., FERGUSON, D., AND KUFFNER, J. 2006. Anytime path planning and replanning in dynamic environments. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA'06).* 2366–2371.

VAN DEN BERG, J., NIEUWENHUISEN, D., JAILLET, L., AND OVERMARS, M. 2005. Creating robust roadmaps for motion planning in changing environments. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'05).* 1053–1059.

VAN DEN BERG, J. AND OVERMARS, M. 2005. Roadmap-Based motion planning in dynamic environments. *IEEE Trans. Robot. 21*, 5, 885–897.

VAN DEN BERG, J. AND OVERMARS, M. 2006. Path planning in repetitive environments. In *Proceedings of the International Conference on Methods and Models in Automation and Robotics (MMAR'06).* 657–662.

VAN DEN BERG, J., PATIL, S., SEWALL, J., MANOCHA, D., AND LIN, M. 2008. Interactive navigation of multiple agents in crowded environments. In *Proceedings of the Symposium on Interactive 3D Graphics (I3D'08).* ACM Press, 139–147.

ZUCKER, M., KUFFNER, J. J., AND BRANICKY, M. S. 2007. Multipartite RRTs for rapid replanning in dynamic environments. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA'07).* 1603–1609.

Received July 2010; revised December 2010; accepted February 2011