In Proceedings of SIGGRAPH 2016 Course Notes, Anaheim, CA. This is the manuscript of the authors and not the final publication.

Geometric and Discrete Path Planning for Interactive Virtual Worlds

Marcelo Kallmann* University of California Merced

Mubbasir Kapadia† Rutgers University

Abstract

Path planning and navigation play a significant role in simulated virtual environments and computer games. Computing collision-free paths, addressing clearance, and designing dynamic representations are examples of important problems with roots in computational geometry and discrete artificial intelligence search methods, and which are being re-visited with innovative new perspectives from researchers in computer graphics and animation. This course provides an overview of algorithms and navigation structures for achieving real-time dynamic navigation for the next generation of multi-agent simulations and virtual worlds. Building on top of classical techniques in computational geometry and discrete search, we review recent developments in real-time planning and discrete environment representations for the efficient computation of paths addressing different constraints in large, complex, and dynamic environments.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Animation;

Keywords: Navigation Meshes, Path Planning, Path Finding, Shortest Paths, Navigation, Discrete Search, Anytime Dynamic Search.

1 Overview

This course reviews planning techniques and navigation structures for the next generation of real-time simulated virtual worlds and computer games. Building on top of classical techniques in computational geometry and discrete search, we review recent developments in real-time planning and environment representation for achieving efficient and robust computation of paths addressing different constraints in large, complex, and dynamic environments. The covered topics target the growing needs for efficient navigation methods in today's increasingly complex simulated virtual worlds. The material of this course is designed for both basic and intermediate level attendees, and is organized in three modules.

Module I: Discrete and Geometric Planning (Section 2) We start reviewing classical Dijkstra and A* discrete search algorithms, the Euclidean Shortest Path problem, and the related main algorithms and data structures from classical Computational Geometry. This introduction motivates and justifies methods recently developed for computing Shortest Path Maps with GPU shaders, and for developing navigation meshes handling clearance and robustness during dynamic updates.

*mkallmann@ucmerced.edu

†mubbasir.kapadia@rutgers.edu

SIGGRAPH '16, July 24-28, 2016, Anaheim, CA

ISBN: 978-1-4503-4289-6/16/07

DOI: http://dx.doi.org/10.1145/2897826.2927310

Module II: Advanced Planning Techniques (Section 3) Departing from classical A* we review anytime and incremental variations of search algorithms for efficiently computing paths in dynamically changing environments while meeting strict time limits and preserving optimality guarantees. We describe extensions that incorporate different types of spatial constraints and that use GPU hardware to provide orders of magnitude speedup. Finally, we describe how the discussed techniques can be put together to solve navigation problems in more complex domains, including behavior planning and multi-agent coordination.

Module III: Planning Techniques for Character Animation (Section 4) In this final part we present examples of how planning algorithms can be applied to solve animation problems beyond path planning. We discuss approaches for multi-agent navigation and for crowd dynamics using different domain representations including triangulations, grids, and bio-mechanically based locomotion models for footstep selection. We then present an overview of planning techniques for whole-body character motion synthesis, including approaches for planning with motion capture data and for addressing locomotion coordination with upper-body actions.

The first version of this course was organized in SIGGRAPH 2014 [Kallmann and Kapadia 2014] in Vancouver, Canada, where it had 199 attendees. Following the good reception of the first version, its original material was significantly revised and expanded into the book "Geometric and Discrete Path Planning for Interactive Virtual Worlds" published by Morgan & Claypool in January 2016 [Kallmann and Kapadia 2016].

In this current SIGGRAPH 2016 version we follow the organization adopted by the book, and we expand the presented topics in module III with new results from our recent research work in the past 2 years. This includes new developments on mesh representations for path planning [Berseth et al. 2015], on full-body motion planning for virtual demonstrators [Huang and Kallmann 2015], and on detecting and embedding contact-rich navigation semantics in environment representations for enhanced character animation [Kapadia et al. 2016].

2 Discrete and Geometric Planning

Modern applications to path planning have become considerably complex and a multitude of factors have to be addressed in combined ways in order to meet with requirements. Nevertheless, some of the underlying problems are still the same as classical well-understood problems long studied by researchers in discrete search and computational geometry areas. This section reviews classical algorithms and data structures with the goal of exposing the nature of the problems and the underlying concepts and solutions. These concepts are important for understanding the properties and limitations of modern approaches, and for designing effective approaches when addressing complex versions of the path planning problem.

2.1 Discrete Search

Given a planar environment, the navigable space of the environment can be decomposed in cells such that the adjacency graph of the cells can be processed by discrete search methods. The process transforms the continuous path planning problem into a dis-

crete graph search problem. For each node of the graph, there are only a discrete number of choices to decide in which direction to move next. The two most notable discrete search algorithms are the Dijkstra and A* algorithms.

2.1.1 Dijkstra Algorithm

Edsger Wybe Dijkstra introduced his algorithm for computing the shortest path between two nodes in 1959 [Dijkstra 1959]. The concept is based on expanding a frontier of equal distance from the source node, expanding outwards from it, step by step, until all nodes are processed or until the goal node is reached. The algorithm is elegant and effective, and when heuristics are not known or not possible, it remains the algorithm of choice for many planning problems.

An efficient implementation of Dijkstra's shortest path algorithm will rely on a priority queue Q able to efficiently maintain the nodes in the expansion front sorted according to their *cost-to-come* values computed so far. The cost-to-come value of a node n is the cost of the lowest-cost path from the source node to n. The sorting ensures that nodes are expanded in order, taking into account positive edge traversal costs, until the goal node is reached and the shortest path to it can be determined.

In the general case, priority queue Q has to process three key operations: insertion, removal, and priority decrease of nodes. The priority value of a node is its cost-to-come value. Let n be a node being inserted in Q with associated priority cost c. The insertion function $Q.insert(n, c)$ will store node n with priority c in Q. Inserted nodes should be kept well organized in Q such that, at any time, a removal call $Q.remove()$ will efficiently remove and return the node of minimal cost among all nodes in Q. Function $Q.decrease(n, c)$ will decrease the priority of n, a node already in Q, to the new priority c.

Algorithm 1 provides an implementation of Dijkstra's algorithm. Priority queue Q sorts the nodes in the expansion frontier according to their current cost-to-come costs, which are stored at the nodes and retrieved with function $g(n)$. Cost function $c(v, n)$ retrieves the positive cost to move from node v to node n. The algorithm receives as input the start node s and the goal target node t. In practice path queries are in the Euclidean plane and will be specified by points and not graph nodes. In this case the input points are converted to their respective nearest graph nodes s and t in the graph, such that **Dijkstra** (s, t) can be then called. See Figure 1 for an example.

Algorithm 1 - Dijkstra Algorithm for Shortest Paths

Input: source node s and goal node t.
Output: shortest path from s to t, or null path if it does not exist.

```
1:  Dijkstra (s, t)
2:  Initialize Q with (s, 0), set g(s) to be 0, and mark s as visited;
3:  while ( Q not empty ) do
4:      v ← Q.remove();
5:      if ( v = t ) return reconstructed branch from v to s;
6:      for all ( neighbors n of v ) do
7:          if ( n not visited or g(n) > g(v) + c(v, n) ) then
8:              Set the parent of n to be v;
9:              Set g(n) to be g(v) + c(v, n);
10:             if ( n visited ) then Q.decrease(n, g(n));
11:             else Q.insert(n, g(n));
12:             Mark n as visited, if not already visited;
13: return null path;
```

Textbooks will often present the Dijkstra algorithm by first inserting in Q all nodes in the graph with an infinite cost, and then costs are gradually reduced as the algorithm progresses. Cost reduction is an operation called *relax* [Cormen et al. 2009], which is equivalent to lines 7–9 in Algorithm 1. Here we rely on marking nodes to distinguish non-processed nodes from already visited ones, eliminating the need to insert all nodes in Q, and having Q to only contain the active frontier being expanded. Once a node is removed from Q its cost will be final because all its predecessors will have been already processed.

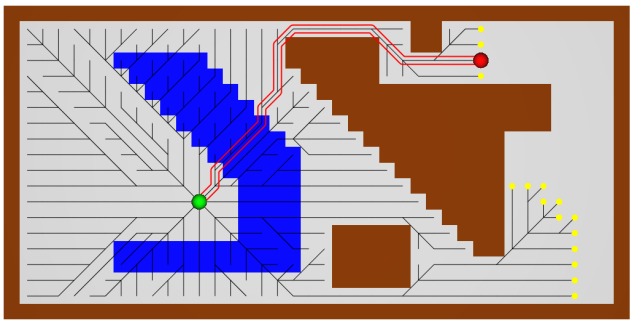


Figure 1: *Example of Dijkstra's algorithm in an 8-connected grid representation. The black segments depict the search tree that was explored until the highlighted solution path was found. The nodes marked as yellow disks represent the nodes in Q at termination time. In this environment brown cells represent obstacles and blue cells have a traversal cost higher than gray cells.*

The overall running time of the Dijkstra algorithm depends on the time taken for each operation in Q, which is usually implemented with a self-balancing binary search tree or with a binary min-heap. Overall the algorithm calls $insert()$ and $remove()$ once per vertex, while $decrease()$ is called up to m times, where m is the number of edges in the graph. Since $insert()$ is called once per vertex, Q will have at most n elements, where n denotes the total number of vertices in the graph. The total running time is therefore $\mathcal{O}((n + m) \log n)$, which reduces to $\mathcal{O}(m \log n)$. With the exception of algorithms seeking for Euclidean shortest paths using visibility graphs (Section 2.2.2), our navigation structures of interest will have $m = \mathcal{O}(n)$ and discrete search will take $\mathcal{O}(n \log n)$.

A good simplification that works well for navigation graphs is to not decrease the key of already inserted nodes and instead just insert a duplicate of the node in Q with the updated cost, eliminating the decrease-key call in line 11. The duplicate nodes in Q with non-updated values can be detected at removal time (line 4) and simply not processed. During the performance of the algorithm the number of elements in Q may become larger; however, the overall running time remains $\mathcal{O}(m \log n)$ and the simplification obtained by not having to perform a decrease-key operation makes the queue implementation simpler and faster.

Another relevant consideration to make is that finding the optimal shortest path in a navigation graph most often does not translate into finding the Euclidean shortest path in the continuous plane the graph represents. Therefore it is worth to consider optimizations even if finding an optimal shortest path in the graph cannot be guaranteed anymore. A useful optimization to the Dijkstra algorithm is to simply not test for the cases that may lead to a decrease-key or duplicate node insertion operation. In this way the algorithm becomes even simpler and the number of elements in Q is reduced. Further analysis and examples are presented in [Kallmann and Kapadia 2016].

2.1.2 A* Algorithm

The A* algorithm [Hart et al. 1968] is an extension of Dijkstra's algorithm which uses heuristics based on knowledge about the specific problem being solved. Because of this strategy, it belongs to a class of algorithms known as heuristic search or informed search. The idea is to expand nodes taking into account their estimated distances to the goal, instead of maintaining a frontier of equal cost from the source node.

A* sorts nodes in Q using a cost function f composed of two parts: $f(n) = g(n) + h(n)$, where $g(n)$ is the same cost-to-come cost of Dijkstra's algorithm, and $h(n)$ is the heuristic cost that estimates the cost of the lowest-cost path from n to the goal node. While implementing an accurate estimate for function h would be difficult, the estimate just needs to be *admissible*; that is, it must not overestimate the cost to reach the goal. For navigation applications a popular simple approach is to encode in $h(n)$ the straight-line distance from n to the goal. Although this estimate is often not accurate, it is admissible, simple to implement, and already provides a useful heuristic. Figure 2 shows an example run using such a heuristic. The heuristic cost is often referred to as the *cost-to-go* cost.

Algorithm 2 presents an implementation of A* following the notation of Algorithm 1. Only the prioritization cost is changed (lines 10-11), where function $f(n) = g(n)+h(n)$ is used instead of $g(n)$ to sort the nodes in Q. This change will expand a frontier of equal lowest-possible overall cost to reach the goal. This results in a frontier where the portions nearest to the goal node t will progress faster towards t, without loosing the optimality of the solution. When the heuristic function is not used ($h(n) = 0$), the algorithm becomes a Dijkstra expansion.

Algorithm 2 - A* Algorithm for Shortest Paths

Input: source node s and goal node t.
Output: shortest path from s to t, or null path if it does not exist.

```
1:  AStar(s, t)
2:  Initialize Q with (s, 0), set g(s) to be 0, and mark s as visited;
3:  while ( Q not empty ) do
4:      v ← Q.remove();
5:      if ( v = t ) return reconstructed branch from v to s;
6:      for all ( neighbors n of v ) do
7:          if ( n not visited or g(n) > g(v) + c(v, n) ) then
8:              Set the parent of n to be v;
9:              Set g(n) to be g(v) + c(v, n);
10:             if ( n visited ) then Q.decrease(n, g(n) + h(n));
11:             else Q.insert(n, g(n) + h(n));
12:             Mark n as visited, if not already visited;
13: return null path;
```

Intuitively, if $h(n) = 0$ for all n the algorithm reverts to the Dijkstra's expansion, which uses no knowledge from the problem domain. It is like any node in the expansion frontier could get to the goal by an arbitrarily low cost edge connection. For problems related to navigation we know that this is not possible and every node will at least have a cost that depends on its straight line distance to the goal in order to reach the goal at some point. Function h encodes that cost. If no more information from the problem domain is available, A* expands no more nodes than any other algorithm that can compute optimal paths. If information is available to improve heuristic estimates, the performance of A* can improve.

In terms of worst-case run time complexity A* performance is equivalent to Dijkstra's algorithm. If the heuristic function h never overestimates the actual minimal cost of reaching the goal, then an optimal solution will be always found with A*. The possible simplifications and optimization discussed for Dijkstra's algorithm are also applicable to A*, meaning that equivalent versions not requiring the decrease-key operation can also be implemented for A*. A good well-informed heuristic can make a significant difference on the performance of A*, and significant research by the AI community has been devoted to heuristic search methods, which are often applied to large grid-based environments. The use of a heuristic cost makes A* outperform Dijkstra in most of the navigation problems encountered in practice, and is thus the algorithm of choice in many applications. Further analysis and examples are presented in [Kallmann and Kapadia 2016].

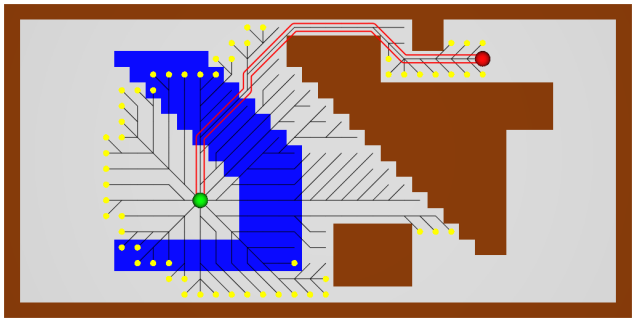


Figure 2: *Example of an A* run in an 8-connected grid representation. The black segments depict the search tree that was explored until the highlighted solution path was found. The nodes marked as yellow disks represent the nodes in Q at termination time. In this example A* clearly expanded less nodes than the Dijkstra run shown in Figure 1.*

2.2 Euclidean Shortest Paths

Euclidean Shortest Paths (ESPs) are paths in the Euclidean plane which have to enforce two properties: they have to be collision-free and they have to be of minimum length. Let n be the number of vertices used to describe a set $\mathcal{S}$ of polygonal obstacles in $\mathbb{R}^2$. Given p and q in $\mathbb{R}^2$, we will say that a path $\pi(p, q)$ exists if π connects both points without crossing any edges of the obstacles in $\mathcal{S}$. If no shorter path exists connecting the two endpoints, the path is then *globally optimal*, it will be here denoted as π^*, and it will be the ESP between p and q.

2.2.1 Shortest Paths in Simple Polygons

The simplest version of the ESP problem is when $\mathcal{S}$ is reduced to a single simple polygon and when p and q are given inside the polygon. In this case, since there are no "holes", all possible paths connecting the two points can be continuously deformed into the globally optimal path π^*. A good concept to use is that of an elastic band that deforms itself to its minimal length state without crossing any edges of $\mathcal{S}$. The result will be $\pi^*(p, q)$. This property eliminates the need for searching among different corridors that could connect p and q, and the efficient *Funnel* algorithm [Chazelle 1982; Lee and Preparata 1984; Hershberger and Snoeyink 1994] can be employed to compute π^* in optimal $O(n)$ time.

The Funnel algorithm assumes that the polygon is first triangulated, what can also be achieved in linear $O(n)$ time [Chazelle 1991; Amato et al. 2000]. In practice however, these optimal triangulation algorithms are difficult to implement and they do not seek for a triangulation of good quality. For these reasons, other triangulation methods such as the Delaunay triangulation (Section 2.3.2) are often preferred even if the computational time increases to at least $O(n \log n)$, depending on the chosen algorithm. Figure 3 exempli-

fies a triangulated simple polygon and the ESP between two points inside it.

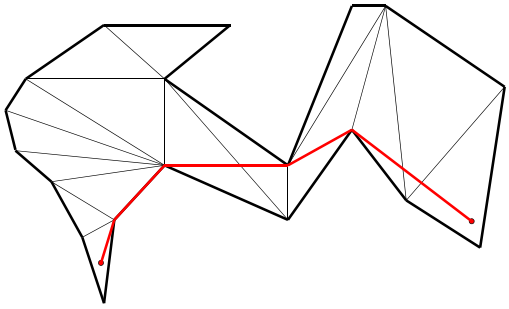


Figure 3: *The Euclidean Shortest Path between two points inside a triangulated simple polygon.*

The Funnel algorithm determines the shortest path inside a simple triangulated polygon in linear time. The original algorithm was designed for obtaining shortest paths for points but we illustrate here the extended version that has been more recently developed in order to take into account a clearance distance r from the edges of the polygon [Kallmann 2010b; Demyen and Buro 2006].

The algorithm performs a sequential expansion of the triangulation edges while maintaining a funnel structure (see Figure 4). Let p be the starting point of the path and u and v be the polygon vertices at the extremities of the funnel (or its *door*). The notation π_r is now used to denote a path that is composed of circle arcs and tangents of distance r to the polygon edges. Paths $\pi_r^*(p, v)$ and $\pi_r^*(p, u)$ may travel together for a while, and at some vertex a they diverge and are concave until they reach the circles centered at u and v. The funnel is the region delimited by segment uv and the concave chains $\pi_r^*(a, v)$ and $\pi_r^*(a, u)$, and a is its apex. The vertices of the funnel are stored in a double-ended queue Q for efficient processing.

Figure 4 illustrates the insertion process of a new vertex w. Points from the v end of Q are popped until b is reached, which is the vertex that will maintain a concave chain to w. If the apex of the funnel is popped during the process, it becomes part of the path so far and the funnel advances until reaching the destination point q. When q is reached it will be then connected to either the apex or one of the boundaries of the funnel in order to finally compose the shortest path, similarly to the original funnel algorithm. In this version with clearance, when clearance values are relatively large it is possible that a new internal turn collapses the boundaries of the funnel. Such situation does not affect the overall logic of the algorithm; however, a specific correction test has to be included each time a new apex a is reached.

2.2.2 Visibility Graphs

While ESPs in simple polygons can be efficiently computed, the generic case is harder because of the many corridors that may exist. Probably the most well-known approach for computing ESPs in generic planar polygonal environments is to build and search the *visibility graph* [Nilsson 1969; Lozano-Pérez and Wesley 1979; De Berg et al. 2008] of the obstacles in $\mathcal{S}$. The visibility graph is composed of all segments connecting vertices that are visible to each other in $\mathcal{S}$. While simple implementations will often rely on $O(n^3)$ or $O(n^2 \log n)$ algorithms, visibility graphs can be computed in $O(n^2)$ time [De Berg et al. 2008] and in the generic case this time cannot be reduced because the graph itself has $O(n^2)$ edges. Graph search algorithms can be then applied to the visibility graph for finding the ESP from p to q after augmenting the graph

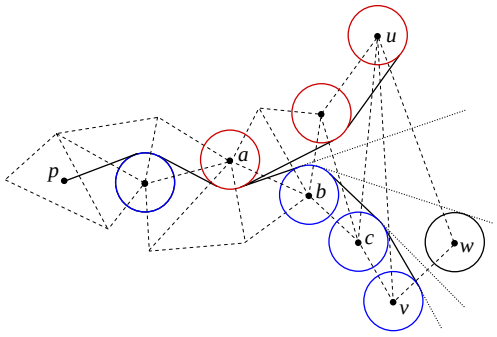


Figure 4: *The r-funnel algorithm. The red circles are centered at the top vertices of the funnel and the blue circles are centered at the bottom vertices.*

with edges connecting all visible vertices to p and q (see Figure 5).

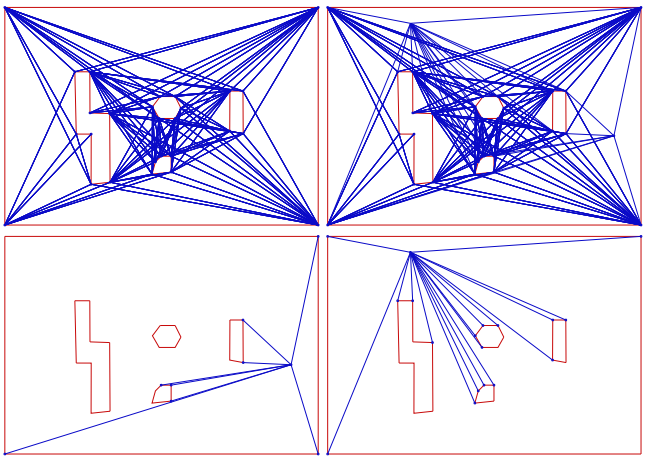


Figure 5: *The Euclidean shortest path between p and q can be found by searching the visibility graph of $\mathcal{S}$ (top-left) augmented by the edges connecting all visible vertices to p and q (top-right). The bottom diagrams show the added edges connecting the visible vertices to p and q.*

Dijkstra and A* graph search algorithms can then be applied to compute the shortest path in the visibility graph. As previously discussed, the typical running time is $O(m \log n)$ time, where m is the number of edges in the graph. In the visibility graph case this time becomes $O(n^2 \log n)$. If Fibonacci heaps are used to speedup priority queue operations, this running time can be reduced $O(n^2)$; however, it is unclear if speed improvements would be observed in practice for typical visibility graph problems.

A simple optimization for reducing the number of edges in the visibility graph is to not include edges that lead to *non-convex* vertices. These edges can be safely discarded because shortest paths will always pass by convex vertices. This optimization can significantly reduce the number of edges, as shown in Figure 6, however the number of edges remain $O(n^2)$.

Visibility graphs represent the most straightforward way to compute ESPs, and extensions for handling arbitrary clearance have also been developed [Chew 1985; Liu and Arimoto 1995; Wein et al. 2007]. The limitation is that searching paths in a graph that may contain a quadratic number of edges is in general too expensive for interactive applications in large environments. This is a

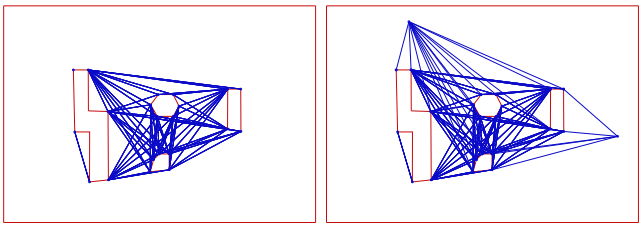


Figure 6: *A popular optimization to reduce the number of edges in the visibility graph is to discard edges that lead to* non-convex *vertices, both when processing the environment (left) and when augmenting the graph with the source and destination points (right).*

consequence of the combinatorial complexity of the ESP problem. The approach taken by the visibility graph is to capture all possible segments that ESPs can have and let graph search algorithms determine the shortest path between two of its nodes. Simpler graphs capturing the free space with $O(n)$ number of edges and nodes may not lead to the ESP when computing the shortest path in the graph.

2.2.3 The Shortest Path Tree

The shortest path tree (SPT) is a structure that is specific to a given source point p. The SPT is the tree formed by all ESPs from p to each reachable vertex in the environment. The SPT is a subgraph of the visibility graph and is usually computed from the visibility graph by running an exhaustive Dijkstra expansion starting from p. A typical implementation is illustrated in Algorithm 3, using the same notation as in Algorithm 2.

Algorithm 3 - SPT Computation by Dijkstra's Expansion

Input: visibility graph and source node s.
Output: shortest path tree rooted at s.

```
1:  SPT(s)
2:  Initialize Q with (s, 0), set g(s) to be 0, and mark s as visited;
3:  while ( Q not empty ) do
4:      v ← Q.remove();
5:      for all ( neighbors n of v ) do
6:          if ( n not visited or g(n) > g(v) + d(v, n) ) then
7:              Set the SPT parent of n to be v;
8:              Set g(n) to be g(v) + d(v, n);
9:              if ( n visited ) then Q.decrease(n, g(n));
10:             else Q.insert(n, g(n));
11:             Mark n as visited, if not already visited;
```

The SPT is specific to a given source point p but it has the advantage that it does not require a graph search to determine paths. Since it is a tree, paths to vertices can be computed by tracing back the tree branch from any given vertex until reaching the root p. See Figure 7. For a generic target point q, the set of visible vertices to q is first computed, and then the shortest path to p can be easily determined by connecting to the visible vertex v that is in the shortest path to p. Vertex v can be quickly determined by storing at every vertex of the SPT the length of its shortest path to the root point, which is what $g(n)$ does in Algorithm 3.

The set of visible vertices from a given point q can be computed in $O(n \log n)$ time using a rotational plane sweep [De Berg et al. 2008]. It can also be computed from a triangulation of the environment by traversing visible adjacent triangles outwards from the one containing q. This is usually efficient in practice because large non-visible areas can be pruned, however the general case will take $O(n^2)$ computation time. A simple efficient variation is to just consider some of the visible points, however loosing the guarantee of finding the shortest path.

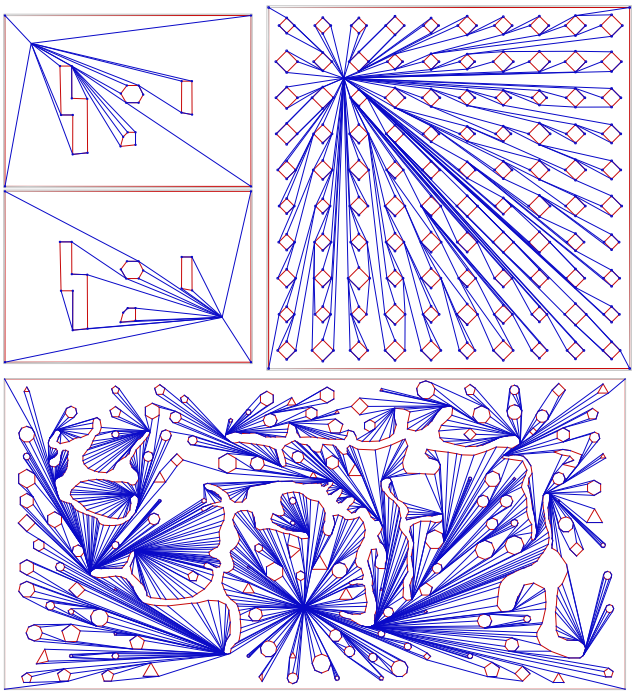


Figure 7: *The shortest path tree contains all ESPs from a source point to each reachable vertex in the environment.*

2.2.4 Continous Dijkstra and the Shortest Path Map

Although visibility graphs may have $O(n^2)$ edges, the ESP problem can be solved in sub-quadratic time by exploring the planarity of the problem. Mitchell provided the first subquadratic algorithms for the ESP problem by introducing the *continuous Dijkstra* paradigm [Mitchell 1991; Mitchell 1993], which simulates the propagation of a continuous wavefront of equal length to the source point, without using the visibility graph. Hershberger and Suri later improved the run time to the optimal $O(n \log n)$ time, while increasing the needed space from $O(n)$ to $O(n \log n)$ [Hershberger and Suri 1997].

The result of the algorithm is the Shortest Path Map (SPM), which is a structure that decomposes the free region in $O(n)$ cells such that it is sufficient to localize the cell of any target point q in order to reconstruct its ESP to the source point p in time proportional to the number of vertices in the shortest path. This is only possible because the cells of the SPM are delimited by not only straight lines but also hyperbolic arcs that result from the wavefront collisions with itself during the SPM construction. See Figure 8.

The SPM can be seen as a generalization of the SPT to every reachable point in the plane, instead of only to the reachable vertices. Although sub-quadratic algorithms exist for computing SPMs and the approach can benefit several applications, the algorithms are not straightforward to implement and they have not yet been much experimented in practice. New GPU techniques recently developed [Camporesi and Kallmann 2014] offer a promising alternative for computing SPMs in applications and were used to generate the examples in Figure 8. Because SPMs encode shortest paths from a source point to all points in the plane, they are in particular interesting when several paths for a same source point are needed.

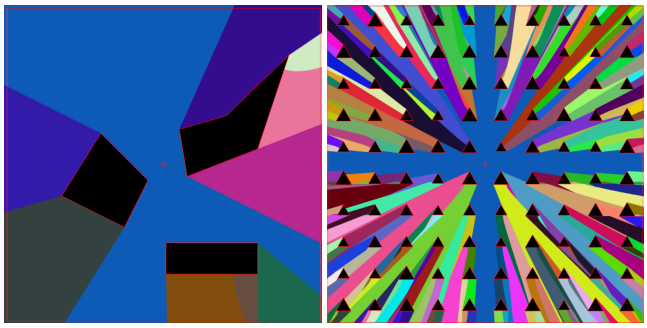


Figure 8: *The shortest path map for a given source point p located at the center of each environment. After the map is computed, the ESP for any given point q in the environment can be found by connecting q to the generator of its region, and then through each parent generator until reaching p.*

2.3 Spatial Partitioning Structures

Given the difficulty in obtaining a simple and efficient approach for solving the ESP problem, practical approaches to path finding have extensively relied on representations that efficiently decompose the free region of the environment according to other useful criteria. These structures however depart from the goal of finding ESPs, and instead provide structures of linear $O(n)$ size that support fast searches for collision-free paths, but which are not necessarily globally shortest ones.

2.3.1 The Voronoi Diagram and the Medial Axis

Probably the most popular classical spatial partitioning structure that is useful for path planning is the Voronoi diagram. The Voronoi diagram is most well-known for a set of seed points in the plane. In this case, the Voronoi diagram will partition the plane in cells and for each seed there will be a corresponding cell consisting of all points closer to that seed than to any other.

The Voronoi diagram can be generalized to a set of seed segments, and in this case the Voronoi cells will delimit the regions of the plane that are closer to each segment. Let's now consider the segments which are the edges delimiting our input obstacles $\mathcal{S}$. The edges of the generalized Voronoi diagram of $\mathcal{S}$ will be the medial axis of $\mathcal{S}$, which is a graph that completely captures all paths of maximum clearance in $\mathcal{S}$. See Figure 9 for an example.

Path planning using the medial axis graph has become very popular. The medial axis is a structure of $O(n)$ size and search methods can therefore efficiently determine the shortest path in the graph after connecting generic source and destination points to their closest points in the graph. The medial axis does not help with finding ESPs but it naturally allows the integration of clearance constraints. Many methods have been proposed for computing paths with clearance by searching the medial axis graph of an environment [Bhattacharya and Gavrilova 2008; Geraerts 2010]. The medial axis can be computed from the Voronoi diagram of the environment in time $O(n \log n)$, and methods based on hardware acceleration have also been developed [Hoff III et al. 2000].

At this point a clear distinction between locally and globally shortest paths can be observed. Consider a path $\pi(p, q)$ determined after computing the shortest path in the medial axis of $\mathcal{S}$. Path π has maximum clearance and therefore it can become shorter with continuous deformations following our elastic band principle, without allowing it to pass over an obstacle edge, and until it reaches its state of shorter possible length. At this final stage the obtained path will be the shortest path in its corridor (or channel) and it can be said to be a *locally* shortest path, which is here denoted as $\pi^l(p, q)$. This is exactly the case of finding the shortest path inside a given polygonal corridor (Figure 3). Path π^l may or not be the globally shortest one π^*, and as we have seen from the previous sections, it is not straightforward to determine if π^l is the globally shortest one without the use of appropriate methods and data structures.

Because in real-time virtual worlds speed of computation is imperative, locally shortest paths have been considered acceptable and practically all the methods reported in the literature have been limited to them. One great benefit of explicitly representing the medial axis in a data structure for path planning is that locally shortest paths can be easily interpolated towards the medial axis in order to reach maximum clearance when needed.

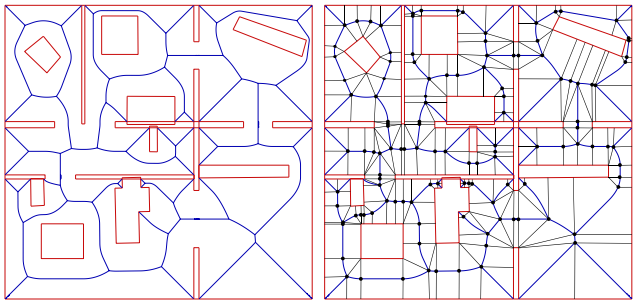


Figure 9: *The medial axis (left) represents the points with maximum clearance in the environment, and its edges can be decomposed such that each edge is associated to its closest pair of obstacle elements (right). The medial axis is composed of line segments and parabolic arcs.*

2.3.2 The Constrained Delaunay Triangulation

Triangulations offer a natural approach for cell decomposition and they have been employed for path planning in varied ways, including to assist with the computation of ESPs. The method of Kapoor et al. (1997) uses a triangulated environment decomposed in corridors and junctions in order to compute the relevant subgraph of the visibility graph for a given ESP query. The method computes ESPs in $O(n + h^2 \log n)$, where h is the number of holes in the environment.

The majority of the methods using triangulations for path planning applications are however limited to simpler solutions based on the Constrained Delaunay Triangulation (CDT) as a cell decomposition for discrete search. CDTs can be defined as follows. Triangulation T will be a CDT of $\mathcal{S}$ if: 1) it enforces obstacle constraints, i.e., all segments of $\mathcal{S}$ are also edges in T, and 2) it respects the *Delaunay criterion* as much as possible, i.e., the circumcircle of every triangle t of T contains no vertex in its interior which is visible from all three vertices of t. One important property of the Delaunay triangulation is that it is the dual graph of the Voronoi diagram. Computing one or the other involves similar work and efficient $O(n \log n)$ algorithms are available. Several textbooks cover the basic algorithms and many software tools are available [The CGAL Project 2014; Tripath Toolkit 2010; Hjelle and Dæhlen 2006].

The CDT is a triangulation that has $O(n)$ cells and therefore discrete search algorithms can compute channels (or corridors) containing locally shortest solutions in optimal times. Since channels are already triangulated, they are ready to be processed by the Funnel algorithm (Figure 4), which has been used in many instances as an efficient way to extract the shortest path inside a channel of a

triangulation [Kallmann 2014; Kallmann 2010b; Demyen and Buro 2006].

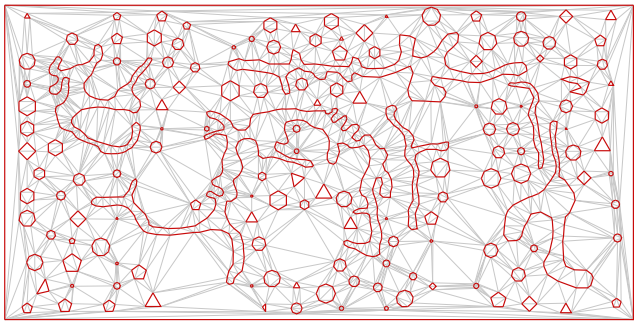


Figure 10: *The constrained Delaunay triangulation provides a $O(n)$ conformal cell decomposition of any polygonal planar environment given as input. Adjacent triangle cells on the free region of the environment form a graph that is thus suitable for path planning.*

Techniques for handling clearance in triangulated environments have also been explored. One simple approach to capture the width of a corridor is to refine constrained edges that have orthogonal projections of vertices on the opposite side of a corridor, adding new free CDT edges with length equal to the width of the corridor [Lamarche and Donikian 2004]. However, such a refinement can only address simple corridors and the total number of vertices added to the CDT can be significant. The more recent Local Clearance Triangulation [Kallmann 2014] shows how a CDT can be refined in order to efficiently compute paths with arbitrary clearance still relying on a triangulation of $O(n)$ cells. LCTs and other structures have been recently proposed as navigation meshes suitable for efficiently computing paths in real-time virtual worlds and they are the topic of the next section.

2.4 Designing Navigation Meshes

The term Navigation Mesh has been mostly used by the computer games community in order to refer to a data structure that is specifically designed for supporting path planning and navigation computations. While the term is well accepted and widely used, no formal definition or expected properties are attached to it. Path finding and navigation have become an important part of video games and navigation meshes have become the most popular approach for designing and implementing a given solution.

Many software tools are available for path planning and most of them can be classified as providing a navigation mesh solution. Examples are: Autodesk Navigation, Spirops, Recast, Path Engine, etc. While there is little information available on the specific techniques employed, most of the approaches are based on computing some sort of graph or mesh structure that covers the free region of the environment, and then path planning is performed with a graph search algorithm running on the structure.

With many new techniques being developed by commercial solutions and by the research community for improving navigation meshes, it is expected that a more formal classification of the achieved properties will become a needed discussion. We provide in this section an overview of basic properties that a navigation mesh structure should observe, and discuss how these properties are addressed in some recent approaches proposed for navigation meshes.

The main function of a navigation mesh is to represent the free environment efficiently in order to allow path queries to be computed in optimal times and to well support other spatial queries useful for navigation. The basic properties that are expected for a navigation mesh are listed below.

- **Linear number of cells.** A navigation mesh should represent the environment with $O(n)$ number of cells or nodes, in order to allow search algorithms to operate on the structure at efficient running times.
- **Quality paths.** A navigation mesh should facilitate the computation of quality paths. Since we accept that ESPs cannot be always found, other guarantees on the type of paths that are computed should be provided. A reasonable expectation is that locally shortest paths should be efficiently provided, and additional definitions of quality may be adopted in the future.
- **Arbitrary clearance.** A navigation mesh should provide an efficient mechanism for computing paths with arbitrary clearance from obstacles. This means that the structure should not need to know in advance the clearance values that will be used. A weaker and less desirable way to address clearance is to pre-compute the environment specifically for each needed clearance value in advance.
- **Representation robustness.** A navigation mesh should be robust to degeneracies in the description of the environment. It should well handle any given description of obstacles, even with undesired self-intersections and overlaps, which often occur from common environment modeling techniques. Being robust is crucial for allowing dynamic updates to occur, in particular when users are allowed to change the environment.
- **Dynamic updates.** A navigation mesh should efficiently update itself to accommodate dynamic changes in the environment. Dynamic updates are crucial for supporting many common events that happen in virtual worlds. Updates can reflect large changes in the environment or small ones, such as doors opening and closing or agents that stop and become obstacles for other agents.

The above properties summarize basic needs that navigation meshes should observe in typical virtual world simulations. We use these properties as a starting point for analyzing and comparing a few approaches for designing navigation meshes.

2.4.1 Methods Based on the Medial Axis

Navigation meshes based on the medial axis are able to well address the desired properties, and should be the approach of choice when the determination of paths with maximal clearance is important. A series of techniques based on the medial axis have been proposed [Geraerts 2010], including algorithms for addressing dynamic updates [van Toll et al. 2012]. The medial axis can be used as a graph for path search (Figure 9-left), and it can also be used to decompose the free space in cells (Figure 9-right) for supporting other navigation queries.

The medial axis is however a structure that is often more complex to be built and maintained than others. It involves straight line segments and parabolic arcs, and not much has been investigated with respect to robustness issues.

2.4.2 Triangulations

The other classical structure that represents an alternative to the medial axis is a triangulation. Triangulations have been extensively employed in the area of finite element mesh generation [Shewchuk 1996] and many algorithms are available. There is also a significant body of research related to addressing robustness [Shewchuk 1997;

Devillers and Pion 2003], which is an important factor that is often overlooked in many methods. Triangulations can also represent environments with less nodes [Kallmann 2014] than the medial axis, they are simpler to be computed, and are some times preferred just because they are in fact a well-known triangle mesh structure.

While triangulations have been used in many ways as a navigation mesh, very few works have achieved useful properties when computing paths directly from a triangulation. The next section describes a recent development that well addresses our desired properties.

2.4.3 Local Clearance Triangulations

One limitation of traditional triangulations is that they cannot easily accommodate arbitrary clearance tests and the recent Local Clearance Triangulation (LCT) has been proposed exactly to overcome this difficulty [Kallmann 2014].

LCTs are computed by refinement operations on a Constrained Delaunay Triangulation of the input obstacle set. The refinements are designed to ensure that two pre-computed local clearance values stored per edge are sufficient to precisely determine if a disc of arbitrary size can pass through any narrow passages of the mesh. This means that at run-time clearance determination is reduced to simple clearance value comparison tests during path search. This property is essential for the correct and efficient extraction of paths with arbitrary clearance directly from the triangulation, without the need to represent the medial axis.

LCTs exactly conform to any given set of polygonal obstacles and dynamic updates and robustness are also addressed. Common degeneracies such as polygon overlaps and intersections can be handled with guaranteed robustness and in a watertight manner. Since LCTs are triangulations, they are well suited for supporting generic navigation and environment-related computations, such as for computing free corridors, visibility, accessibility and proximity queries [Kallmann 2010a].

In comparison to approaches based on the medial axis, LCTs offer a triangular mesh decomposition that carries just enough clearance information to be able to compute paths of arbitrary clearance, without the need to represent the intricate shapes the medial axis can have. Some indicative comparisons have been performed shwoing that the LCT decomposition graph can use about 75% less nodes than a full medial axis representation [Kallmann 2014]. The approach is highly flexible and has been successfully adopted in the video game The Sims 4. See Figure 11 for examples.

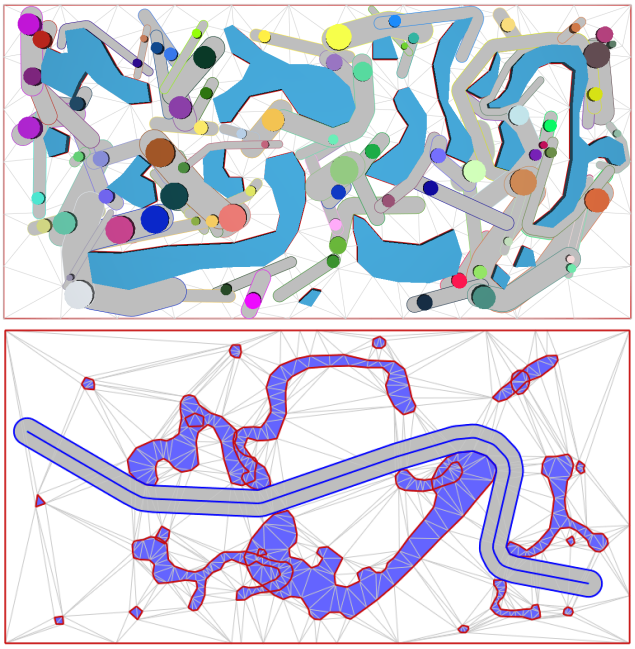


Figure 11: *LCTs efficiently support the computation of paths with arbitrary clearance (top), and well address dynamic updates and robustness to degenerate input such as when obstacles intersect or overlap during dynamic movements (bottom).*

2.5 Other Methods

While LCTs already reduce the number of nodes in the representation when comparing to the full medial axis, it is also possible to devise approaches that further reduce the number of nodes. One example is the Neogen structure, which is based on large almost-convex cells partitioning the free region of a given environment [Oliva and Pelechano 2013a]. By relying on large cells the total number of nodes in the underlying adjacency graph is reduced and paths can be searched in faster times. The structure remains of linear $O(n)$ space. The drawback is that, since the structure is coarser, more computation is needed to compute locally optimal solutions and to guarantee arbitrary clearances. While clearance can be computed, it is not readily available in the structure and pre-computation for each clearance value may be needed.

Several extensions to navigation meshes have also been proposed for allowing the interconnection of floor plans in multi-layer and non-planar environments, in order to address 3D scenes [Lamarche 2009; Jorgensen and Lamarche 2011; Oliva and Pelechano 2013b; van Toll et al. 2011]. Such techniques can however be seen to not be specific to a given choice of navigation mesh representation.

A common factor in all these representations is that they do not address the computation of globally shortest paths. The focus is instead on achieving fast computation of a reasonable collision-free path, and what is reasonable is often not well-defined. The lack of global optimality seems to not be much important, but in any case there is a lack of well-formalized definitions of path quality. It is important to observe that path computation methods will often need to integrate additional properties such as costs of particular regions (high density, undesired terrain, etc.) and behavioral constraints and preferences of the considered agents. Also, reactive behaviors with their own heuristics will always be present to steer the agents during path following. It is expected that navigation meshes will soon provide new solutions to better support these higher-level mechanisms and to better deliver paths suitable for them.

As a final example, path planning is also very useful to guide other types of search methods. There is an extensive body of research on planning motions from motion capture clips based on the concept of *motion graphs* [Kovar et al. 2002; Arikan and Forsyth 2002], which are unstructured graphs automatically built from clips that can be concatenated under a given error metric threshold. The unrolling of the graph in an environment for achieving realistic locomotion involves expensive discrete search on the motion graph, and a common method is to prune all search branches that are far away from a pre-computed path with clearance [Mahmudi and Kallmann 2013]. See Figure 12.

The concept has been later extended with the pre-computation of motion maps built from the motion graph, such that they can be efficiently concatenated to achieve path following at interactive rates from the motion graph data [Mahmudi and Kallmann 2012]. These examples show the importance of the concept of pruning a large

search space with a fast 2D path planning method from a navigation mesh.

The next section reviews discrete search methods suitable for dynamic domains and additional techniques for addressing more advanced versions of the path planning and navigation problem.

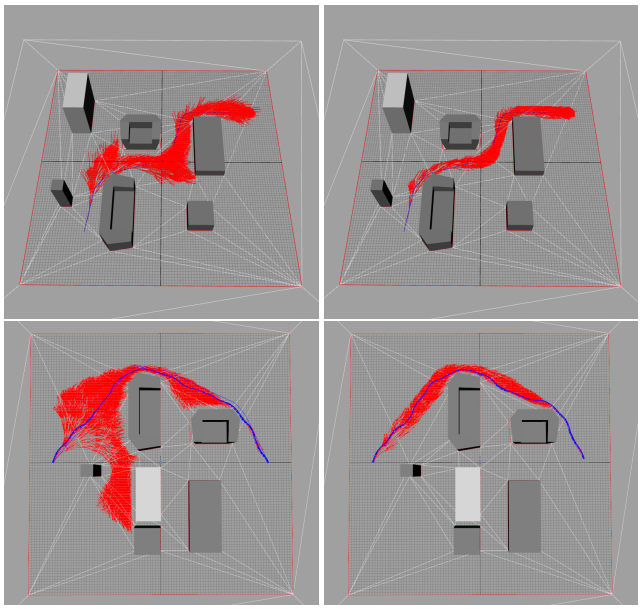


Figure 12: *The left images illustrate the space traversed by a standard discrete search over the connected clips in a motion graph, in order to reach a destination point around obstacles. The examples on the right column show how the search space can be reduced by pruning the branches of the search that become far away from a 2D path efficiently computed from a navigation mesh before the motion graph search expansion starts.*

3 Advanced Planning Techniques

While classical search algorithms are designed to run until completion, anytime and incremental versions are available to meet strict time constraints and to efficiently repair solutions to accommodate dynamic environment changes. We review these algorithms in this section, and we also describe recent extensions that incorporate spatial navigation constraints and exploit massively parallel graphics hardware in order to provide orders of magnitude computational speedups.

3.1 Anytime Dynamic Search

A* search (Algorithm 2) and its variants are extensively studied approaches that generate optimal paths [Hart et al. 1968]. These algorithms process the minimum number of states possible while guaranteeing an optimal solution. Many planning problems [Kapadia and Badler 2013], however, are often too large to solve optimally within an acceptable time, and even minor changes in the obstacle set may severally invalidate the solution, requiring a replan from scratch. In order to handle many planning agents (e.g., crowds), approaches often coarsen the resolution of the search [Kapadia et al. 2009; Kapadia et al. 2012] to meet real-time constraints while sacrificing solution quality. This section introduces fundamental advances in real-time search methods that meet strict time limits and efficiently handle dynamic environments, while preserving bounds on optimality.

3.1.1 Anytime Planning

Anytime planning presents an appealing alternative. Anytime planning algorithms try to find the best plan they can within the amount of time available to them. They quickly find an approximate, and possibly highly suboptimal plan and iteratively improve this plan while reusing previous plan efforts. In addition to being able to meet time deadlines, many of these algorithms also make it possible to interleave planning and execution: while the agent executes its current plan the planner works on improving the plan ahead. A popular anytime version of A* is called Anytime Repairing A* (ARA*) [Likhachev et al. 2003]. This algorithm has control over a suboptimality bound for its current solution, which it uses to achieve the anytime property: it starts by finding a suboptimal solution quickly using a loose bound, then tightens the bound progressively as time allows. Given enough time it finds a provably optimal solution. While improving its bound, ARA* reuses previous search efforts and, as a result, is very efficient.

3.1.2 Incremental Planning

Replanning, or incremental algorithms effectively reuse the results of previous planning efforts to help find a new plan when the problem has changed slightly. D* Lite [Koenig and Likhachev 2002] is an example of an incremental search technique which is widely used in artificial intelligence and robotics. D* performs A* to generate an initial solution, and repairs its previous solution to accommodate world changes by reusing as much of its previous search efforts as possible. As a result, they can be orders of magnitude more efficient than replanning from scratch every time the world model changes.

3.1.3 Anytime Dynamic A*

Anytime Dynamic A* [Likhachev et al. 2005] combines the incremental properties of D* Lite [Koenig and Likhachev 2002] and the anytime properties of ARA* [Likhachev et al. 2003] to efficiently repair solutions after world changes and agent movement. It quickly generates an initial suboptimal plan, bounded by an initial inflation factor which focuses search efforts towards the goal. This initial plan is iteratively improved by reducing the inflation factor until it becomes 1.0, thus guaranteeing optimality in the final solution.

A typical implementation of AD* is presented in Algorithm 4. It performs a backward search and maintains a least cost path from the goal s_{goal} to the start s_{start} by storing the cost estimate $g(s)$ from s to s_{goal}. However, in dynamic environments, edge costs in the search graph may constantly change and expanded nodes may become inconsistent. Hence, a one-step look ahead cost estimate $rhs(s)$ is introduced [Koenig and Likhachev 2002] to determine node consistency. A priority queue `OPEN` contains the states that need to be expanded for every plan iteration, with the priority defined using a lexicographic ordering of a two-tuple **key**(s), defined for each state.

`OPEN` contains only the inconsistent states ($g(s) \neq rhs(s)$) which need to be updated to become consistent. Nodes are expanded in increasing priority until there is no state with a key value less than the start state. A heuristic function $h(s, s')$ computes an estimate of the optimal cost between two states, and is used to focus the search towards s_{start}. Instead of processing all inconsistent nodes, only those nodes whose costs may be inconsistent beyond a certain bound, defined by the inflation factor ϵ are expanded. It performs an initial search with an inflation factor ϵ_0 and is guaranteed to expand each state only once. An `INCONS` list keeps track of already expanded nodes that become inconsistent due to cost changes in

neighboring nodes. Assuming no world changes, ϵ is decreased iteratively and plan quality is improved until an optimal solution is reached ($\epsilon = 1$).

Each time ϵ is decreased, all states made inconsistent due to change in ϵ are moved from INCONS to OPEN with **key**(s) based on the reduced inflation factor, and CLOSED is made empty. This improves efficiency since it only expands a state at most once in a given search and reconsidering the states from the previous search that were inconsistent allows much of the previous search effort to be reused, requiring only a minor amount of computation to refine the solution. **ComputeOrImprovePath** (Algorithm 4 [15–23]) gives the routine for computing or refining a path from s_{start} to s_{goal}. When change in edge costs are detected, new inconsistent nodes are placed into OPEN and node expansion is repeated until a least cost solution is achieved within the current ϵ bounds. When the environment changes substantially, it may not be feasible to repair the current solution and it is better to increase ϵ so that a less optimal solution is reached more quickly.

Algorithm 4 Anytime Dynamic Planner

```
1: function KEY(s)
2:     if (g(s) > rhs(s)) then
3:         return [rhs(s) + ε · h(s, s_goal); rhs(s)];
4:     else
5:         return [g(s) + ·h(s, s_goal); g(s)];

6: function UPDATESTATE(s)
7:     if (s ≠ s_start) then
8:         s' = arg_{s'∈pred(s)} min(c(s, s') · M_C(s, s') + g(s'));
9:         rhs(s) = c(s, s') · M_C(s, s') + g(s');
10:        prev(s) = s';
11:    if (s ∈ OPEN) remove s from OPEN;
12:    if (g(s) ≠ rhs(s)) then
13:        if (s ∉ CLOSED) insert s in OPEN with key(s);
14:        else insert s in INCONS;
15:    Insert s in VISITED;

16: function COMPUTEORIMPROVEPATH(t_max)
17:    while (min_{s∈OPEN}(key(s) < key(s_goal) ∨ rhs(s_goal) ≠
       g(s_goal) ∨ Π(s_start, s_goal) = NULL) ∧ t < t_max do
18:        s = arg_{s∈OPEN} min(key(s));
19:        if (g(s) > rhs(s)) then
20:            g(s) = rhs(s);
21:            CLOSED = CLOSED ∪ s;
22:        else
23:            g(s) = ∞;
24:            UpdateState(s);
```

3.2 Planning with Constraints

While there is extensive literature for computing optimal, collision-free paths, there is relatively little work that explores the satisfaction of additional spatial constraints between objects and agents at the global navigation layer. In this section, we present a planning framework [Kapadia et al. 2013b] that satisfies multiple spatial constraints imposed on the path. The type of constraints specified could include staying behind a building, walking along walls, or avoiding the line of sight of patrolling agents. The anytime-dynamic planner is extended to compute constraint-aware paths, while efficiently repairing solutions to account for varying dynamic constraints or an updating world model. The method can be used on challenging navigation problems in complex environments for dynamic agents using combinations of hard and soft, attracting and repelling constraints, defined by both static and moving obstacles.

3.2.1 Environment Representation

A coarse-resolution representation, such as a triangulation-based method, facilitates efficient search but cannot directly accommodate all constraints due to insufficient resolution in regions of the environment where constraints may be specified. Since dynamic constraints are not known ahead of time, it is difficult to simply increase triangulation density near constraints. It is possible to devise methods for increasing resolution at specific regions when needed, or the alternative is to rely on a dense representation of the environment that can account for all constraints (including dynamic objects), but at the cost of not being efficient in large environments. To avoid these limitations, we propose a hybrid environment representation that has sufficient resolution while still accelerating search computations by exploiting longer, coarser transitions, when possible.

Triangulations. We define a simple triangulated representation of free space in the environment, represented by $\Sigma_{\text{tri}} = \langle \mathbf{S}_{\text{tri}}, \mathbf{A}_{\text{tri}} \rangle$ where elements of $\mathbf{S}_{\text{tri}}$ are the midpoints of the edges in the navigation mesh and elements of $\mathbf{A}_{\text{tri}}$ are the six directed transitions per triangle, two bi-directional edges for each vertex pair. This triangulation can be easily replaced by more complex solutions [Recast 2014; Kallmann 2014], and produces a low-density representation of the state and action space. Figure 13(a) illustrates Σ_{tri} for a simple environment. The triangulation domain Σ_{tri} provides a coarse-resolution discretization of free space in the environment, facilitating efficient pathfinding. However, the resulting graph is too sparse to represent paths adhering to constraints such as spatial relations to an object.

To offset this limitation, we can annotate objects in the environment with additional geometry to describe relative spatial relationships (e.g., Near, Left, Between, etc.). These annotations generate additional triangles in the mesh, which expands Σ_{tri} to include states and transitions that can represent these spatial relations. Annotations, and the corresponding triangulation, are illustrated in Figure 13(b). These annotations are useful for constraints relative to static objects; however, Σ_{tri} cannot account for dynamic objects as the triangulation cannot be efficiently recomputed on the fly. To handle dynamic constraints, we provide a dense graph representation, described below.

Dense Uniform Graph. To generate $\Sigma_{\text{dense}} = \langle \mathbf{S}_{\text{dense}}, \mathbf{A}_{\text{dense}} \rangle$, we densely sample points in the 3D environment, separated by a uniform distance d_{grid}, which represents the graph discretization. For each of these points, we add a state to $\mathbf{S}_{\text{dense}}$ if it is close to the navigation mesh (within $\frac{\sqrt{3}}{2} d_{\text{grid}}$ of the nearest point), and clamp it to that point. Since the graph is sampled in 3D, each state in $\mathbf{S}_{\text{dense}}$ could have a maximum of 26 neighbors; however, in practice, each state has no more than 8 neighbors if the domain operates in a locally-planar environment (such as a game or real-world map). The dense domain Σ_{dense} can be precomputed or generated on the fly, depending on environment size and application requirements. Regardless of how it is implemented, however, a dense domain greatly increases the computational burden of the search due to the increased number of nodes and transitions compared with a sparse domain.

Hybrid Graph. In the first of our two attempts to mitigate the performance problem of Σ_{dense}, we combine Σ_{dense} and Σ_{tri} to generate a hybrid domain $\Sigma_{\text{hybrid}} = \langle \mathbf{S}_{\text{hybrid}} = \mathbf{S}_{\text{dense}}, \mathbf{A}_{\text{hybrid}} = \mathbf{A}_{\text{dense}} \cup \mathbf{A}_{\text{tri}} \rangle$. First, we add all the states and transitions in Σ_{dense}

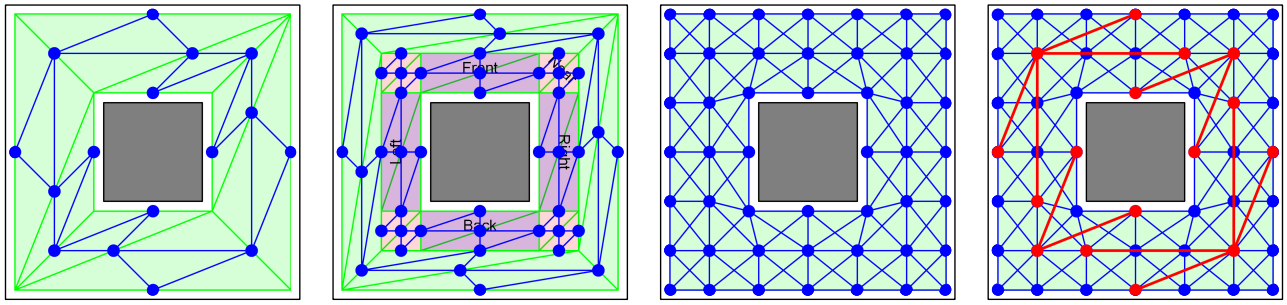


Figure 13: *(a) Environment triangulation Σ_{tri}. (b) Object annotations, with additional nodes added to Σ_{tri}, to accommodate static spatial constraints. (c) Dense uniform graph Σ_{dense}, for same environment. (d) A hybrid graph Σ_{hybrid} of (a) Σ_{tri} and (c) Σ_{dense}; highways are indicated in red.*

to Σ_{hybrid}. For each state in $\mathbf{S}_{\text{tri}}$, we find the closest state in $\mathbf{S}_{\text{dense}}$, creating a mapping between the state sets, $\lambda : \mathbf{S}_{\text{tri}} \rightarrow \mathbf{S}_{\text{dense}}$. Then, for each transition $(s, s') \in \mathbf{A}_{\text{tri}}$, we add the corresponding transition $(\lambda(s), \lambda(s'))$ in $\mathbf{A}_{\text{dense}}$. The resulting hybrid domain Σ_{hybrid} has the same states as Σ_{dense} with additional transitions. These transitions are generally much longer than those in $\mathbf{A}_{\text{dense}}$, creating a low-density network of *highways* through the dense graph.

In Σ_{hybrid}, a pathfinding search can automatically choose highways for long distances, and only use the dense graph when necessary or the planner has additional time to compute an exact plan. As before, the dense graph allows the planner to find paths that adhere to constraints. But when there is no strong influence of nearby constraints, the planner can take highways to improve its performance. In addition, with a planner like AD* [Likhachev et al. 2005], we can inflate the influence of the heuristic to produce suboptimal paths very quickly that favor highway selection, and iteratively improve the path quality by using dense transitions, while maintaining interactive frame rates.

3.2.2 Constraints

Constraints imposed on how an agent navigates to its destination greatly influence the motion trajectories that are produced, and often result in global changes to the paths that cannot be met using existing local solutions. A constraint-aware planning problem is represented by a start state s_{start}, a goal state s_{goal}, and a set of constraints $\mathbf{C} = \{c_i\}$. Each constraint is defined as $c_i = (\texttt{(In|Near) Annotation}_i$ with weight $w_i)$. Despite the simplicity of such a definition, it is important to note its flexibility: both goals and constraints can encode very rich semantics for a wide variety of planning problems. In addition, multiple problem specifications can be chained together to create more complex commands; for example, "move to the street corner, then patrol the alleyway", where "patrol" can be described as a repeating series of commands going back and forth between two points.

Annotations. An annotation is simply a region that allows the user to define the area of influence of a constraint. By attaching annotations to an object in the environment, a user can provide positional information, which are used to semantically construct customized prepositional constraints (for example, "in the grass", "near the wall", or "west of the train tracks"). Common annotations include `Back`, `Front`, `Left`, and `Right`, for a static object in the environment. Annotations can be easily added for dynamic objects as well, for example, to specify an agent's `LineOfSight`. The relationships between multiple objects can be similarly described by introducing annotations such as `Between`. The annotations define the area of influence of the constraint, relative to the position of the object.

Hard Constraints. A hard constraint comprises just one field: an "annotation." This annotation represents an area in which transitions in Σ_{hybrid} are pruned. Hard constraints can only be `Not` constraints; environment regions where an agent is not allowed to navigate. In order to specify hard attracting constraints, we use a sequence of goals that the agent must navigate to (e.g., go behind the building and then to the mailbox).

Soft Constraints. A soft constraint specification consists of three fields: (1) a preposition, (2) an annotation, and (3) the constraint weight. We define two simple prepositions, `Near` and `In` which define the boundaries of the region of influence. The weight defines the influence of a constraint, and can be positive or negative. For example, one constraint may be a weak preference ($w = 1$), while another may be a very strong aversion ($w = -5$) where a negative weight indicates a repelling factor. Weights allow us to define the influence of constraints relative to one another, where one constraint may outweigh another, facilitating the superposition of multiple constraints in the same area with consistent results.

3.2.3 Multiplier Field

Constraints must modify the costs of transitions in the search graph, in order to have an effect on the resulting path generated. The influence of a constraint is defined using a continuous multiplier field $m(\vec{x})$, which denotes the multiplicative effect of the constraint at a particular position $\vec{x}$ in the environment. It is important to note that, due to its continuous nature, the multiplier field can be easily translated to any pathfinding system; it is not specific to graph search representations of pathfinding problems. For a single constraint c, the cost multiplier field $m_c(\vec{x})$ is defined as follows:

$$m_c(\vec{x}) = 1.1^{-W_c(\vec{x})},$$

where $W_c(\vec{x})$ is the constraint weight field defined as a position-dependent weight value for a constraint. For `In` constraints, it has a discrete definition:

$$W_c(\vec{x}) = \begin{cases} w, & \text{if } \vec{x} \in \text{annotation}_c, \\ 0, & \text{otherwise.} \end{cases}$$

For `Near` constraints it provides a soft falloff with a fixed radius of $|w|$ outside of the annotation:

$$W_c(\vec{x}) = w \cdot \max\left(0, \frac{|w| - r_c(\vec{x})}{|w|}\right),$$

where $r_c(\vec{x})$ is the shortest distance between the position $\vec{x}$ and the volume of the annotation on the constraint c. Outside of the fixed radius $|w|$, a `Near` constraint has no effect.

Multiple Constraints. For a set of constraints $\mathbf{C}$, we define the aggregate cost multiplier field,

$$\begin{aligned} m_{\mathbf{C}}(\vec{x}) &= \max\left(1, m_0 \prod_{c \in \mathbf{C}} m_c(\vec{x})\right) \\ &= \max\left(1, 1.1^{W_0 - \sum_{c \in \mathbf{C}} W_c(\vec{x})}\right). \end{aligned}$$

To accommodate attractor constraints which reduce cost, we define a "base" multiplier m_0 or base weight W_0, which is automatically calculated based on the weight values of the constraints in $\mathbf{C}$. This multiplier affects costs even in the absence of constraints, which allows attractors to reduce the cost of a transition, without it ever going below the original (geometric) cost. The resulting cost multiplier is also limited to be greater than or equal to 1, to preserve the optimality guarantees of the planner.

Cost multiplier for a transition. The cost multiplier for a transition $(s \to s')$, given a set of constraints $\mathbf{C}$, is defined as follows:

$$M_{\mathbf{C}}(s, s') = \int_{s \to s'} m_{\mathbf{C}}(\vec{x})\, d\vec{x}.$$

We choose to define this as a path integral because it is generalized to any path, not just a single discrete transition, and because it perfectly preserves cost under any path subdivision. For our graph representations, we estimate the path integral using a four-part Riemann approximation by taking the value of the multiplier field at several points along the transition.

3.2.4 Planning Algorithm

The modified cost of reaching a state s from s_{start}, under the influence of constraints, is computed as follows:

$$g(s_{\text{start}}, s) = g(s_{\text{start}}, s') + M_{\mathbf{C}}(s, s') \cdot c(s, s'),$$

where $c(s, s')$ is the cost of a transition from $s \to s'$, and $M_{\mathbf{C}}(s, s')$ is the aggregate influence of all constraint multiplier fields. This is recursively expanded to produce:

$$g(s_{\text{start}}, s) = \sum_{(s_i, s_j) \in \Pi(s_{\text{start}}, s)} M_{\mathbf{C}}(s_i, s_j) \cdot c(s_i, s_j),$$

which utilizes the constraint-aware multiplier field to compute the modified least-cost path from s_{start} to s, under the influence of active constraints $\mathbf{C}$. States keep track of the set of constraints that influence its cost, which mitigates the need of exhaustively evaluating every constraint to compute the cost of each transition. When the area of influence of a constraint changes, the states are efficiently updated, as described below.

Accommodating Dynamic Constraints. Over time, objects associated with a constraint may change in location, affecting the constraint multiplier field which influences the search. For example, an agent constrained by a `LineOfSight` constraint may change position, requiring the planner to update the plan to ensure that the constraint is satisfied. Each constraint multiplier field $m_c(\vec{x})$ has a region of influence **region**$(m_c, \vec{x})$, which defines the finite set of states $\mathbf{S}_c$ that is currently under its influence. When a constraint c moves from $\vec{x}_{\text{prev}}$ to $\vec{x}_{\text{next}}$, the union of the states that were previously and currently under its region of influence ($\mathbf{S}_c^{\text{prev}} \cup \mathbf{S}_c^{\text{next}}$) are marked as inconsistent (their costs have changed) and they must be updated. Additionally, for states $s \in \mathbf{S}_c^{\text{next}}$, if c is a hard constraint, its cost $g(s) = \infty$. Algorithm 5 provides the pseudo code for **ConstraintChangeUpdate**. The routine **UpdateState**(s) is modified slightly from its original definition to incorporate the multiplier fields during cost calculation.

Algorithm 5 ConstraintChangeUpdate $(c, \vec{x}_{\text{prev}}, \vec{x}_{\text{next}})$

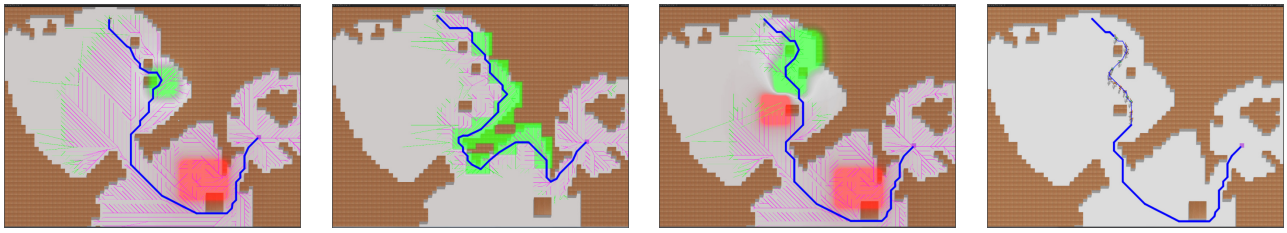
```
1:  S_c^prev = region(m_c, x_prev);
2:  S_c^next = region(m_c, x_next);
3:  for each (s ∈ S_c^prev ∪ S_c^next) do
4:      if (pred(s) ⋂ VISITED ≠ NULL) then
5:          UpdateState(s);
6:      if (s' ∈ S_c^next ∧ c ∈ C_h) then g(s') = ∞;
7:          if (s' ∈ CLOSED) then
8:              for each (s'' ∈ succ(s')) do
9:                  if (s'' ∈ VISITED) then
10:                     UpdateState(s'');
```

3.3 Dynamic Search on the GPU

To meet the growing needs of computationally intensive applications, Kapadia *et al.* [2013] introduce a massively parallelizable, wavefront-based approach to path planning that can exploit graphics hardware to considerably reduce the computational time, while still maintaining strict optimality guarantees. The approach performs efficient updates to accommodate world changes and agent movement, while reusing previous computations. A new termination condition is introduced to ensure that the plans returned are strictly optimal, even on search graphs with non-uniform costs, while requiring minimum GPU iterations. Furthermore, the computational complexity of the approach is independent of the number of agents (traveling to the same goal), facilitating optimal, dynamic path planning for a large number of agents in complex dynamic environments, opening the application to large-scale crowd applications.

3.3.1 Method

The method relies on appropriate data transfer between the CPU and GPU at specific times. In the initial setup, the CPU calls $generateMap(rows, columns)$ which allocates $rows \times columns$ states in the GPU to represent the entire world. Initially, all free states s have an associated cost of -1, $g(s) = -1$, which represents a state that needs to be updated, while obstacles have infinite cost, $g(s) = \infty$. Given an environment configuration with start and goal state(s), ***computePlan*** is executed which repeatedly invokes ***plannerKernel*** (a GPU operation) until a solution is achieved. We keep two copies of the world map: one for reading state costs and the other for writing updated state costs. After each iteration (*i.e.*, kernel execution), the two maps are swapped. This strategy addresses the synchronization issues inherent in GPU programming, by ensuring that the main kernel does not write to the same map used for reads.



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

Figure 14: *Navigation under different constraint specifications. (a) Attractor to go behind an obstacle and a repeller to avoid going in front of an obstacle. (b) Combination of attractors to go along a wall. (c) Combination of attractors and repellers to alternate going in front of and behind obstacles, producing a (d) Lane formation with multiple agents under same constraints.*

Once the planner is done executing, each agent can just follow the least cost path from the goal to its own state to find the generated plan. If an obstacle moves from state s to state s', the GPU map is updated by setting $g(s') = \infty$ and $g(s) = -1$. This means that s' is now an obstacle and the cost for s is invalid and needs to be updated. In addition, the neighbors of s' are checked and marked as inconsistent if they had s' as their least cost predecessor. The planner kernel monitors states that are marked as inconsistent and efficiently computes their updated costs (while also propagating inconsistency) without the need for resetting the entire map. Agent movement (change in start) is also efficiently handled by performing the search in a backward fashion from the goal to the start, and marking the previous state of the agent as inconsistent to ensure a plan update. Algorithm 6 provides a pseudocode for ***computePlan***.

Algorithm 6 ***computePlan***$(*m_{cpu})$

```
1: m_r ← m_cpu;
2: m_w ← m_cpu;
3: while (flag = 0) do
4:     flag ← 0;
5:     plannerKernel(m_r, m_w, flag);
6:     swap (m_r, m_w);
7: m_cpu ← m_r;
```

The wavefront algorithm sets up a map with a initial state which contains an initial cost. At each iteration, every state at the frontier is expanded computing its cost relative to its predecessor's cost. This process repeats until the cost for every state is computed, thus creating the effect of a wave spreading across the map. Wavefront-based approaches are inherently parallelizable, but existing techniques require the entire map to be recomputed to handle dynamic world changes and agent movement. Algorithm 7 describes the shortest path wavefront algorithm ported to the GPU. The planner first initializes the cost of every traversable state to a default value, $g(s) = -1$, indicating it needs to be updated. States occupied by obstacles take a value of infinity, $g(s) = \infty$, and the goal state is initialized with a value of 0, $g(s) = 0$. The planner finds the value g of reaching any state s from the *goal* by launching a kernel at each iteration that computes $g(s)$ as follows:

$$g(s) = min_{s' \in succ(s) \wedge g(s') \geq 0}(c(s, s') + g(s')),$$

where $0 \leq c(s, s') \leq \infty$ is the cost of traversing from state s to s', and is used to encode regions of the environment which should be avoided (*e.g.*, rough terrain, dangerous areas), in addition to obstacles that cannot be traversed. This process continues until all states have been updated at which point the planner terminates execution. To address the concurrency problem inherent in a massively parallel application, two maps are used, one as read-only m_r and the other write-only m_w. Each thread in the kernel reads the necessary values to calculate the cost of its corresponding state from m_r, and writes it to its given state in m_w. This ensures that the map being read will not change while the kernel is executing. Once the kernel finishes execution, m_r and m_w are swapped, thus allowing the threads to read the most recent map while preventing race conditions.

Algorithm 7 ***plannerKernel***$(*m_r, *m_w, *flag)$

```
s ← threadState;
if (s ≠ obstacle ∧ s ≠ goal) then
    for each (s' in neighbor(s)) do
        if (s' ≠ obstacle) then
            newg ← g(s') + c(s, s');
            if ((newg < g(s) ∨ g(s) = −1) ∧ g(s') > −1) then
                pred(s) ← s';
                g(s) ← newg;
                evaluate_termination_condition;
```

The kernel also takes as a parameter a `flag` which is set depending on the termination condition used. A new termination condition is used that can greatly reduce the number of iterations required to find an optimal plan in large environments with non-uniform search graphs. If at any iteration, it is found that the minimum g-value expanded corresponds to that of the agent, this means that a path to that agent is available and any other possible path would yield a higher cost. To make sure that the agent state is expanded at each iteration (to compare to the other states expanded), a g-value of -1 if given to it before each kernel run, marking it as a state that needs to be updated. To implement this strategy, it is enough to just adjust the condition that would set the `flag` that terminates the execution:

$$\textbf{if } (g(s) < g(start) \vee g(agent) = -1) \texttt{ flag} = 1.$$

Once the planner has finished executing, an agent can simply generate its plan by following the reverse of the least-cost path from the goal to its position.

3.3.2 Efficient Plan Repair for Dynamic Environments and Moving Agents

To handle obstacle movements, inconsistent states are identified and resolved. A state is inconsistent if its predecessor is not the neighbor with lowest cost or if its successor is inconsistent. If an obstacle moves from state s to state s', then new costs are set with $g(s') = \infty$ and $g(s) = -1$, which marks s for update. A kernel is then run (Algorithm 8) which sets the g-value of every inconsistent state to -1. A state s with predecessor s' is inconsistent if $g(s) \neq g(s') + c(s, s')$. This kernel is executed repeatedly until all inconsistencies are resolved and is detected when there are no

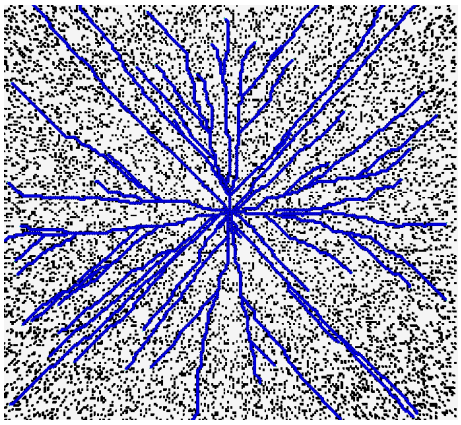


Figure 15: *Complex* 512×512 *with* 200 *agents. Goal is in center of map, and computed paths are shown in blue. Every black area is an obstacle.*

updates performed by any thread during an iteration. Algorithm 8 is triggered when environment changes are detected to ensure that node inconsistency is propagated and resolved in the entire map. Keep in mind that the following code will run in parallel, and that all read and write operations are done in two distinct maps.

Algorithm 8 Algorithm to propagate state inconsistency

$s \leftarrow threadState$;
if $(pred(s) \neq NULL)$ **then**
  if $(g(s) == obstacle \vee pred(s) == obstacle \vee g(s) \neq g(pred(s)) + c(s, s'))$ **then**
    $pred(s) = NULL$;
    $g(s) = -1$;
    $incons = \texttt{true}$;

Handling agent movement is straightforward. For the non-optimized planner, the cost to reach every state has already been computed, so the agent would only require to reconstruct its path again. In the case of the optimized version of the planner it is necessary to run the planner again so that any state between the goal and the new agent position that has not been expanded gets a chance to update its cost.

3.4 Multi-Agent Planning

The kernel is executed until all agent states have been reached, and the maximum g-value of a state that is occupied by an agent is less than the g-value of any other state that was updated during the current iteration, as captured by the following Boolean expression:

$$((g(s) < max_{a_i \in \{a\}} g(a_i)) \vee (g(a_i) = -1 \forall a_i \in \{a\})).$$

The number of iterations for convergence depends on the distance from the goal to the farthest agent. When the map is updated, each agent simply follows the least cost path from the goal to its position to find an optimal path. Note that the approach requires no additional computational cost to handle many agents, provided they share the same goal.

3.4.1 Adaptive Resolution Grids

To address the memory limitations associated with uniform grids [Mubbasir Kapadia and Badler 2013], the environment can be adaptively discretized into variable resolution quads, using finer resolution only where necessary, thus significantly reducing the size of the state space. This also reduces considerably the memory footprint allowing us to handle very large environments in GPU memory and also accelerates the search process. Using an adaptive environment representation on the GPU has two main challenges: indexing is no longer constant time and handling dynamic changes is computationally expensive, since the number and location of neighbor quads varies as obstacles move in the world. Given these properties, it is impossible to know ahead of time how many neighbors a given quad has, which ones those neighbors are, and how many quads are needed to represent the state space. In addition, dynamically allocating memory on the GPU to accommodate these changes can be very expensive. These challenges are addressed by: (1) using a *quadcode* scheme for performing efficient indexing, update, and neighbor finding operations on the GPU, and (2) efficiently handling dynamic environment changes by performing local repair operations on the quad tree, and performing plan repair to resolve state inconsistencies without having to plan from scratch. Please refer to [Garcia et al. 2014] for a detailed description of this method.

4 Planning Techniques for Character Animation

In this section the techniques and principles described in the previous sections are applied to more complex planning problems, demonstrating the generality and applicability of real-time planning techniques for interactive virtual worlds. We start by exploring the use of multiple heterogenous domains of control to solve more challenging navigation problems with space-time constraints at interactive rates. Section 4.3 then demonstrates the use of planning for interactive narrative. Additional applications showcasing the use of planners for complex problem domains include footstep-based planning for crowds [Singh et al. 2011], path planning for coherent and persistent groups [Huang et al. 2014], and multi-actor behavior synthesis [Kapadia et al. 2011].

4.1 Multi-Domain Planning

The problem domain of interacting autonomous agents in dynamic environments is extremely high-dimensional and continuous, with infinite ways to interact with objects and other agents. Having a rich action set, and a system that makes intelligent action choices, facilitates robust, intelligent virtual characters, at the expense of interactivity and scalability. Greatly simplifying the problem domain yields interactive virtual worlds with hundreds and thousands of agents that exhibit simple behavior. The ultimate, far-reaching goal is still a considerable challenge: a real-time system for autonomous character control that can handle many characters, without compromising control fidelity.

One approach for solving such problems has been presented by a real-time planning framework for multi-character navigation that enables the use of multiple heterogeneous problem domains of differing complexities for navigation in large, complex, dynamic virtual environments [Kapadia et al. 2013a]. The original navigation problem is decomposed into a set of smaller problems that are distributed across planning tasks working in these different domains. An anytime dynamic planner is used to efficiently compute and repair plans for each of these tasks, while using plans in one domain to focus and accelerate searches in more complex domains. Using

multiple heterogeneous domains enables the solution of many challenging multi-agent scenarios in complex dynamic environments requiring space-time precision and explicit coordination between interacting agents, by accounting for dynamic information at all stages of the decision-making process. Figure 18 illustrates some of the challenging scenarios and their solutions.

4.1.1 Multiple Domains of Control

The 4 domains defined below provide a nice balance between global static navigation and fine-grained space-time control of agents in dynamic environments. Figure 16 illustrates the different domain representations for a given environment. The static navigation mesh domain Σ_1 uses a triangulated representation of free space and only considers static immovable geometry. Dynamic obstacles and agents are not considered in this domain. The Dynamic Navigation Mesh Domain Σ_2 also uses triangulations to represent free spaces and coarsely accounts for dynamic properties of the environment to make a more informed decision at the global planning layer. A time-varying density field is defined to store the density of moveable objects (agents and obstacles) for each polygon in the triangulation at some point of time.

The Grid Domain Σ_3 discretizes the environment into grid cells where a valid transition is considered between adjacent cells that are free (diagonal movement is allowed). An agent is modeled as a point with a radius (orientation and agent speed is not considered in this domain). This domain only accounts for the current position of dynamic obstacles and agents, and cannot predict collisions in space-time. The cost and heuristic are distance functions that measure the Euclidean distance between grid cells. The Space-Time Domain Σ_4 models the current state of an agent as a space-time position with a current velocity The domains described here are *not* a comprehensive set and only serve to showcase the ability of this framework to use multiple heterogeneous domains of control in order to solve difficult problem instances at a fraction of the computation cost.

4.1.2 Problem Decomposition

Figure 17(a) illustrates the use of tunnels to connect each of the 4 domains, ensuring that a complete path from the agents initial position to its global target is computed at all levels. Figure 17(b) shows how Σ_2 and Σ_3 are connected by using successive waypoints in $\Pi(\Sigma_2)$ as start and goal for independent planning tasks in Σ_3. This relation between Σ_2 and Σ_3 allows finer-resolution plans being computed between waypoints in an independent fashion. Limiting Σ_3 (and Σ_4) to plan between waypoints instead of the global problem instance ensures that the search horizon in these domains is never too large, and that fine-grained space-time trajectories to the initial waypoints are computed quickly. However, completeness and optimality guarantees are relaxed as Σ_3, Σ_4 never compute a single path to the global target.

Σ_1 is first used to compute a path from start to goal, ignoring dynamic obstacles and other agents. $\Pi(\Sigma_1)$ is used to accelerate computations in Σ_2, which refines the global path to factor in the distribution of dynamic objects in the environment. Depending on the relationship between Σ_2 and Σ_3, a single planning task or multiple independent planning tasks are used in Σ_3. Finally, the plan(s) of $T(\Sigma_3)$ are used to accelerate searches in Σ_4.

Changes trigger plan updates which are propagated through the task dependency chain. $T(\Sigma_2)$ monitors plan changes in $T(\Sigma_1)$ as well as the cumulative effect of changes in the environment to refine its path. Each $T(\Sigma_3)$ instance monitors changes in the waypoints along $\Pi(\Sigma_2)$ to repair its solution, as well as nearby changes in obstacle and agent position. Finally, $T(\Sigma_4)$ monitors plan changes

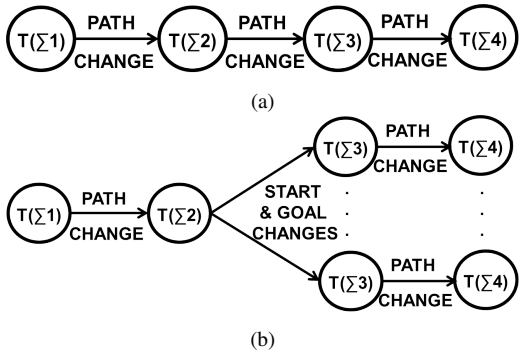


Figure 17: *Relationship between domains. (a) Use of tunnels to connect each of the 4 domains. (b) Use of successive waypoints in $\Pi(\Sigma_2)$ as start, goal pairs to instantiate multiple planning tasks in Σ_3 and Σ_4.*

in $T(\Sigma_3)$ (which it depends on) and repairs its solution to compute a space-time trajectory that avoids collisions with static and dynamic obstacles, as well as other agents.

Events are triggered (outgoing edges) and monitored (incoming edges) by tasks, creating a cyclic dependency between tasks, with $\mathtt{T}_0$ (agent execution) monitoring changes in the plan produced by the particular $T(\Sigma_4)$, which monitors the agents most imminent global waypoint. Tasks that directly affect the agent's next decision, and tasks with currently invalid or sub-optimal solutions are given higher priority. Given the maximum amount of time to deliberate $t_{\max}$, the agent pops one or more tasks that have highest priority and divides the deliberation time across tasks (most imminent tasks are allocated more time). Task priorities constantly change based on events triggered by the environment and other tasks.

4.1.3 Domain Mapping

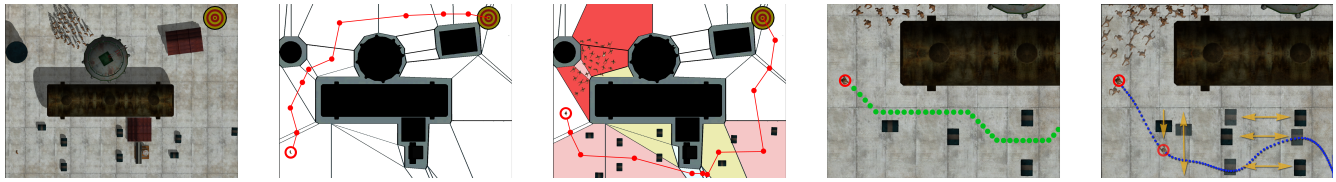
$\lambda(s, \Sigma, \Sigma^{'})$ is defined as an $1 : n$ function that allows us to map states in $\mathbf{S}(\Sigma)$ to one or more equivalent states in $\mathbf{S}(\Sigma^{'})$:

$$\lambda(s, \Sigma, \Sigma^{'}) : s \rightarrow \{s' | s' \in \mathbf{S}(\Sigma^{'}) \wedge s \equiv s'\}. \quad (1)$$

Mapping functions are defined specifically for each domain pair. For example, $\lambda(s, \Sigma_1, \Sigma_2)$ maps a polygon $s \in \mathbf{S}(\Sigma_1)$ to one or more polygons $\{s' | s' \in \mathbf{S}(\Sigma_2)\}$ such that s' is spatially contained in s. If the same triangulation is used for both Σ_1 and Σ_2, then there exists a one-to-one mapping between states. Similarly, $\lambda(s, \Sigma_2, \Sigma_3)$ maps a polygon $s \in \mathbf{S}(\Sigma_2)$ to multiple grid cells $\{s' | s' \in \mathbf{S}(\Sigma_3)\}$ such that s' is spatially contained in s. $\lambda(s, \Sigma_3, \Sigma_4)$ is defined as follows:

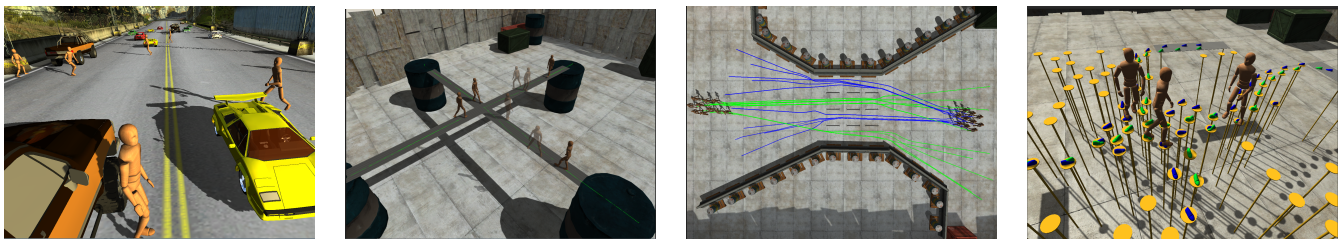
$$\lambda(s, \Sigma_3, \Sigma_4) : (\mathbf{x}) \rightarrow \{(\mathbf{x} + W(\Delta\mathbf{x}), t + W(\Delta t))\}, \quad (2)$$

where $W(\Delta)$ is a window function in the range $[-\Delta, +\Delta]$. The choice of t is important in mapping Σ_3 to Σ_4. Since λ effectively maps a plan $\Pi(\Sigma_3, s_{\text{start}}, s_{\text{goal}})$ in Σ_3 to a tunnel in Σ_4, it can exploit the path and the temporal constraints of s_{start} and s_{goal} to define t for all states along the path. This is achieved by calculating the total path length and the time to reach s_{goal}. This allows the computation of the approximate time of reaching a state along the path, assuming the agent is traveling at a constant speed along the path.



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

Figure 16: *(a) Problem definition with initial configuration of agent and environment. (b) Global plan in static navigation mesh domain Σ_1 accounting for only static geometry. (c) Global plan in dynamic navigation mesh domain Σ_2 accounting for cumulative effect of dynamic objects. (d) Grid plan in Σ_3. (e) Space-time plan in Σ_4 that avoids dynamic threats and other agents.*



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

Figure 18: *Different scenarios. (a) Agents crossing a highway with fast moving vehicles in both directions. (b) 4 agents solving a deadlock situation at a 4-way intersection. (c) 20 agents distributing themselves evenly in a narrow passage, to form lanes both in directions. (d) A complex environment requiring careful foot placement to obtain a solution.*

Mapping Successive Waypoints to Independent Planning Tasks. Successive waypoints along the plan from one domain can be used as start and goal for a planning task in another domain. This effectively decomposes a planning problem into multiple independent planning tasks, each with a significantly smaller search depth.

Consider a path $\Pi(\Sigma_2) = \{s_i | s_i \in \mathbf{S}(\Sigma_2), \forall i \in (0, n)\}$ of length n. For each successive waypoint pair (s_i, s_{i+1}), a planning problem $\mathbf{P}_i = \langle \Sigma_3, s_{\text{start}}, s_{\text{goal}} \rangle$ is defined such that $s_{\text{start}} = \lambda(s_i, \Sigma_2, \Sigma_3)$ and $s_{\text{goal}} = \lambda(s_{i+1}, \Sigma_2, \Sigma_3)$. Even though λ may return multiple equivalent states, only one candidate state is chosen. For each problem definition $\mathbf{P}_i$, an independent planning task $T(\mathbf{P}_i)$ is instantiated which computes and maintains path from s_i to s_{i+1} in Σ_3. Figure 17 illustrates this connection between Σ_2 and Σ_3.

Tunnels. The work in [Gochev et al. 2011] observes that a plan in a low dimensional problem domain can often be exploited to greatly accelerate high-dimensional complex planning problems by focusing searches in the neighborhood of the low dimensional plan. They introduce the concept of a tunnel $\tau(\Sigma_{hd}, \Pi(\Sigma_{ld}), t_w)$ as a sub graph in the high dimensional space Σ_{hd} such that the distance of all states in the tunnel from the low dimensional plan $\Pi(\Sigma_{ld})$ is less than the tunnel width t_w. Based on their work, this approach uses plans from one domain in order to accelerate searches in more complex domains with much larger action spaces. A planner is input a low dimensional plan $\Pi(\Sigma_{ld})$ which is used to focus state transitions in the sub graph defined by the tunnel $\tau(\Sigma_{hd}, \Pi(\Sigma_{ld}), t_w)$.

To check if a state s lies within a tunnel $\tau(\Sigma_{hd}, \Pi(\Sigma_{ld}), t_w)$ without precomputing the tunnel itself, the low dimensional plan $\Pi(\Sigma_{ld})$ is first converted to a high dimensional plan $\Pi'(\Sigma_{hd}, s_{\text{start}}, s_{\text{goal}})$ by mapping all states of Π to their corresponding states in Π', using the mapping function $\lambda(s, \Sigma_{ld}, \Sigma_{hd})$ as defined in Equation 1. Note that the resulting plan Π' may have multiple possible trajectories from s_{start} to s_{goal} due to the $1:n$ mapping of λ. A distance measure $\mathbf{d}(s, \Pi(\Sigma))$ computes the distance of s from the path $\Pi(\Sigma)$. During a planning iteration, a state is generated if and only if $\mathbf{d}(s, \Pi(\Sigma_{hd})) \leq t_w$. This is achieved by redefining the $\texttt{succ}(s)$ and $\texttt{pred}(s)$ to only consider states that lie in the tunnel. Furthermore, node expansion can be prioritized to states that are closer to the path by modifying the heuristic function with:

$$h_t(s, s_{\text{start}}) = h(s, s_{\text{start}}) + |\mathbf{d}(s, \Pi(\Sigma))|. \quad (3)$$

4.2 Precomputed Environment Semantics for Dexterous Character Animation

The widespread availability of high-quality motion capture data and the maturity of solutions to animate virtual characters has paved the way for the next generation of interactive virtual worlds exhibiting intricate interactions between characters and the environments they inhabit. However, current motion synthesis techniques have not been designed to scale with complex environments and contact-rich motions, requiring environment designers to manually embed motion semantics in the environment geometry in order to address online motion synthesis. We introduce an automated approach for analyzing both motions and environments in order to represent the different ways in which an environment can afford a character to move. We extract the salient features that characterize the contact-rich motion repertoire of a character and detect valid transitions in the environment where each of these motions may be possible, along with additional semantics that inform which surfaces of the environment the character may use for support during the motion. The precomputed motion semantics can be easily integrated into standard navigation and animation pipelines in order to greatly enhance the motion capabilities of virtual characters. The computational efficiency of our approach enables two additional applications. Environment designers can interactively design new environments and get instant feedback on how characters may potentially interact, which can be used for iterative modeling and refinement. End users can dynamically edit virtual worlds and characters will

automatically accommodate the changes in the environment in their movement strategies.

4.2.1 Overview

Fig. 19 illustrates the main steps of our framework. Given raw motion data and geometry data of the environment, our system analyzes both of them in order to find valid motion transitions in the environment. Motion transitions are then annotated in the environment combined with standard navigation mesh representations for runtime pathfinding and motion synthesis.

Motion Analysis. Given a database of motion clips representing different motion skills for a character, we analyze each motion (or a set of similar motions) to define its *motion signature* – which characterizes the different contacts between the character's body parts and the environment during the motion, the spatial constraints between pairs of contacts, as well as the collision bounds of the character while performing the motion.

Environment Analysis. Given an arbitrarily complex 3D environment, our system identifies surfaces that can be used to support the character using different kinds of contacts. For each motion signature (a sequence of contact configurations), a corresponding sequence of proximal contact surfaces is identified which satisfy the spatial constraints between the contacts in the motion. We use a "projection and intersection" based method that efficiently identifies spatial relationships between surfaces and can handle deformable motion models. The set of valid surfaces are pruned to ensure that the resulting motion performed in that part of the environment is collision-free, producing an annotated environment that identifies areas where different motion skills are possible.

Runtime Pathfinding and Motion Synthesis. We extend traditional navigation graph approaches to include additional edges for connecting disconnected surfaces using a particular motion skill, with semantics that codify motion, contact, and collision constraints. During runtime, standard search techniques can be used on this navigation graph to generate a path that includes these behaviors. A data-driven animation system is used to animate characters to navigate along this path, and the annotated motion transitions are used to animate the character to perform complex animations that satisfy the exact contact constraints without the need for expensive configuration-space motion planning.

4.2.2 Motion Analysis

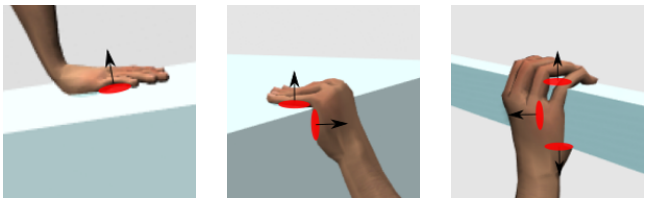
The movement repertoire of an animated character is defined by a set of motion skills $\mathbf{m} \in \mathbf{M}$. A motion skill may be a single motion clip or a set of similar motion clips which can be blended together to create a parameterized motion skill that can produce new contact variations that are not present in the original motions.

Each motion skill is abstracted as $\mathbf{m} = \langle \mathbf{G}, \mathbf{V}, \mathbf{L} \rangle$ where: (1) the contact constraint graph $\mathbf{G} = \langle \mathbf{C}, \mathbf{E} \rangle$ is defined by a set of contacts $\mathbf{c} \in \mathbf{C}$, and spatial constraints $\mathbf{e}(i,j) \in \mathbf{E}$ between pairs of contacts $(\mathbf{c_i}, \mathbf{c_j}) \in \mathbf{C}$. (2) $\mathbf{V}$ represents the collision bounds of the character over the course of the motion. (3) $\mathbf{L}$ is a data-driven controller that is used for animating the character and may be a single motion clip coupled with inverse kinematics, or a parameterized blend tree of similar motions.

Contact Constraint Graph

Over the course of the motion, different parts of the characters body might come in contact with different surfaces of the environment. We define a contact pose $\mathbf{P} = \{\mathbf{c_i}\}$ as a body part (e.g., hands or feet) coming in contact with one or more surfaces in the environment. Each contact $\mathbf{c} = \{\mathbf{p}, \mathbf{n}\}$ denotes the contact of a body part with one environment surface, defined using the relative position $\mathbf{p}$ and surface normal $\mathbf{n}$ of the contact, with respect to the initial position of the center of mass of the character. Fig. 20 illustrates example contact poses.

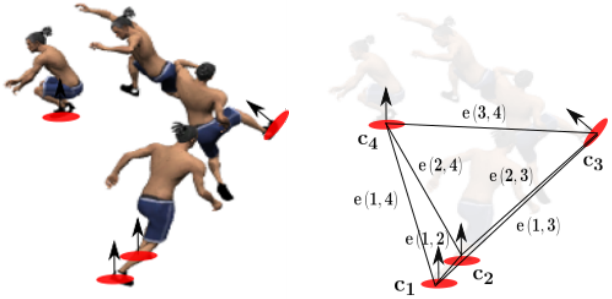
A constraint $\mathbf{e}(i,j) \in \mathbf{E}$ defines the distance interval (d_{min}, d_{max}) between the two contacts $\mathbf{c_i}, \mathbf{c_j} \in \mathbf{C}$. For a motion skill that is a single motion clip without any procedural deformation, this will be a single numeric value. We do not consider temporal constraints as they are not necessary in a static snapshot of an environment.



(a) (b) (c)

Figure 20: *Contact Poses. (a) Planar contact pose. (b) L-shape contact pose: two surfaces with angle between normals close to 90 degrees. This corresponds to placing hands or other body part on a perpendicular edge in the environment. (c) Grasping contact: three planar surfaces with normals spanning 180 degrees, corresponding to placing hands on a curved surface.*

For each motion, we identify the set of keyframes during which a contact takes place using a semi-automated method, as described in [Xiao et al. 2006]. A keyframe may have multiple body parts in contact at a time, leading to simultaneous contact poses. Fig. 21 illustrates the keyframes of selected motions with their contacts highlighted. Fig. 22 illustrates an example of a contact constraint graph for a tic-tac motion. Our constraint graph allows to represent constraints between all pairs of contacts. This is useful to specify global constraints, for example, such as lower and upper limits between the first and last contacts for the behavior to be executed.



(a) (b)

Figure 22: *Contact Constraint Graph for a tic-tac motion. The nodes represent the contacts* $\mathbf{c} \in \mathbf{C}$ *over the course of the motion, and the edges represent spatial constraints* $\mathbf{e}(.,.) \in \mathbf{E}$ *between contacts.*

4.2.3 Environment Analysis

Once the motions have been processed we analyze the environment to automatically detect locations where motion skills may occur.

Contact Surface Detection

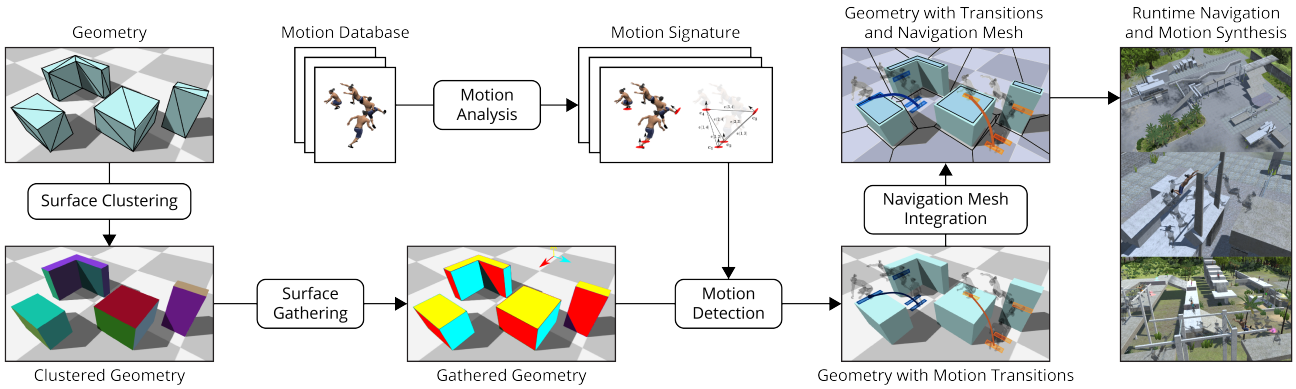


Figure 19: *Framework Overview.*

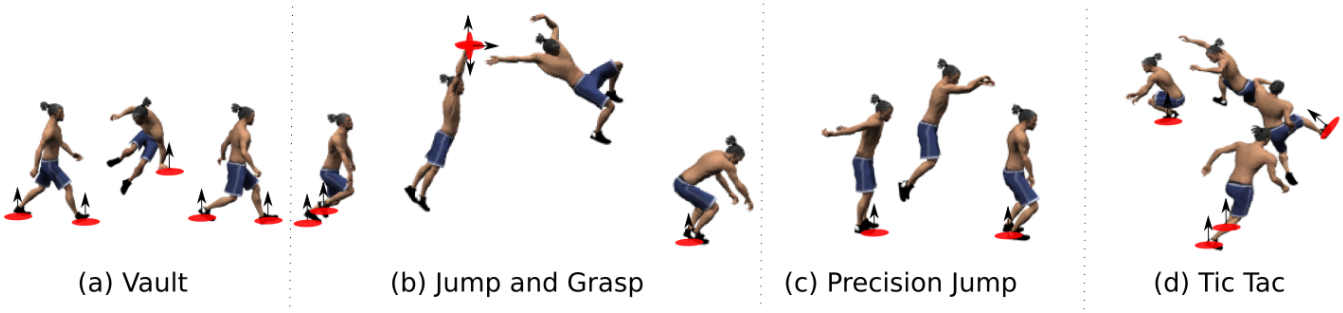


Figure 21: *Keyframes of selected motions with annotated contacts.*

The environment $\mathbf{W}$ is a collection of objects $\mathbf{o}$. The geometry of each object is represented as a set of discrete triangles generated from the environment modeling stage. We first cluster adjacent triangles into larger surfaces $\mathbf{s}$ that have the same surface normals $\mathbf{n_s}$. Starting from a triangle in the environment, we perform a breadth-first expansion to neighbouring triangles and cluster adjacent triangles that have similar facing normals. This clusters the environment triangulation into a set of contact surfaces $\mathbf{s} \in \mathbf{S}$, where $\mathbf{s}$ is a collection of adjacent triangles that face the same (or similar) direction. Fig. 23 illustrates the contact surface clustering process.

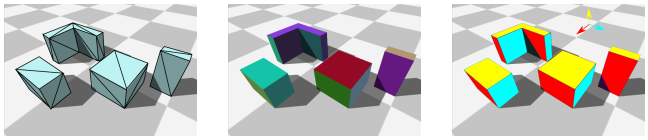


Figure 23: *Surface Clustering and Gathering. (a) Original Environment mesh. (b) Triangles with the same normals are clustered together to form surfaces. (c) Surfaces that share the same normal properties are grouped and mapped to corresponding contacts in the motion.*

Motion Detection

Given a set of surfaces $\mathbf{s} \in \mathbf{S}$ characterizing the environment, we seek to identify spatially co-located surfaces that satisfy the constraints on the motion signature of a specific motion skill, while avoiding collisions with any geometry, and to identify the valid sub-regions on each surface where each contact in the motion might take place. We provide an overview of the steps for detecting a valid transition in the environment for a motion skill $\mathbf{m}$.

Contact Rotation. The contacts $\mathbf{c} \in \mathbf{C}$ of a motion skill contain the relative positions of all contacts in the motion with respect to the initial position of the character's center of mass, assuming the character is initially facing along the positive X axis. To detect the execution of the motion skill in any direction, we first sample the set of valid start directions and transform the contact signature for the chosen direction. Note that the spatial constraints between contacts remain unaffected as a result of this rotation.

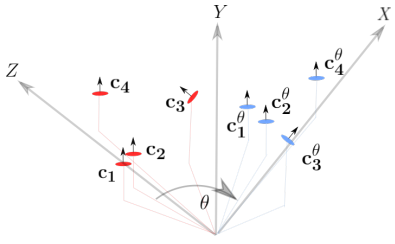


Figure 24: *Contact Rotation.*

Surface Gathering. Next, we identify all contact surfaces $\mathbf{s_i} \subseteq \mathbf{S}$ with a surface normal which is consistent with each contact $\mathbf{c_i} \in \mathbf{C}$ for the motion skill. Fig. 23(c) illustrates the surface gathering process.

Contact Constraint Satisfaction. For each pair of contact surfaces

$(\mathbf{s_a}, \mathbf{s_b}) \in \mathbf{S_1} \times \mathbf{S_2}$, we find the subregions $(\mathbf{s_a'}, \mathbf{s_b'})$ on each surface that satisfy the spatial constraint $\mathbf{e}(1,2)$ between the first two contacts $\mathbf{c_1}$ and $\mathbf{c_2}$ in the motion. This is accomplished using the algorithm described above. This produces a new set $\mathbf{R} \in \mathbf{S_1'} \times \mathbf{S_2'}$ comprising pairs of contact surfaces that satisfy the spatial constraint between the first two contacts. This process is iteratively repeated for each subsequent contact in $\mathbf{C}$ to produce $\mathbf{R} \in \mathbf{S_1'} \times \cdots \mathbf{S_{|C|}'}$ which contains a set of surface sequences that accommodate all the contacts in the motion, and each surface represents the set of all valid locations where the particular contact in the motion may take place.

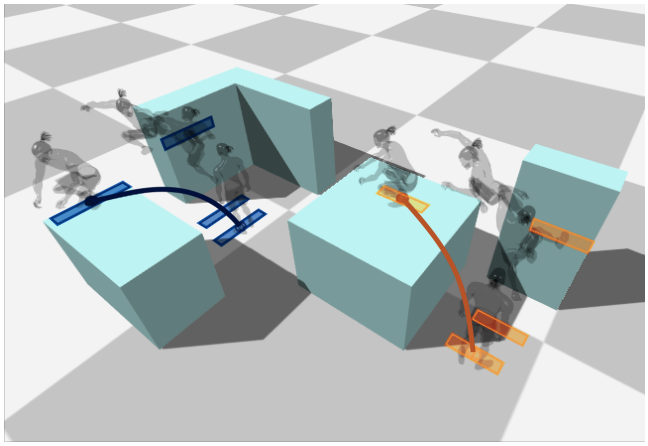


Figure 25: *Motion detection results.*

Contact Constraint Satisfaction

We check the spatial relationship between two contact surfaces and return subsurfaces that satisfy the distance constraint imposed between them, as specified by the contact constraint graph. We consider a non-deformable motion skill (e.g., a single motion clip with no procedural modification), where the spatial constraint is a single discrete value. For motion skills that have deformable motion models (e.g., a blend tree of multiple similar motion clips and/or procedural deformation using inverse kinematics), this is a distance interval, which is not considered here.

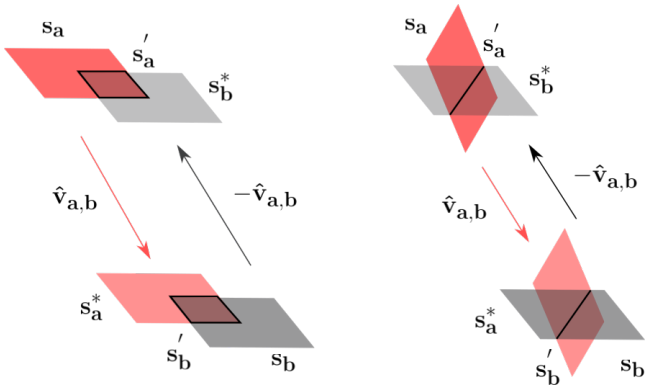


Figure 26: *(a) Two contact surfaces are parallel to each other, the overlapping region will either be a 2D polygon or null. (b) Two contact surfaces are in arbitrary positions relative to each other, the overlapping region will either be a line segment or null.*

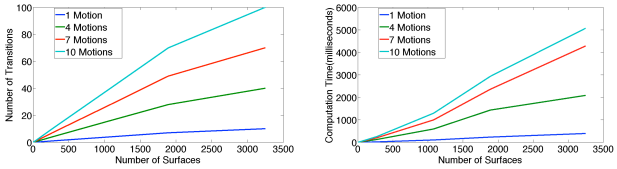
4.2.4 Results

As shown in Fig. 28, our system seamlessly integrates into existing navigation and animation pipelines to produce virtual characters that can autonomously reason about their environments while harnessing an increased repertoire of motion skills to navigate in complex virtual worlds. Our approach scales to handle multiple characters while still maintaing real-time rates.

Dynamic Game Worlds. The computational efficiency of our approach also allows motion transitions to be repaired at interactive rates, enabling new applications including dynamic game worlds. We demonstrate two example game demos in the supplementary video. In the first example, a player interactively manipulates objects in the game world while an autonomous character navigates in the environment. The second example shows many characters chasing a player-controlled character in the game.

Interactive Authoring of Game Worlds. We have developed a simple authoring tool that allows game level designers to interactively author game worlds while harnessing the power of our approach. Using our tool, an author simply drags and drops geometry into the scene. Our system works behind the scenes to detect and annotate potential ways in which a character may interact in the environment by highlighting the different possible motion transitions between objects. At any time, the author can see a real-time animation of the character navigating in the current environment to get instant visual feedback, which can be used to iteratively refine the initial design. The supplementary video shows the interactive authoring session to create the complex scene shown in Fig. 28.

Computational Efficiency. Fig. 27 analyses the computational efficiency of our approach. For complex environments with greater than 3000 surface elements (more than 20,000 triangles), nearly 100 motion transitions for 10 motion skills are detected within 5 seconds. For local repair of motion transitions for dynamic and interactive use, the operation is instantaneous.



(a) (b)

Figure 27: *Computational Efficiency Analysis.*

4.3 Event-Centric Planning

Complex virtual worlds with sophisticated, autonomous virtual populaces [Kapadia et al. 2014; Shoulson et al. 2013b] are essential for many applications. Directed interactions between virtual characters help tell rich narratives within the virtual world and personify the characters in the mind of the user. However, creating and coordinating fully-fledged virtual actors to behave realistically and act according to believable goals is a significant undertaking.

One approach to collapse a very large, mechanically-oriented character action space into a series of tools available to a narrative planner is to define and use a domain of *events*: dynamic and reusable behaviors for groups of actors and objects that carry well-defined narrative ramifications. Events facilitate precise authorial control over complex interactions involving groups of actors and objects, while planning allows the simulation of causally consistent character actions that conform to an overarching global narrative. By

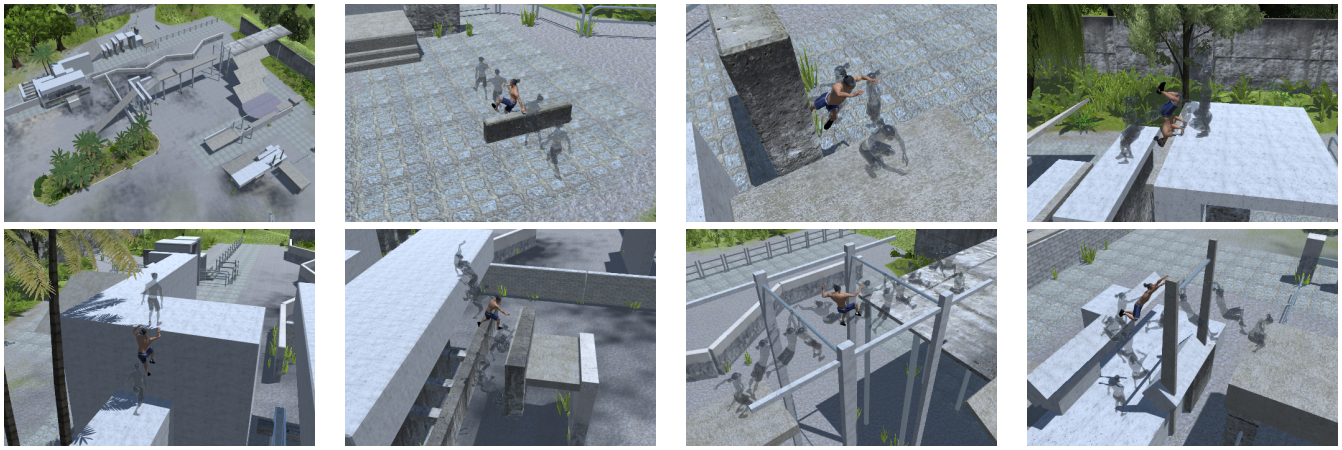


Figure 28: *Navigation in complex static environment.*

planning in the space of events rather than in the space of individual character capabilities, virtual actors can exhibit a rich repertoire of individual actions without causing combinatorial growth in the planning branching factor. Such a system [Shoulson et al. 2013a] produces long, cohesive narratives at interactive rates, allowing a user to take part in a dynamic story that, despite intervention, conforms to an authored structure and accomplishes a predetermined goal.

4.3.1 Revised Action Space

Unlike traditional multi-agent planning, the planner does not operate in the space of each actor's individual capabilities. Rather, the system's set of actions is taken from a dictionary of *events*, authored parameterizable interactions between groups of characters and props. Because of this disconnection between character behaviors and the action space for the planner, actors can have a rich repertoire of possible actions without making an impact on the planner's computational demands. Two characters involved in an interaction could have dozens of possible actions for displaying diverse, nuanced sets of gesticulations. Where a traditional planner would need to expand all of these possibilities at each step in the search, our planner only needs to invoke the conversation event between the two characters and let the authored intelligence of the event itself dictate which gestures the characters should display and when.

Formulating Narrative Events. Events are pre-authored dynamic behaviors that take as parameters a number of actors or props as participants. When launched, an event suspends the autonomy of any involved object and guides those objects through a series of arbitrarily complex actions and interactions. Events are well-suited for interpersonal activities such as conversations, or larger-scale behaviors such as a crowd gathering in protest. Each event is temporary, possessing a clear beginning and at least one end condition. When an event terminates, autonomy is restored to its participants until they are needed for any other event. To implement events we use Parameterized Behavior Trees (PBTs), a flexible, graphical programming language for single- or multi-actor behaviors. Each event is defined as

$$e = \langle t, c, \mathbf{R} = (r_1, \ldots, r_m), \phi : \mathbf{R} \times \mathbf{W}^m \rightarrow \{0, 1\}, \delta : S \rightarrow S' \rangle,$$

where the PBT t contains event behavior, c is the event's cost, the role list $\mathbf{R}$ defines the number of objects that can participate in the event and each participant's required role index number, the precondition function ϕ transforms the list of m roles and a selection of m objects from the world into a true or false value, and the postcondition function δ transforms the world state as a result of the event.

```
Event UnlockDoor(Prisoner : a, Door: d) {
  Precondition φ:
   Closed(d) ∧ ¬Guarded(d)
   ∧ Locked(d) ∧ CanReach(a,d);
  Postcondition δ:
   ¬Locked (d)
}
```

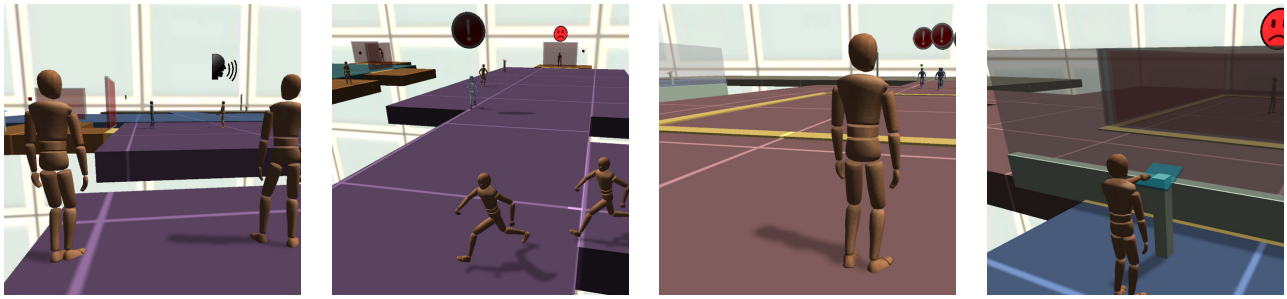
Figure 30: *Event definition for UnlockDoor.*

Roles for an event are defined based on an object's narrative value. A conversation may take two human actors, whereas an event for an orator giving a speech with an audience might take a speaker, a pedestal object, and numerous audience members. Preconditions and postconditions define the rules of the world, so that a character can pull a lever if the lever can be reached from the character's position. Figure 30 illustratates the pre- and postconditions for an *UnlockDoor* event. This event metadata would be accompanied with a PBT that instructs the character to approach the door and play a series of animations illustrating the door being unlocked.

Goal Specification. Goals can be specified as: (1) the desired state of a specific object, (2) the desired state of *some* object, or (3) a certain event that must be executed during the simulation. These individual primitives can be combined to form narrative structures suited for the unfolding dynamic story.

4.3.2 Planning in Event Space

The event system plans a sequence of events that satisfy a given narrative. The set of permissible events at each search node $I \in \mathbf{I}$ determines the transitions in the search graph. A valid instance of an event e is defined as $I_e = \langle e, O \in \mathbf{W}^{|\mathbf{R}_e|} \rangle$ with $\phi_e(\mathbf{R}_e, O) = 1$. This implies that the event's preconditions have been satisfied for a valid set of participants O, mapped to their designated roles $\mathbf{R}_e$. The transitions $\mathbf{I}$ are used to produce a set of new states: $\{S' | S_e = \delta(S, I_e) \forall I_e \in \mathbf{I}\}$ by applying the effects of each candidate event to the current state in the search. Cost is calculated based on the event's authored and generated cost, and stored alongside the transition. The planner generates a sequence of event tran-



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

Figure 29: *Characters exhibiting two cooperative behaviors. Two actors hide while a third lures a guard away (a), then sneak past once the guard is distracted (b). An actor lures the guards towards him (c), while a second presses a button to activate a trap (d).*

sitions $\Pi(S_{\text{start}}, S_{\text{goal}})$ from the S_{start} to S_{goal} that minimizes the aggregate cost.

If all of the characters and objects could participate equally in an event, then the branching factor in planning would be $|\mathbf{E}|\binom{|\mathbf{W}|}{|\mathbf{R}|}$, which would grow prohibitively large. In order to curb the combinatorial cost of this search process, actors and objects are divided into role groups. Though there may be many objects in the world, very few of them will fill any particular role, which greatly reduces the size of the event instance list. Checking an object's role is a very fast operation which is used filtering before the more costly operation to validate the event's preconditions on its candidates. The maximum branching factor of this technique will be $|\mathbf{E}|(max_{e\in\mathbf{E}}|\mathbf{R}_e|)^{max_{r\in\mathbf{R}}|\ell_r|}$, and the effective branching factor can be calculated by taking average instead of maximal values. In other words, the branching factor is bounded by the number of events, the maximum number of participants in any event, and the size of the role group with the most members.

Role groups mitigate the effect of the total number of objects in the world from on the growth of the branching factor as long as new objects are evenly divided into small groups. This increases efficiency and allows the system to scale to large groups of actors and objects at the cost of flexibility (all of the objects in a role group may be busy when needed) and authoring burden (the author must ensure that objects are divided properly). Note that the branching factor also grows independently of an actor or object's individual capabilities, allowing characters to exhibit rich and varied behaviors without affecting the planner's overhead.

The discussed techniques allow to integrate high level goals and specification of behaviors within a path planning framework in order to achieve complex simulated virtual worlds. The presented techniques can be employed and extended in several ways. They are not techniques that can be found in textbooks, they are rather the subject of an evolving interdisciplinary research field between computer animation and AI disciplines.

4.4 Planning Motions in High Dimensions

The most popular approach for locomotion synthesis around obstacles is to reduce the planning problem to 2D path planning and to rely on a locomotion controller that is able to follow 2D paths. Realistic results are often achieved with locomotion controllers built based on motion capture clips carefully organized and parameterized. Once a suitable locomotion controller is built, the approach achieves fast results and is commonly employed in real-time applications and video games.

Relying on locomotion controllers however still requires significant engineering work for achieving generic and realistic results, and several planning methods have been developed with the goal to automatically synthesize locomotion directly from an input set of unorganized motion capture data. Such an approach largely simplifies the goal of transferring to the virtual character example locomotion performances with different styles captured directly from real actors.

While locomotion controllers still represent the best approach in terms of computation performance, planning algorithms operating directly on motion capture data spaces can produce better-quality results (by imposing minimal modification to the original data), and they can plan motions of higher complexity. The several planning techniques proposed in the literature which are based on *motion graphs* lie in this category, and a representative approach in this area is illustrated in Figure 12.

Motion synthesis involving object interaction can also be translated into a high-dimensional motion planning problem with the additional need to address a higher number of challenging constraints. The next two sections present an overview of two recent approaches for full-body motion planning addressing interaction with objects.

4.5 Planning Full-Body Manipulations

One particularly challenging type of motion planning problem is when locomotion and manipulation have to work together in order to successfully accomplish a given desired action. This is the case of several full-body motions involving manipulation, such as in the example shown in Figure 31.

A new planning-based approach for whole-body motion synthesis able to generate high-quality motions for such challenging mobile-manipulation scenarios has been recently proposed [Mahmudi and Kallmann 2015]. The approach decouples the problem in specialized locomotion and manipulation skills, and proposes a multi-modal planning scheme that explores the search space of each skill together with the possible transition points between skills. In order to achieve high-quality results the locomotion skill is designed to be fully data-driven, while manipulation skills can be algorithmic or data-driven according to data availability and the complexity of the environment.

The method is able to automatically generate complex motions with precise manipulation targets among obstacles and in coordination with locomotion. Of particular importance is the ability to consider multiple locomotion trajectories during the overall search for a solution, such that the trajectory that enables the successful execution of the upper-body action is the one that eventually leads to the solution motion. The drawback of the approach is that search times can easily become prohibitive for real-time applications, requiring the development of additional simplification and pre-computation techniques.

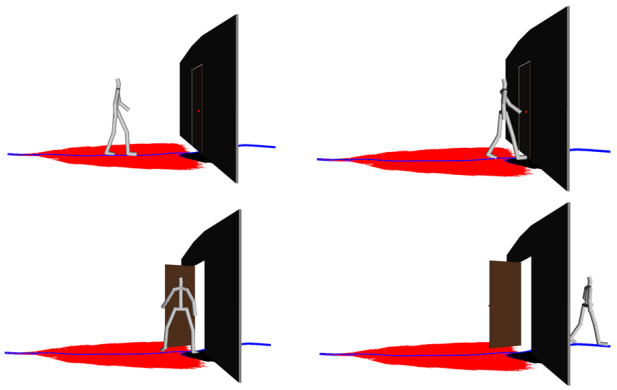


Figure 31: *Example of planning the concatenation of locomotion clips for achieving realistic locomotion while algorithmically performing an upper body manipulation (opening the door) with Inverse Kinematics.*

4.6 Planning Motions for Virtual Demonstrators

Virtual humans and embodied conversational agents are promising in the realm of human-computer interaction applications. One central goal in the area is to achieve virtual assistants that can effectively interact, train, and assist people in a wide variety of tasks. The need to demonstrate objects and procedures appears in many situations and the underlying motion synthesis problem involves several planning processes at different levels.

Simple everyday demonstrations involve a series of coordinated steps that a virtual agent needs to replicate. The agent needs to walk while avoiding obstacles along the way, stop at an appropriate demonstration location with clear view to the target and observer, interact with the object (e.g. point to it and deliver information), and also maintain visual engagement with the observer.

This section presents an overview of the main approaches taken by the planner PLACE [Huang and Kallmann 2015], which addresses such harmonious multi-level orchestration of actions and behaviors (see Figure 32).

The overall planning model was built from experiments with human subjects where participants were asked to freely approach target objects at different positions and to deliver object information to observers at various locations. These experiments provided ground truth data for defining a coordination model that is able to orchestrate the involved pieces of a demonstration task.

The result is a whole-body motion planning framework, called PLACE, that addresses the five main pieces of the overall problem in an unified way:

- Placement: optimal character placement is essential for addressing target and observer visibility, locomotion accessibility, and action execution constraints.
- Locomotion: locomotion synthesis among obstacles and towards precise placements allows the character to position itself in order to perform a demonstration.
- Action: realistic action execution needs to address arbitrary object locations and to avoid nearby obstacles when needed.
- Coordination: coordination is important for transitioning well from locomotion to the upper-body demonstrative action.
- Engagement: observer engagement is obtained with a gaze behavior that interleaves attention to the observer and the target in order to achieve effective demonstrations.

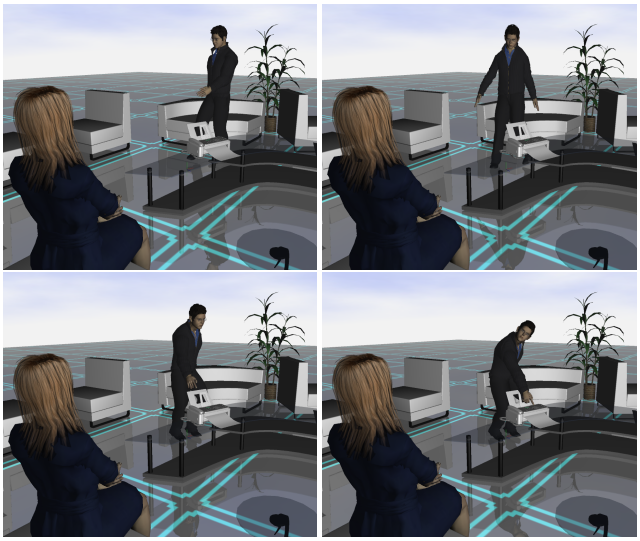


Figure 32: *The PLACE planner synthesizes whole-body demonstrations for arbitrarily located targets and observers, also taking into account obstacles and visual occluders.*

The realism of the solution is addressed at two levels. At the behavioral level, placement, coordination and engagement are solved following models extracted from experiments with human subjects. At the motion synthesis level, locomotion and actions are synthesized from collections of motion capture clips organized for efficient synthesis and coordination. The presented techniques were developed such that solutions can be computed at interactive rates in realistic, reasonably complex, environments.

The approach is based on first applying the coordination model in order to achieve a range of suitable locations for performing the object demonstration. Then, data-based locomotion is planned towards the best-ranked location, with gradual coordination with the upper-body demonstrative action and the gaze behavior. For a detailed exposition we refer the reader to the original publication [Huang and Kallmann 2015].

5 Final Notes

These course notes assemble a comprehensive overview of planning methods, techniques, approaches, and applications. We started from classical geometric path planning topics and discrete search, then gradually moved to more advanced discrete search techniques, and finalized with examples of latest techniques being developed for addressing complete simulated virtual worlds that are dynamic and with multiple agents responding to different constraints and

types of events. A more detailed and in-depth exposition of the topics covered in Sections 2 and 3 is available in the form of a book [Kallmann and Kapadia 2016].

As simulated virtual worlds become more and more complex it is expected that planning techniques will further progress in different directions, from problem modeling and algorithmic research to parallelization of algorithms and integration with behavioral planning. Collaboration between multidisciplinary researchers and the existence of related conferences and journals are also important for promoting new developments. We look forward to contributing to these developments and encourage others to get involved.

Acknowledgements

Portions of the work presented in these notes were partially supported by NSF Award IIS-0915665 and by generous gifts from Disney Research.

References

AMATO, N. M., GOODRICH, M. T., AND RAMOS, E. A. 2000. Linear-time triangulation of a simple polygon made easier via randomization. In *In Proceedings of the 16th Annual ACM Symposium of Computational Geometry*, 201–212.

ARIKAN, O., AND FORSYTH, D. A. 2002. Synthesizing constrained motions from examples. *Proceedings of SIGGRAPH 21*, 3, 483–490.

BERSETH, G., KAPADIA, M., AND FALOUTSOS, P. 2015. Acclmesh: Curvature-based navigation mesh generation. In *Proceedings of the 8th ACM SIGGRAPH Conference on Motion in Games*, ACM, New York, NY, USA, MIG '15, 97–102.

BHATTACHARYA, P., AND GAVRILOVA, M. 2008. Roadmap-based path planning - using the voronoi diagram for a clearance-based shortest path. *Robotics Automation Magazine, IEEE 15*, 2 (june), 58 –66.

CAMPORESI, C., AND KALLMANN, M. 2014. Computing shortest path maps with gpu shaders. In *Proceedings of Motion in Games (MIG)*.

CHAZELLE, B. 1982. A theorem on polygon cutting with applications. In *SFCS '82: Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, IEEE Computer Society, 339–349.

CHAZELLE, B. 1991. Triangulating a simple polygon in linear time. *Discrete Computational Geometry 6*, 5 (Aug.), 485–524.

CHEW, L. P. 1985. Planning the shortest path for a disc in $o(n^2 logn)$ time. In *Proceedings of the ACM Symposium on Computational Geometry*.

CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. 2009. *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press.

DE BERG, M., CHEONG, O., AND VAN KREVELD, M. 2008. *Computational geometry: algorithms and applications*. Springer.

DEMYEN, D., AND BURO, M. 2006. Efficient triangulation-based pathfinding. In *AAAI'06: Proceedings of the 21st national conference on Artificial intelligence*, AAAI Press, 942–947.

DEVILLERS, O., AND PION, S. 2003. Efficient exact geometric predicates for delaunay triangulations. In *Proceedings of the 5th Workshop Algorithm Engineering and Experiments*, 37–44.

DIJKSTRA, E. W. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik 1*, 269–271.

GARCIA, F., KAPADIA, M., , AND BADLER, N. I. 2014. Gpu-based dynamic search on adaptive resolution grids. In *Proceedings of the IEEE International Conference on Robtics and Automation*, IEEE, ICRA.

GERAERTS, R. 2010. Planning short paths with clearance using explicit corridors. In *ICRA'10: Proceedings of the IEEE International Conference on Robotics and Automation*.

GOCHEV, K., COHEN, B. J., BUTZKE, J., SAFONOVA, A., AND LIKHACHEV, M. 2011. Path planning with adaptive dimensionality. In *SOCS*.

HART, P., NILSSON, N., AND RAPHAEL, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics 4*, 2, 100–107.

HERSHBERGER, J., AND SNOEYINK, J. 1994. Computing minimum length paths of a given homotopy class. *Computational Geometry Theory and Application 4*, 2, 63–97.

HERSHBERGER, J., AND SURI, S. 1997. An optimal algorithm for euclidean shortest paths in the plane. *SIAM Journal on Computing 28*, 2215–2256.

HJELLE, O., AND DÆHLEN, M. 2006. *Triangulations and Applications (Mathematics and Visualization)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

HOFF III, K. E., CULVER, T., KEYSER, J., LIN, M., AND MANOCHA, D. 2000. Fast computation of generalized voronoi diagrams using graphics hardware. In *ACM Symposium on Computational Geometry*.

HUANG, Y., AND KALLMANN, M. 2015. Planning motions and placements for virtual demonstrators. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*.

HUANG, T., KAPADIA, M., BADLER, N. I., , AND KALLMANN, M. 2014. Path planning for coherent and persistent groups. In *Proceedings of the IEEE International Conference on Robtics and Automation*, IEEE, ICRA '14.

JORGENSEN, C.-J., AND LAMARCHE, F. 2011. From geometry to spatial reasoning: automatic structuring of 3d virtual environments. In *Proceedings of the 4th international conference on Motion in Games (MIG)*, Springer-Verlag, Berlin, Heidelberg, 353–364.

KALLMANN, M., AND KAPADIA, M. 2014. Navigation meshes and real-time dynamic planning for virtual worlds. In *ACM SIGGRAPH 2014 Courses*, ACM, New York, NY, USA, SIGGRAPH '14, 3:1–3:81.

KALLMANN, M., AND KAPADIA, M. 2016. *Geometric and Discrete Path Planning for Interactive Virtual Worlds*. Morgan and Claypool Publishers.

KALLMANN, M. 2010. Navigation queries from triangular meshes. In *The Third International Conference on Motion in Games (MIG)*.

KALLMANN, M. 2010. Shortest paths with arbitrary clearance from navigation meshes. In *Proceedings of the Eurographics / SIGGRAPH Symposium on Computer Animation (SCA)*.

KALLMANN, M. 2014. Dynamic and robust local clearance triangulations. *ACM Transactions on Graphics 33*, 5.

KAPADIA, M., AND BADLER, N. I. 2013. Navigation and steering for autonomous virtual humans. *Wiley Interdisciplinary Reviews: Cognitive Science*.

KAPADIA, M., SINGH, S., HEWLETT, W., AND FALOUTSOS, P. 2009. Egocentric affordance fields in pedestrian steering. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, ACM, New York, NY, USA, I3D '09, 215–223.

KAPADIA, M., SINGH, S., REINMAN, G., AND FALOUTSOS, P. 2011. A behavior-authoring framework for multiactor simulations. *Computer Graphics and Applications, IEEE 31*, 6 (nov.-dec.), 45 –55.

KAPADIA, M., SINGH, S., HEWLETT, W., REINMAN, G., AND FALOUTSOS, P. 2012. Parallelized egocentric fields for autonomous navigation. *The Visual Computer 28*, 12, 1209–1227.

KAPADIA, M., BEACCO, A., GARCIA, F., REDDY, V., PELECHANO, N., AND BADLER, N. I. 2013. Multi-domain real-time planning in dynamic environments. In *Proceedings of the 12th ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, ACM, New York, NY, USA, SCA '13, 115–124.

KAPADIA, M., NINOMIYA, K., SHOULSON, A., GARCIA, F., AND BADLER, N. 2013. Constraint-aware navigation in dynamic environments. In *Proceedings of Motion on Games*, ACM, New York, NY, USA, MIG '13, 89:111–89:120.

KAPADIA, M., MARSHAK, N., AND BADLER, N. I. 2014. ADAPT: The agent development and prototyping testbed. *IEEE Transactions on Visualization and Computer Graphics 99*, 1.

KAPADIA, M., XIANGHAO, X., NITTI, M., KALLMANN, M., COROS, S., SUMNER, R. W., AND GROSS, M. 2016. Precision: Precomputing environment semantics for contact-rich character animation. In *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, I3D '16, 29–37.

KOENIG, S., AND LIKHACHEV, M. 2002. D* Lite. In *National Conf. on AI*, AAAI, 476–483.

KOVAR, L., GLEICHER, M., AND PIGHIN, F. H. 2002. Motion graphs. *Proceedings of SIGGRAPH 21*, 3, 473–482.

LAMARCHE, F., AND DONIKIAN, S. 2004. Crowd of virtual humans: a new approach for real time navigation in complex and structured environments. *Computer Graphics Forum 23*, 3, 509–518.

LAMARCHE, F. 2009. Topoplan: a topological path planner for real time human navigation under floor and ceiling constraints. *Computer Graphics Forum 28*, 2, 649–658.

LEE, D. T., AND PREPARATA, F. P. 1984. Euclidean shortest paths in the presence of rectilinear barriers. *Networks 3*, 14, 393–410.

LIKHACHEV, M., GORDON, G. J., AND THRUN, S. 2003. ARA*: Anytime A* with Provable Bounds on Sub-Optimality. In *NIPS*.

LIKHACHEV, M., FERGUSON, D. I., GORDON, G. J., STENTZ, A., AND THRUN, S. 2005. Anytime Dynamic A*: An Anytime, Replanning Algorithm. In *ICAPS*, 262–271.

LIU, Y. H., AND ARIMOTO, S. 1995. Finding the shortest path of a disk among polygonal obstacles using a radius-independent graph. *IEEE Transactions on Robotics and Automation 11*, 5, 682–691.

LOZANO-PÉREZ, T., AND WESLEY, M. A. 1979. An algorithm for planning collision-free paths among polyhedral obstacles. *Communications of ACM 22*, 10, 560–570.

MAHMUDI, M., AND KALLMANN, M. 2012. Precomputed motion maps for unstructured motion capture. In *Eurographics/SIGGRAPH Symposium on Computer Animation (SCA)*.

MAHMUDI, M., AND KALLMANN, M. 2013. Analyzing locomotion synthesis with feature-based motion graphs. *IEEE Transactions on Visualization and Computer Graphics 19*, 5, 774–786.

MAHMUDI, M., AND KALLMANN, M. 2015. Multi-modal data-driven motion planning and synthesis. In *Proceedings of the 8th ACM SIGGRAPH Conference on Motion in Games, MIG 2015, Paris, France, November 16-18, 2015*, 119–124.

MITCHELL, J. S. B. 1991. A new algorithm for shortest paths among obstacles in the plane. *Annals of Mathematics and Artificial Intelligence 3*, 83–105.

MITCHELL, J. S. B. 1993. Shortest paths among obstacles in the plane. In *Proceedings of the ninth annual symposium on computational geometry (SoCG)*, ACM, New York, NY, USA, 308–317.

MUBBASIR KAPADIA, FRANCISCO GARCIA, C. D. B., AND BADLER, N. I. 2013. Dynamic search on the gpu. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, IEEE, IROS '13.

NILSSON, N. 1969. A mobile automaton: an application of artificial intelligence techniques. In *In Proceedings of the 1969 International Joint Conference on Artificial Intelligence (IJCAI)*, 509–520.

OLIVA, R., AND PELECHANO, N. 2013. A generalized exact arbitrary clearance technique for navigation meshes. In *In Proceedings of the ACM SIGGRAPH conference on Motion in Games (MIG)*.

OLIVA, R., AND PELECHANO, N. 2013. Neogen: Near optimal generator of navigation meshes for 3d multi-layered environments. *Computer & Graphics 37*, 5, 403–412.

RECAST, 2014. Recast navigation mesh. https://github.com/memononen/recastnavigation.

SHEWCHUK, J. R. 1996. Triangle: Engineering a 2d quality mesh generator and delaunay triangulator. In *Applied Computational Geometry: Towards Geometric Engineering*, M. C. Lin and D. Manocha, Eds., vol. 1148 of *Lecture Notes in Computer Science*. Springer-Verlag, May, 203–222. From the First ACM Workshop on Applied Computational Geometry.

SHEWCHUK, J. R. 1997. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry 18*, 3 (Oct.), 305–363.

SHOULSON, A., GILBERT, M. L., KAPADIA, M., AND BADLER, N. I. 2013. An event-centric planning approach for dynamic real-time narrative. In *Proceedings of Motion on Games*, ACM, New York, NY, USA, MIG '13, 99:121–99:130.

SHOULSON, A., MARSHAK, N., KAPADIA, M., AND BADLER, N. I. 2013. Adapt: the agent development and prototyping testbed. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, I3D '13, 9–18.

SINGH, S., KAPADIA, M., REINMAN, G., AND FALOUTSOS, P. 2011. Footstep navigation for dynamic crowds. *Computer Animation and Virtual Worlds 22*, 2-3, 151–158.

THE CGAL PROJECT. 2014. *CGAL User and Reference Manual*, 4.4 ed. CGAL Editorial Board. http://doc.cgal.org/4.4/Manual/packages.html.

TRIPATH TOOLKIT, 2010. Triangulation and path planning toolkit. http://graphics.ucmerced.edu/software/tripath/.

VAN TOLL, W. G., IV, A. F. C., AND GERAERTS, R. 2011. Navigation meshes for realistic multi-layered environments. In *In Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 3526–3532.

VAN TOLL, W. G., IV, A. F. C., AND GERAERTS, R. 2012. A navigation mesh for dynamic environments. *Computer Animation and Virtual Worlds (CAVW) 23*, 6, 535–546.

WEIN, R., VAN DEN BERG, J., AND HALPERIN, D. 2007. The visibility-voronoi complex and its applications. *Computational Geometry: Theory and Applications 36*, 1, 66–78.

XIAO, J., ZHUANG, Y., YANG, T., AND WU, F. 2006. An efficient keyframe extraction from motion capture data. In *Advances in Computer Graphics*, vol. 4035 of *LNCS*. Springer Berlin Heidelberg, 494–501.