22

Collision Avoidance for Preplanned Locomotion

Bobby Anguelov

22.1 Introduction
22.2 Collision Avoidance for Preplanned Locomotion
22.3 Collision Detection and Trivial Collision Resolution
22.4 Nontrivial Collision Resolution through Path Modification
22.5 Performance and Visual Quality
22.6 Conclusion

22.1 Introduction

Collision avoidance for NPCs is a critical component of any character locomotion system. To date, the majority of collision avoidance approaches are based either on Reynold's seminal steering articles [Reynolds 99] or one of the many reciprocal velocity obstacle techniques (RVO) [Guy et al. 10, v.d. Berg et al. 08]. In trying to increase the visual fidelity of our character locomotion by reducing artifacts such as foot sliding, some developers are moving away from traditional steering-based locomotion in favor of *animation-driven locomotion* (ADL) systems in combination with preplanned motion. This article discusses the implications of moving your locomotion system to a preplanned ADL system with regards to avoidance and why traditional collision avoidance systems may be overkill for preplanned motion. We present the collision avoidance approach used in *Hitman: Absolution* (HMA) and discuss how this system can be adapted for use with any preplanned locomotion system.

In traditional steering systems, characters are usually simulated as moving spheres and character trajectories are calculated based on the current velocities of these spheres. Appropriate character animations are then layered on top of this simulation to give the illusion that the character is actually moving. Since there exists a disconnect between

the animation and the simulation, the animation is not guaranteed to exactly match the simulation and results in noticeable artifacts like foot sliding. ADL takes the opposite approach wherein a character's trajectory updates are read directly from the animation, meaning that the character's position updates and animations are in-sync, completely eliminating foot sliding. Unfortunately, in using ADL, we constrain character motion to the set of animations available, thereby potentially sacrificing the wide range of motion offered by steering systems. Furthermore, ADL systems have an inherent latency associated with them resulting from the fact that we can only change our motion whenever a foot is planted on the ground, meaning that we often have to wait for a footstep to complete before we can adjust our movement. A detailed discussion of these systems is beyond the scope of this article and interested readers are referred to [Anguelov et al. 12] for more information.

Once we have our characters moving around, we would ideally like to have them navigate from one location to another in our game world. The key difference, at least with regards to collision avoidance, is due to the path-following behavior in the locomotion system. In most cases, path points are simply treated as rough goals, and the steering system is tasked with navigating between them. In steering-based systems, the path resulting from the steering actions can deviate significantly from the original path found. Unfortunately, depending on the ADL motion constraints as well as the ADL latency, we could end up in problematic situations where characters clip corners or potentially leave the navigation mesh (navmesh).

This can occur when the steering desires are too fine grained for the locomotion system to satisfy or the locomotion system doesn't take the ADL latency into account, which is extremely problematic at high movement speeds. These problems can be ameliorated through complex adjustments to the steering system taking the ADL constraints into account as well as other measures such as constraining characters to the navmesh. Unfortunately, the complexity of such adjustments rapidly increases to the point where our locomotion system is significantly complex, without even taking collision avoidance into account. Now this is not to say such an approach will not work, since many developers use exactly this approach with great success, but we feel there is a simpler solution.

We think the important thing is to not look at the ADL reduced motion set as a disadvantage, but rather as a benefit, since the reduction in options makes it feasible for us to preplan our motion for the entire path. What we mean by preplanning is to plan the exact path, and potentially the set of animations needed, to reach our end goal prior to starting locomotion. This means that we can, for any given point in time, predict both the exact position and velocity of an agent. There are various ways to achieve this preplanning, and readers are referred to [Champandard 09] and [Anguelov 12] for more information.

It is with these preplanned systems that traditional avoidance techniques start to lose their applicability due to the exact precomputation of our locomotion. Standard RVO systems resolve collision by trying to find a local, collision-free velocity for a character, relative to other characters in the scene. The character's velocity is then adjusted to match the collision-free velocity. This is all done within a local neighborhood and only returns the immediate collision-free velocity, not taking anything else into account; this can potentially result in anomalous behavior such as agent oscillation. Discussing the potential problems with RVO is beyond the scope of this article and we simply wish to point out that RVO is a local avoidance system, which, given our global knowledge of our agents' locomotion, may not be the best approach to solving the avoidance problem.

22.2 Collision Avoidance for Preplanned Locomotion

We built a very simple yet robust avoidance system that allows us to detect collision on a more global scale than what RVO would have allowed. Our solution also allows us to resolve collision in a high fidelity manner entirely within the constraints of our ADL system. *Hitman: Absolution's* locomotion is a preplanned ADL-based system, with each character following a smoothed path precisely. These smoothed paths are created by postprocessing an existing navmesh path, and then converting this path into a set of continuous quadratic Bézier curves. Interested readers are referred to Anguelov [12] for information on the path postprocessing used in HMA.

Our characters will then follow these smoothed paths precisely, with the distance traveled per frame being read from the currently selected animation. Simply put, you could think of our characters as being on rails. These paths serve as the primary input to our avoidance system with the secondary input being the characters' current state and motion data, which is populated by the locomotion system.

Our avoidance system consists of three distinct stages: collision detection, trivial collision resolution, and nontrivial collision resolution. Our avoidance system is run once per agent per frame, and each agent is checked for a collision with every other agent (now termed *colliders*) in the scene sequentially. The result of this collision check is either a speed modification/stop order or a request for path replanning.

The first stage and the core of our avoidance approach is the collision detection mechanism. Our characters (now termed *agents*) are modeled as collision spheres with a fixed collision radius, and the premise behind the collision detection system is to simply slide our spheres along our paths and check whether they make it to the end of their paths without colliding with any other spheres.

During the frame update of each agent's animation/locomotion programs, the agent will query the avoidance system to see whether its current path is collision-free. The avoidance system will perform a collision detection pass as well as attempt to trivially resolve a detected collision. The details of this system are too complex to discuss here, since it is built around our locomotion system, so we will not attempt to do so but rather we will try to describe the higher level concepts the best we can to try to inspire you to build similar systems.

22.3 Collision Detection and Trivial Collision Resolution

Our collision detection works as follows: We first calculate a collision detection range for our agent. In our case, we know exactly how much time and distance is necessary for our agent to stop if we had to immediately issue a stop command. This stopping distance is then added to a specified time horizon length (in seconds) multiplied by our current velocity, the result being the collision detection range for our agent (refer to Figure 22.1a).

Stopping distance is important as we don't want to trigger a "stop" command which will result in our agent stopping in the path of another agent or, even worse, end up stopping inside another agent. The time horizon window allows us to detect collisions much further in advance than an RVO system could. In our testing, we've found it sufficient to only check two seconds ahead for a good balance of performance and fidelity.

Before we actually check the agent's path, we perform two exclusion checks on the collider. We first run a simple dot product check to determine whether the collider is in

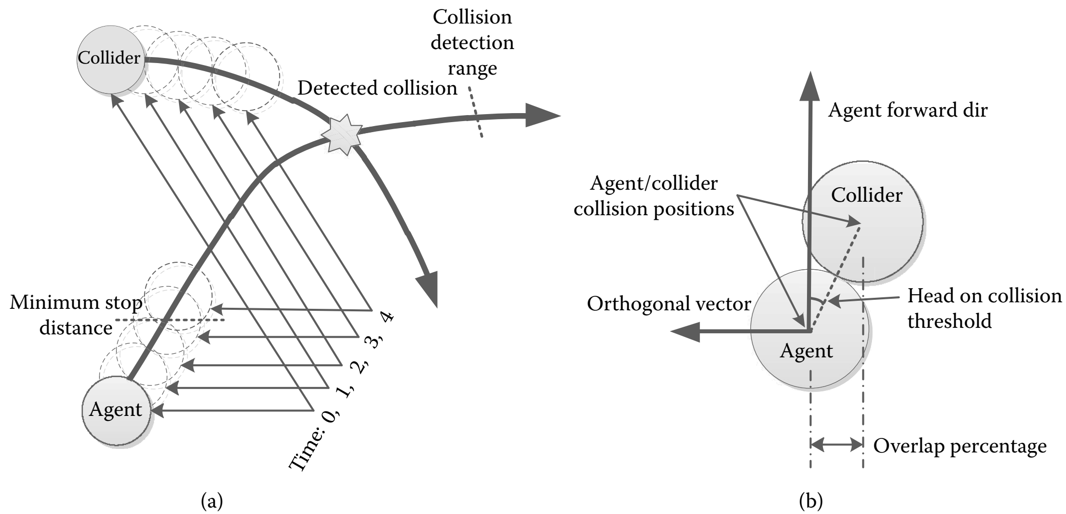

Figure 22.1

(a) Collision detection along an agent's path. (b) The stored collision data for a given collision.

front of the agent relative to the movement direction. In ignoring any colliders behind the agent, we implicitly delegate the responsibility for any potential collisions that may occur to those colliders. If the collider is determined to be in the direction of our movement, we perform a simple sphere–sphere intersection test [Ericson 05a] using the agent and collider positions for the sphere origins and the agent and collider's calculated collision detection ranges as the sphere radii. If both tests succeed, we then proceed to check the agent's path for a collision with the collider.

The path checking is performed by moving the agent along its path by the *agent's collision radius* (ACR). By moving the agent along its path in discrete intervals, we are discretizing our continuous path into ACR length pieces. The time needed for the agent to complete one ACR length movement (i.e., the *time per movement* or TPM) is determined by the agent's current velocity and is used to calculate the average velocity vector for the move. We then proceed to move the collider along its path by calculating the collider's position at the end of the TPM interval to ensure temporal consistency; this position and time is used to calculate a collider velocity vector.

In our case, our agents always move with a fixed velocity so this calculation is trivial, but it will be more complex if you have varying velocities per animation clip. This process of moving an agent's collision sphere along the path is visualized for 4 TPM intervals in Figure 22.1a. Once we have the agent and collider final positions and the average velocity vectors, we perform a moving-sphere–moving-sphere check [Ericson 05b].

If we detected an intersection between the spheres, we need to record some data regarding the collision for later use. This data initially contains the positions of the agent and collider at the point of collision. We also store whether the collision was head-on by checking whether the angle between the agent's forward direction and the vector to the center of the collider is within some threshold (in our case ~10 degrees). A rough estimate of the

potential overlap percentage between the two collision spheres for that collision is then calculated and stored. The overlap percentage is calculated as the actor collision sphere diameter minus the min distance between the two spheres (along their respective paths) divided by the actor collision diameter. The final bit of data we store is the orthogonal vector to the agent's forward direction, away from the collider's direction of movement. This orthogonal vector is used for the path modification later on. All the necessary data we store for a detected collision is visualized in Figure 22.1b.

If no collision is detected, we calculate the collision sphere motion for the next TPM interval and repeat the check until we reach the end of the agent's collision detection range. If we have detected a collision then we immediately terminate the detection stage and move onto the trivial resolution stage. It is important to note that our system is only concerned with the first detected collision and so only tries to resolve that collision. The assumption is made that any further collisions along the agent's path will be detected and resolved on subsequent frames.

The trivial resolution stage attempts to resolve the detected collision through simple speed modification. We attempt to do this by performing the path checking algorithm at a different agent speed. If this new agent speed results in a collision-free path, then we simply instruct the agent to change speed. This adjustment is immediate, and any subsequent calls into the avoidance system will make use of the updated agent speed. This means that any other agents running an avoidance check will make use of the updated agent speed.

The speed modification check is run for all available speeds, and if all the speeds still result in a collision, then we check whether our stopping distance is collision-free; if it is then we instruct the agent to stop and wait until he can continue on a collision-free path.

This simple system resolved the bulk of our existing in-game collisions, but we needed an additional system to handle collisions that couldn't be avoided through simple speed modification (e.g., stationary agents or head-on collisions). This secondary system is used for what we termed nontrivial collision resolution, but before we carry on we need to discuss some details regarding the collision detection stage.

First, we've made an assumption that all of our characters are moving but that is not always the case. In many situations characters are stationary, either performing some level-specific or idle act (e.g., using an ATM or leaning against a wall). These stationary characters have to be handled differently since they have no paths allocated. The collision checks are performed in the same way, except now the agent is stationary so we simply don't need to calculate an update position for the collider.

Second, even though agents are moving, they might be in a transition animation for starting or stopping. In our case, our agents travel with a linear velocity, which greatly simplifies the math in the path collision check. When starting or stopping, we had a nonlinear velocity during the transition, so our prediction of agent velocity and position during those transitions was rather complex. We didn't want to unnecessarily increase complexity by modeling the nonlinear transition velocity in our avoidance code, so we simply resorted to estimating the velocity within those transitions.

Agent velocities during transitions were estimated by dividing the remaining distance of the transition by the time of the transition. We also tried to ensure that our start-and-stop animations were as short as possible, further reducing the error of this estimation. Something to keep in mind is that your transitions may have long periods

wherein the agent is not actually moving. For example, we had a *turn-on-spot* starting transition where the agent would be turning for more than half of the transition, but didn't actually change position; this broke our estimation code and required us to have to preprocess all animations to determine the portion within the animations that actually move the character. Luckily, these nonmoving intervals are usually only at the start or end of an animation, which makes it easy to deal with by simply decreasing the transition time and treating the collider as stationary for that time.

Since our default avoidance query is for already moving agents, we also need to take into consideration agents that are stationary and wish to begin moving. We created a custom *'CheckForCollisionFreeStart'* avoidance query that takes into account our nonlinear start transition and determines when it is safe to start moving. This additional check allows us to wait for other agents to get out of the way before we start moving. We added two additional game-specific queries to the avoidance system dealing with combat sidesteps and *shoot-from-cover* acts. Since we don't want an agent to step out into another agent's path, we provided an interface for the combat programs to query whether a proposed new position was collision-free before issuing any move/act orders.

22.4 Nontrivial Collision Resolution through Path Modification

Our nontrivial collision resolution is a path modification system: we modify an agent's path around an obstacle without any need for replanning on a pathfinder level. Making use of the collision data stored during the detection stage, we calculate an *avoidance point* (AP) that will resolve the collision and a *reconnection point* (RP) on the original path.

The entire path starting from the agent's current position to the RP is replaced with a new path that is made up of two cubic Bézier curves and which goes through the AP. The AP is calculated as a point that lies at some distance along the orthogonal vector to the agent's forward vector at the point of collision. We apply a relatively large distance of around 5× the agent collision radius along the orthogonal vector to calculate the avoidance point. It is important to note that the actual position of the avoidance point is not all that important; in simply altering the length of the path, we also affect the time at which we would reach the previous collision point, which in itself is often enough to resolve the collision.

After calculating the avoidance point, we perform a navmesh query to ensure that the avoidance point is both on the navmesh and straight-line-reachable from the agent's collision position. If the point is off the navmesh, we simply truncate the point to the furthest on-navmesh point along the orthogonal vector that is further than some minimum avoidance threshold. We then set the tangent of the AP to be the same as the tangent on the original path at the point of collision. This means that the avoidance point simply acts as an offset to the path. If we have truncated the AP onto an exterior edge of the navmesh, we have to modify the tangent at the AP to be the same as the tangent to the exterior edge of the navmesh; doing this will help ensure that the cubic Bézier curves stay within the navmesh at the AP (refer to Figure 22.2b).

The next step is to find the furthest straight-line-reachable navmesh point along the path from the point of collision; this is the RP. It is important to ensure that the tangent at the AP and the vector to the reconnection point are dissimilar by some threshold (in our case 8 degrees) to ensure that reconnection curve doesn't cross over the original path. Finally, we then cut the path segment from the agent's current position to the RP

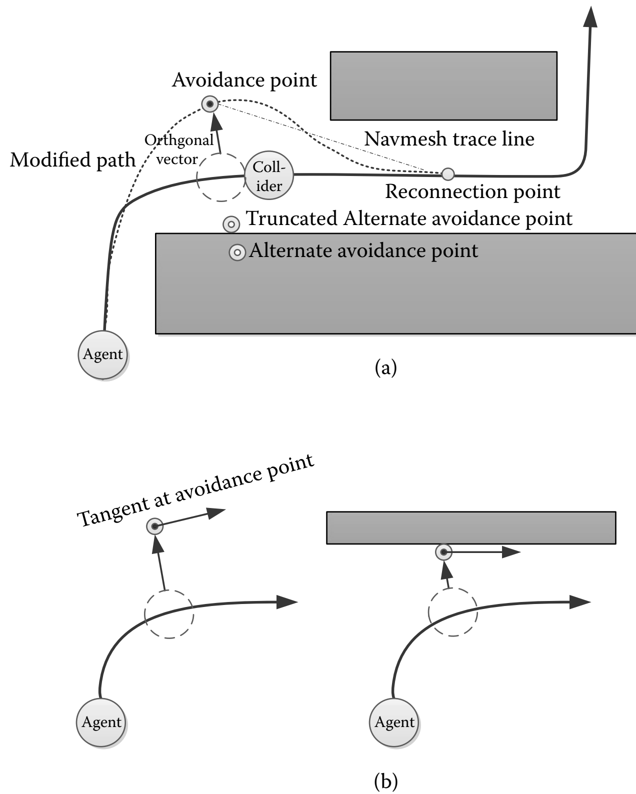

Figure 22.2

(a) Path modification. (b) Avoidance point tangents.

and replace it with the two cubic Bézier curves which pass through the avoidance point respecting the set avoidance point tangent.

Before we can accept this modified path, it is necessary to run some validation checks on it. The first check discretizes the path into short straight line segments, and we run navmesh straight-line-reachable queries for each segment to ensure that the path does not leave the navmesh. If the path leaves the navmesh, it is immediately discarded. The second check performed is a path collision detection check, exactly the same as in the detection stage, at the current agent speed on the newly modified path though. Unlike the detection stage, this check is considered successful if the result is either collision-free or is a speed modification instruction. If we receive a speed modification instruction, then we can safely accept the path, knowing that on the next frame the speed modification instruction will be received and executed.

Sometimes the modified path will result in an unavoidable collision; so what do we do in that case? We decided to try to find an alternate modified path by calculating a new AP using the negated orthogonal collision vector. We validate this alternate path and if it once again results in an unavoidable collision, then we simply pick between the two modified paths by selecting the path which results in the lowest collision overlap percentage.

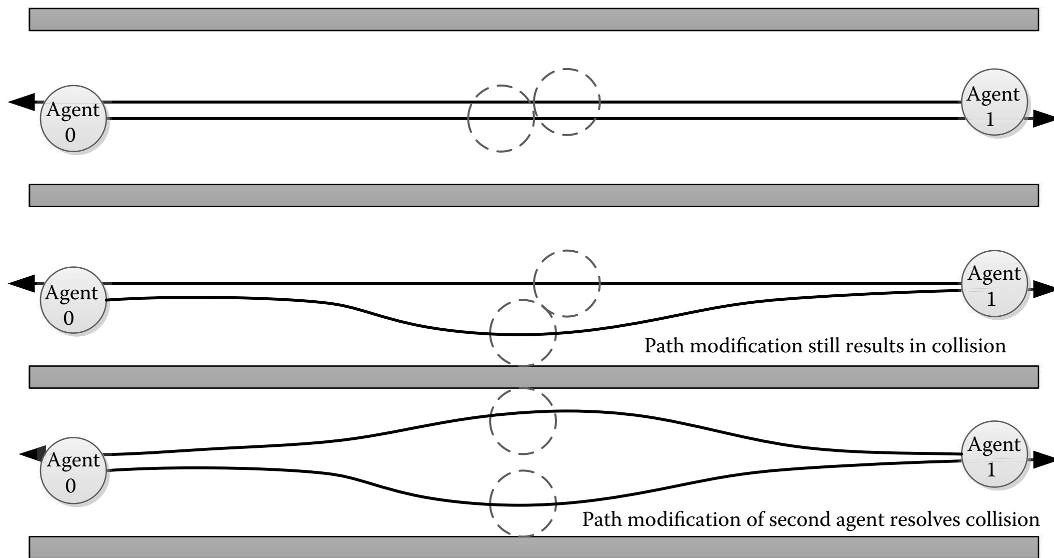

Figure 22.3

Collision resolution through multiple agent path modification.

In many cases, neither of the paths will resolve the collision, so we try to minimize the collision as much as possible. In many cases, upon running the avoidance query upon the collider and again trying to minimize the collision, you will potentially resolve it. This is exactly what we see happening when several agents try to navigate through a narrow corridor. The added benefit is that it results in flow channels forming. An example of collision resolution through collision minimization is illustrated in Figure 22.3.

22.5 Performance and Visual Quality

Visual quality was extremely important to us, so we added a few things to the avoidance system to help with the overall visual fidelity of our locomotion. We noticed that when sending groups of agents to investigate events, the agents would tend to bunch up and move together. Even though the agents were collision-free, the result looked poor. We decided to add a further check at the end of our collision detection stage (if we don't detect any collisions) to ensure that we maintain a minimum distance from any agents walking in the same direction as ourselves. We simply checked that if an agent was walking in the same direction as a collider and the agent was less than the minimum distance from the collider, we simply slowed down until we satisfied the minimum distance requirement. This simple check had a huge impact on the visual quality of our agents. We also had a problem where bugs in our AI would sometimes send more than one agent to the same position, so we leveraged the path modification to allow us to modify the end point of our path, ensuring that agents did not end up right on top of one another.

In general, the visual fidelity of our avoidance system is extremely high, having agents modify their speeds or simply stop and wait looks quite good (and in our opinion quite natural) especially within confined spaces. Furthermore, our system doesn't exhibit any of the oscillation side effects that are quite prevalent in the RVO family. Since our collision detection is performed along the agent's path, we have the added benefit over RVO that our detection works around corners allowing us to resolve collision well in advance of the agent reaching the corner.

It has been suggested that most of the problems with RVO can be ameliorated through tweaking of the parameters or the algorithm, but this only further increases the cost and complexity of such an approach, which is in stark contrast to the simplicity of our approach. Performance-wise, we found our system to be extremely cheap in that we are able to perform avoidance queries, in our production levels, for around 30 agents at a max total cost of around 1~1.5% (0.3~0.5 ms) of the frame time on Playstation 3.

22.6 Conclusion

We have presented a very simple alternative to RVO-based avoidance for use with preplanned locomotion. The system makes use of simple geometric intersection tests to perform collision detection and had two stages of collision resolution. The trivial stage made use of speed modification and stopping to resolve collisions, while the nontrivial stage calculated a new path in an attempt to avoid the detected collision. We discussed the high level concepts of such a system in *Hitman: Absolution* and the results thereof. This system is able to handle multiple agents in a complex environment and results in emergent flow fields developing in confined spaces. Furthermore, the premise behind the system is extremely simple and easy to extend, allowing developers a large degree of freedom in applying these concepts to their future games.

References

[Anguelov et al. 12] B. Anguelov, S. Harris, and G. Le Blanc. "Animation driven locomotion for smoother navigation." *Game Developers Conference (GDC)*, 2012.

[Champandard 09] A. J. Champandard. "Dynamic Locomotion by Example with Alex Champandard." http://aigamedev.com/premium/tutorial/dynamic-locomotion/, 2009.

[Ericson 05a] C. Ericson. *Real Time Collision Detection*. San Francisco, CA: Elsevier, 2005, pp. 88–89.

[Ericson 05b] C. Ericson. *Real Time Collision Detection*. San Francisco, CA: Elsevier, 2005, pp. 223–226.

[Guy et al. 10] S. J. Guy, M. C. Lin, and D. Manocha. "Modeling collision avoidance behavior for virtual humans." *Proc. of the 9th Int. Conf. on Autonomous Agents and Multi-agent Systems (AAMAS)*. 2010.

[Reynolds 99] C. W. Reynolds. "Steering behaviors for autonomous characters." *Game Developers Conference, 1999*. Available Online (http://www.red3d.com/cwr/steer/gdc99/).

[v.d. Berg et al. 08] J. v.d. Berg, M. Lin, and D. Manocha. "Reciprocal velocity obstacles for real-time multi-agent navigation." *IEEE International Conference on Robotics and Automation (ICRA 08*). 2008.