Reciprocal Collision Avoidance and Multi-Agent Navigation for Video Games

Jamie Snape, Stephen J. Guy, Ming C. Lin, Dinesh Manocha
Department of Computer Science,
University of North Carolina at Chapel Hill
Chapel Hill, North Carolina 27599
{snape, sjguy, lin, dm}@cs.unc.edu

Jur van den Berg
School of Computing,
University of Utah
Salt Lake City, Utah 84112
berg@cs.utah.edu

Abstract
Collision avoidance and multi-agent navigation is an important component of modern video games. Recent developments in commodity hardware, in particular the utilization of multi-core and many-core architectures in personal computers and consoles are allowing large numbers of virtual agents to be incorporated into game levels in increasing numbers. We present the hybrid reciprocal velocity obstacle and optimal reciprocal collision avoidance methods for reciprocal collision avoidance and navigation in video games and described their implementations in C++ as HRVO Library and RVO2 Library. The libraries can efficiently simulate groups of twenty-five to one thousand virtual agents in dense conditions and around moving and static obstacles.

1 Introduction
Collision avoidance and navigation among virtual agents is an important component of modern video games. Recent developments in commodity hardware, in particular the utilization of multi-core and many-core architectures in personal computers and consoles are allowing large numbers of virtual agents to be incorporated into game levels in increasing numbers and with increasing fidelity. As a result, there is a need for efficient techniques to automatically generate realistic behaviors for such groups of virtual agents.

Simple local collision avoidance behaviors, such as flocking (Reynolds 1987), have been implemented using force-based models in many recent video games and commercial game engines. These methods model groups of virtual agents as particle systems, with each particle applying a force on nearby particles. The laws of physics are used to compute the motion of the particles, along with a set of behaviors specified by game developers that influence properties of the system such as separation, alignment, and cohesion of particles. Examples of video games using this method include Capcom’s Dead Rising (2006) and Ubisoft’s Assassin’s Creed (2007).

More recently, velocity-based methods (Fiorini and Shiller 1998) have exhibited improvements in terms of local collision avoidance and behavior of virtual agents, and improved computational performance, over force-based collision avoidance methods. Rather than using virtual forces to prevent nearby virtual agents from collisions, velocity-based methods use the current velocity of each virtual agent in the group and then extrapolate the position of each virtual agent for some short time interval under the assumption that the virtual agent will maintain almost a constant velocity over some short time interval. Based on predicting the future positions of other virtual agents, each virtual agent tends to choose an avoiding new velocity based on some optimization. THQ’s video game Warhammer 40,000: Space Marine (2011) uses a velocity-based approach.

Reciprocal collision avoidance (van den Berg, Lin, and Manocha 2008) is an extension of the velocity-based approaches. The main difference with prior velocity-based methods lies in the fact that reciprocal collision avoidance considers the reciprocity between pairs of virtual agents. Each virtual agent is assumed to be attempting to avoid a collision with the other, rather than seeing the other virtual agent as a moving obstacle. Incorporating reciprocity into velocity-based approaches typically ensures smoother motion for the virtual agents and may also cause emergent phenomena in groups of virtual agents, such as arching, jamming, bottlenecks, and wake formation (Guy et al. 2010).

2 Local Collision Avoidance
2.1 Hybrid Reciprocal Velocity Obstacles
The velocity obstacle (Fiorini and Shiller 1998) of a virtual agent $A$, with position $p_A$, induced by a moving obstacle $B$, with position $p_B$, in a game level is the set of all velocities...
for the virtual agent that will result in a collision between the virtual agent and the moving obstacle within some short time interval into the future, assuming that the dynamic obstacle maintains a constant velocity $v_B$:

$$VO_{A|B} = \{v \mid \exists t > 0 : \tau(t(v - v_B)) \in D(p_B - p_A, r_A + r_B)\},$$

where $D(p, r)$ is an open disc of radius $r$ centered at $p$, as shown in Figure 1(b). It follows that if the virtual agent chooses a velocity within the region corresponding to the velocity obstacle, then the virtual agent and the moving obstacle will potentially collide. If the velocity chosen is outside the velocity obstacle, then a collision will not occur.

Unfortunately, the velocity obstacle approach does not work very well for local collision avoidance within a group of virtual agents where each virtual agent is actively changing its velocity to avoid the other virtual agents, since it assumes that other virtual agents may not change their velocities. If all virtual agents were to use velocity obstacles to choose a new velocity, there would be oscillations in the motion of the virtual agents between successive time steps. The reciprocal velocity obstacle (van den Berg, Lin, and Manocha 2008) addresses the problem of oscillations caused by the velocity obstacle by allowing for the reactive nature of the other virtual agents. Instead of one virtual agent having to take all the responsibility for avoiding collisions, reciprocal velocity obstacles let a virtual agent take just half of the responsibility for avoiding a collision and assume that the other virtual agent reciprocates by taking care of the other half:

$$RVO_{A|B} = \{v \mid 2v - v_A \in VO_{A|B}\},$$

where $v_A$ is the velocity of virtual agent $A$, as shown in Figure 1(c).

The reciprocal velocity obstacle approach guarantees that if both virtual agents select a velocity outside the reciprocal velocity obstacle induced by the other, and both virtual agents choose to pass each other on the same side, then the motion of both robots will be free of collisions and oscillations. However, in order to make progress, the virtual agents are usually required to select the velocity closest to their preferred velocity, defined as the velocity directed from each virtual agent towards its goal position. If a virtual agent has to move away from the goal to avoid collisions, then the preferred velocity will differ greatly from the current velocity. The presence of a third virtual agent may also cause at least one of the virtual agents to choose a velocity even farther from its current velocity. This means virtual agents may not necessarily choose the same side to pass, which may result in undesirable oscillations known as “reciprocal dances” that may not resolve quickly and may leave virtual agents effectively deadlocked.

The hybrid reciprocal velocity obstacle (Snape et al. 2011) resolves the problem of reciprocal dances by combining the velocity obstacle and the reciprocal velocity obstacle, taking one side from each to form a hybrid reciprocal velocity obstacle that is enlarged on one side to discourage virtual agents from passing each other on different sides. If the velocity of a virtual agent is to the right of the centerline of its reciprocal velocity obstacle induced by some other virtual agent, then the virtual agent should choose a velocity to the right of the reciprocal velocity obstacle. To encourage such behavior, the reciprocal velocity obstacle is enlarged by replacing the edge on the side that the virtual agents should not pass, for example, the left side in this case, by the edge of the corresponding velocity obstacle. If the velocity of the virtual agent is to the left of the centerline, the procedure is mirrored, exchanging left and right sides. The geometric interpretation of a hybrid reciprocal velocity obstacle $HRVO_{A|B}$ for a virtual agent $A$ with respect to a virtual agent $B$, including the location of the centerline and an indication of the enlarged area, is shown in Figure 1(d).

### 2.2 Optimal Reciprocal Collision Avoidance

$$ORCA^+_A|B = \{v \mid (v - (v_A + \frac{1}{2}u)) \cdot n \geq 0\}.$$

Optimal reciprocal collision avoidance (van den Berg et al. 2011) solves the problem of reciprocal dances addressed by the hybrid reciprocal velocity obstacle in a different way. This approach augments the velocity obstacle with a half-plane that defines a set of velocities that are both collision free and will additionally ensure that the motion of the virtual agents will be smooth in all but dense scenarios.
The optimal reciprocal collision avoidance half-plane ORCA\(_{A|B}\) for a virtual agent \(A\) with respect to a virtual agent \(B\) is defined as follows. As shown in Figure 2(c), let \(u\) be the vector from the relative velocity \(v_A - v_B\) of the virtual agents \(A\) and \(B\) to the closest point on the boundary of the truncated velocity obstacle for virtual agent \(A\) induced by virtual agent \(B\):

\[
u = (\arg\min_{v\in\partial VO_{A|B}} \|v - (v_A - v_B)\|) - (v_A - v_B).
\]

Let \(n\) be the outward normal of the boundary of the velocity obstacle at \(v_A - v_B + u\). It follows that \(u\) is the smallest change required to the relative velocity of virtual agents \(A\) and \(B\) to avoid a collision. Incorporating reciprocity, each virtual agent should adjust its velocity by at least \(\frac{1}{2}u\) to avoid the collision. Therefore the velocities permitted by optimal reciprocal collision avoidance are in a half-plane in the direction of \(n\) starting at the point \(v_A + \frac{1}{2}u\):

![Figure 2: (a) Two virtual agents \(A\) and \(B\). (b) The truncated velocity obstacle \(VO_{A|B}\) for virtual agent \(A\) induced by virtual agent \(B\). (c) The optimal reciprocal collision avoidance half-planes of permitted velocities \(ORCA_{A|B}\) for virtual agents \(A\) and \(B\).](image)

### 3 Implementation

#### 3.1 Libraries

The hybrid reciprocal velocity obstacle approach and optimal reciprocal collision avoidance have been implemented as C++ libraries, HRVO Library and RVO2 Library, respectively.

Essentially, the algorithm in RVO2 Library computes the optimal reciprocal collision avoidance half-planes for a virtual agent induced by the other virtual agents, and then intersects these half-planes to form a region of permitted velocities for the virtual agent. The algorithm then computes the preferred velocity \(v_{pref}\) (see Section 3.2) of the virtual agent and computes a new velocity \(v_{new}\) using two-dimensional linear programming (see Section 3.3) that is within the region of permitted velocities and as close as possible to the preferred velocity:

\[
v_{new}^{A} = \arg\min_{v\in ORCA_{A|B}} \|v - v_{pref}^{A}\|
\]

If there are many virtual agents nearby and there is no velocity within the region of permitted velocities, then some constraints are relaxed and a new velocity is found using three-dimensional linear programming (see Section 3.4).

The algorithm used in HRVO Library is broadly similar except that it uses the ClearPath geometric algorithm (Guy et al. 2009) to compute new velocities.

#### 3.2 Preferred Velocity

Both HRVO Library and RVO2 Library choose a new velocity by computing the velocity that is closest to the preferred velocity and is collision free. If the goal position of the virtual agent is visible, then the preferred velocity is in the direction of the goal. If the goal position is not visible, the preferred velocity should be directed to the nearest node on waypoint graph to the goal or to some point on the nearest edge on a navigation mesh path or a roadmap that leads to the goal.

#### 3.3 Linear Programming

RVO2 Library uses an efficient randomized linear programming algorithm (de Berg et al. 2008) that adds the constraints one by one in random order while keeping track of the current optimal new velocity for a virtual agent in the group. Linear programming is an optimization technique, commonly used in operations research, for finding one specific solution to a set of linear equality and inequality constraints that optimizes a given linear function of the variables. Geometrically, a linear programming algorithm computes a point in a polygon where the function has its maximum or minimum value if such a point exists. A randomized linear programming algorithm adds the linear constraints in a random order in order to compute the optimum solution.

For each virtual agent in the group, the randomized linear programming algorithm has a linear expected running time with respect to the number of virtual agents that are input into the algorithm. The algorithm computes the velocity in that is closest to the preferred velocity of the virtual agent, and reports failure if the linear program is infeasible.
3.4 Dense Scenarios

In dense scenarios, when a group of virtual agents is packed tightly together in part of the game level, there may not be a velocity that satisfies all the constraints of the linear program and the algorithm would return that the linear program is infeasible. When this occurs, RVO2 Library computes the safest possible velocity for the virtual agent, the velocity that minimally penetrates the constraints induced by the other virtual agents. This can be interpreted geometrically as moving the edges of the half-planes perpendicularly outward with equal speed, until exactly one velocity becomes valid. This velocity may be computed using a three-dimensional linear program. The same randomized linear programming algorithm as before may be used by projecting the problem down onto plane, such that all geometric operations can be performed in two-dimensions. The three-dimensional linear program is always feasible, so it always returns a solution. The running time of the algorithm is still linear with respect to the number of virtual agents.

3.5 Game Engine Integration

RVO2 Library has been integrated into several game engines to either perform local collision avoidance and navigation for groups of virtual agents or improve upon the default implementations provided by the game engine developers. Examples include a multi-platform package for Unity Technologies’ Unity 3 written in C# and a DLL for Epic Games’ Unreal Development Kit written in C++ for Microsoft Windows with UnrealScript bindings. A screenshot from the Unreal Development Kit integration is shown in Figure 3. An approach broadly similar to the hybrid reciprocal velocity obstacle, as used in HRVO Library, has been incorporated into the Detour component of the game navigation toolset Recast and Detour to provide local collision avoidance within a navigation mesh, as shown in Figure 4.

4 Conclusion

We have presented the hybrid reciprocal velocity obstacle and optimal reciprocal collision avoidance methods for reciprocal collision avoidance and navigation in video games and described their implementations in C++ as HRVO Library and RVO2 Library. The libraries can efficiently simulate groups of twenty-five to one thousand virtual agents in dense conditions and around moving and static obstacles. RVO2 Library is on average at least twice as fast as HRVO Library, but HRVO Library results in fewer collisions between virtual agents of RVO2 Library and therefore results in better local interactions between the virtual agents.

5 Acknowledgments

This work was supported in part by Army Research Laboratory Contract W911NF-04-1-0088, by National Science Foundation Awards 1000579 and 1117127, and by Intel Corporation.

References


