# Reciprocal Collision Avoidance and Navigation for Video Games

*Jamie Snape,[1] Stephen J. Guy,[1] Deepak Vembar,[2] Adam Lake,[2] Ming C. Lin,[1] and Dinesh Manocha[11]*

## 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 *Dead Risin*g* (Capcom, 2006) and *Assassin's Creed** (Ubisoft, 2007).

More recently, velocity-based methods (Fiorini & Shiller, 1998; van den Berg, Lin, & Manocha, 2008) 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 that time. Based on predicting the future positions of other virtual agents, each virtual agent tends to choose an avoiding new velocity based on some optimization. The recent video

[1] Jamie Snape, Stephen J. Guy, Ming C. Lin, and Dinesh Manocha are with the Department of Computer Science, University of North Carolina at Chapel Hill, Chapel Hill, NC 27599 (e-mail: snape@cs.unc.edu; sjguy@cs.unc.edu; lin@cs.unc.edu; dm@cs.unc.edu).

[2] Deepak Vembar and Adam Lake are with the Visual Computing Group, Intel Corporation, Hillsboro, OR 97124 (e-mail: deepak.s.vembar@intel.com; adam.t.lake@intel.com).

game *Warhammer 40,000: Space Marine** (THQ, 2011) uses a velocity-based approach.

Reciprocal collision avoidance (van den Berg, Lin, & 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 (van den Berg, Patil, Sewall, Manocha, & Lin, 2008; Guy, Chhugani, Curtis, Dubey, Lin, & Manocha, 2010; van den Berg, Lin, & Manocha, 2008).

## 2. Local Collision Avoidance

### A. Velocity Obstacles (VO) and Reciprocal Velocity Obstacles (RVO)

The *velocity obstacle* (Fiorini & Shiller, 1998) of a virtual agent induced by a moving obstacle 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. 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. A geometric interpretation of a velocity obstacle $VO_{A|B}$ for a virtual agent $A$ with respect to a virtual agent $B$ corresponding to a cone is shown in Figure 1 (center).

The velocity obstacle has been successfully used to navigate one virtual agent through a game level containing multiple moving obstacles by having the virtual agent select a velocity in each time step that is outside any of the velocity obstacles induced by different moving obstacles. 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, & Manocha, 2008; de Berg, Cheong, van Kreveld, & Overmars, 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. The geometric interpretation of a reciprocal velocity obstacle $RVO_{A|B}$ for a virtual agent $A$ with respect to a virtual agent $B$ as a velocity obstacle with its apex translated is shown in Figure 1 (right). It is also possible to share the responsibility of collision avoidance between two virtual agents in some other manner, for example, 75:25 rather than 50:50.

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.

If each virtual agent chooses the new velocity closest to its current velocity, then the virtual agents will automatically pass each other on the same side. Rather than choosing velocities closest to their current velocities, in order to make progress, the virtual agents are usually required to select the velocity closest to their preferred velocity, usually the velocity directed from each virtual agent towards its goal position (see Section 3.B below for further details). 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.

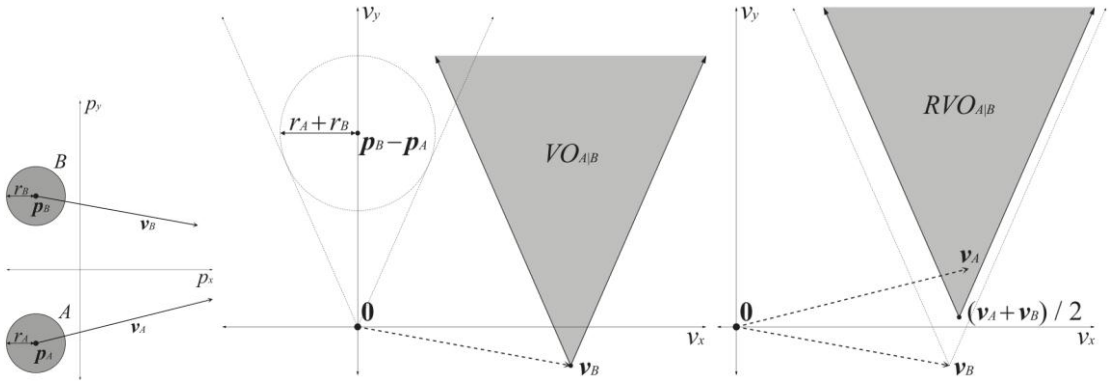


**Figure 1: Two virtual agents *A* and *B* (left). The velocity obstacle $VO_{A|B}$ for virtual agent *A* induced by virtual agent *B* (center). The reciprocal velocity obstacle $RVO_{A|B}$ for virtual agent *A* induced by virtual agent *B* (right).**

## B. Hybrid Reciprocal Velocity Obstacles (HRVO)

The *hybrid reciprocal velocity obstacle* (Snape, van den Berg, Guy, & Manocha, 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 2.

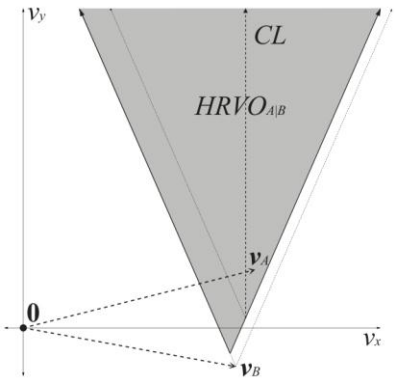


**Figure 2: The hybrid velocity obstacle $HRVO_{A|B}$ for virtual agent $A$ induced by virtual agent $B$. The current velocity $v_A$ is right of the centerline $CL$, so the left side of $HRVO_{A|B}$ is the left side of $VO_{A|B}$ (shown in Figure 1 above) and the right side of $HRVO_{A|B}$ is the right side of $RVO_{A|B}$ (also in Figure 1).**

## C. Optimal Reciprocal Collision Avoidance (ORCA)

*Optimal reciprocal collision avoidance* (van den Berg, Guy, Lin, & Manocha, 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 3, let the $\boldsymbol{u}$ be the vector from the relative velocity $\boldsymbol{v}_A - \boldsymbol{v}_B$ 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 $B$. Let $\boldsymbol{n}$ be the outward normal of the boundary of the velocity obstacle at $\boldsymbol{v}_A - \boldsymbol{v}_B + \boldsymbol{u}$. It follows that $\boldsymbol{u}$ 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 $\frac{1}{2}\boldsymbol{u}$ to avoid the collision. Therefore, the velocities permitted by optimal reciprocal collision avoidance are in a half-plane in the direction of $\boldsymbol{n}$ starting at the point $\boldsymbol{v}_A + \frac{1}{2}\boldsymbol{u}$.

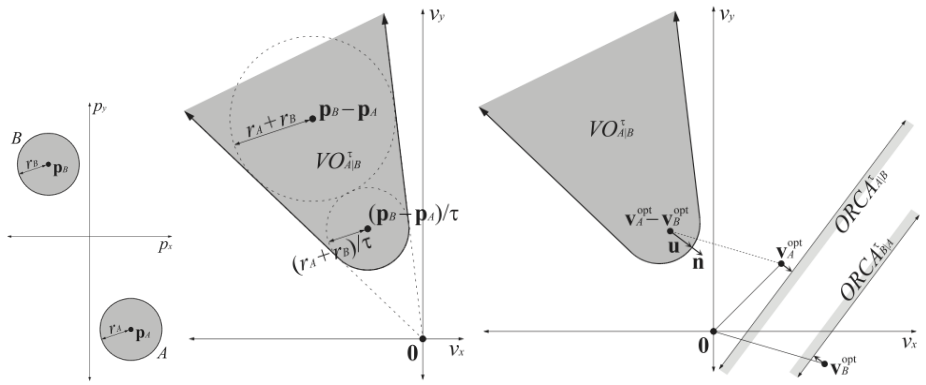


**Figure 3: Two virtual agents *A* and *B* (left). The truncated velocity obstacle *VO* for virtual agent *A* induced by virtual agent *B* (center). The optimal reciprocal collision avoidance half-planes of permitted velocities *ORCA* for virtual agents *A* and *B*. (right).**

# 3. Implementation

## A. *HRVO Library* and *RVO2 Library*

The hybrid reciprocal velocity obstacle approach and optimal reciprocal collision avoidance have been implemented as C++ libraries, *HRVO Library*[3] and *RVO2 Library*,[4] 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 (see Section 3.B below) of the virtual agent and computes a new velocity using two-dimensional linear programming (see Section 3.C below) that is within the region of permitted velocities and as close as possible to the preferred velocity. 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 3.E below). The overall algorithm for *RVO2 Library* is shown in

**Figure 4**.

The algorithm used in *HRVO Library* is broadly similar except that it uses the *ClearPath* geometric algorithm (Guy, et al., 2009; Reynolds, 1987; Snape, van den Berg, Guy, & Manocha, 2011; van den Berg, Guy, Lin, & Manocha, 2011) to compute new velocities.

[3] See http://gamma.cs.unc.edu/HRVO/.

[4] See http://gamma.cs.unc.edu/RVO2/.

```
input LA : list of virtual agents, t : time step
loop
    for all A in LA do
        Get position and velocity for A
        Compute n nearest neighbors NN in LA for A
        for all A' in NN do
            Get position and velocity for A'
            Construct velocity obstacle VO_{A|A'}
            Construct ORCA_{A|A'}
        end for
        Construct ORCA_A from the intersection of all ORCA_{A|A'}
        Compute preferred velocity for A
        Compute new velocity for A in ORCA_A that is closest to the preferred
        velocity using 2D linear programming
        if 2D linear program is infeasible then
        Compute new velocity for A that is closest to the preferred velocity
        using 3D linear programming
        end if
    end for
    for all A in LA do
        Set velocity to new velocity for A and increment position by
        t × new velocity
    end for
end loop
```

**Figure 4: The algorithm used in *RVO2 Library*.**

## B. 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 a 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.

## C. Linear Programming

*RVO2 Library* uses an efficient randomized linear programming algorithm (de Berg, Cheong, van Kreveld, & Overmars, 2008; van den Berg, Guy, Lin, & Manocha, 2011) 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.

### D. Nearest-Neighbor Computation

The efficiency and scalability of computing new velocities for a virtual agent may be increased by not including the constraints of all other virtual agents, but instead only of those virtual agents that are close by. Distant virtual agents have little chance of interacting with the virtual agent in the short time interval, so the possibly of collision is not increased substantially. *RVO2 Library* uses a *k*d-tree for nearest-neighbor computation. The tree is built at the start of each time step and then queried during the new velocity computations for each virtual agent during that time step.

### E. 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.

### F. Task Parallelism

Intel® VTune™ Amplifier XE was used to analyze existing bottlenecks in *RVO2 Library* and optimize it to take advantage of multi-core processors. The computation steps involved in finding the nearest neighbors, computing new velocities and updating the position (Figure 5) are performed iteratively for each virtual agent and are independent of each other in the simulation. *RVO2 Library* takes advantage of modern multi-core processors by splitting the algorithm steps for each virtual agent using either OpenMP* or Intel® Threading Building Blocks (Intel® TBB).

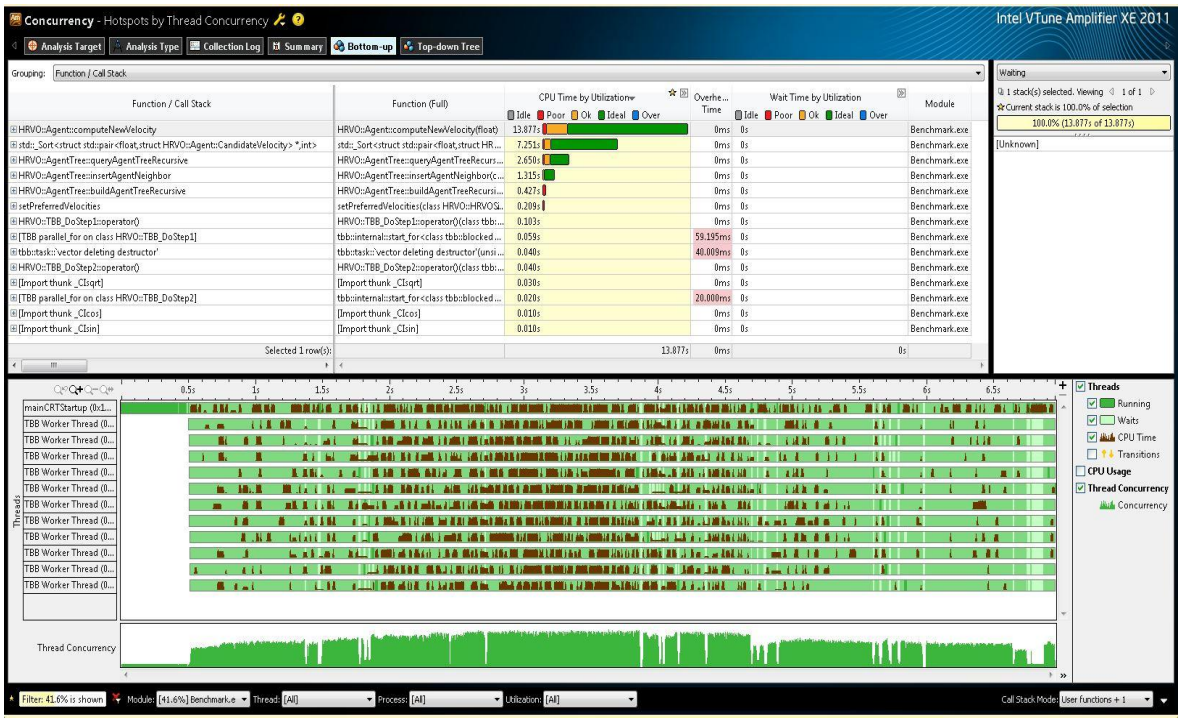


**Figure 5: Intel® VTune™ Amplifier XE concurrency analysis screenshot of Benchmark application on an Intel® Core™ i7-980X Processor (6C/12T). The application shows good concurrency (bottom green bars) with overall good per-function threading provided by Intel® Threading Building Blocks (Intel® TBB).**

Figure 5 shows a run of Intel® VTune™ Amplifier XE used to analyze the *Benchmark* scenario with *RVO2 Library*. The *Benchmark* scenario consisted of 1000 virtual agents arranged on the perimeter of a circle of known radius, with the goal to navigate to the opposite side of the circle avoiding collision with the other virtual agents. The *Benchmark* scenario was run on an Intel® Core™ i7-980X Processor Extreme Edition with 6 cores, each with Intel® Hyper-Threading Technology (6C/12T). Concurrency and Hotspots analysis from Intel® VTune™ Amplifier XE shows that the application is well threaded, with the major significant hotspots in the code having optimal CPU utilization time (green bars at the top). We find that all 12 threads in the CPU are fully subscribed with very few inactive periods (green bar chart at the bottom).

To understand the effect of different CPUs on algorithm performance, the *Benchmark* scenario was run on two different platforms with identical configurations except for the CPU. Performance data was collected from the Intel® Core™ i7-980X Processor Extreme Edition (codename Gulftown) and 2nd Generation Intel® Core™ i7-2820QM Processor (codename Sandy Bridge). The number of virtual agents in the simulation was varied from 50, 100, 500, to 1000 agents. The average computation time per time step, as well as average collisions per time step, were collected for *RVO2 Library* and *HRVO Library*.

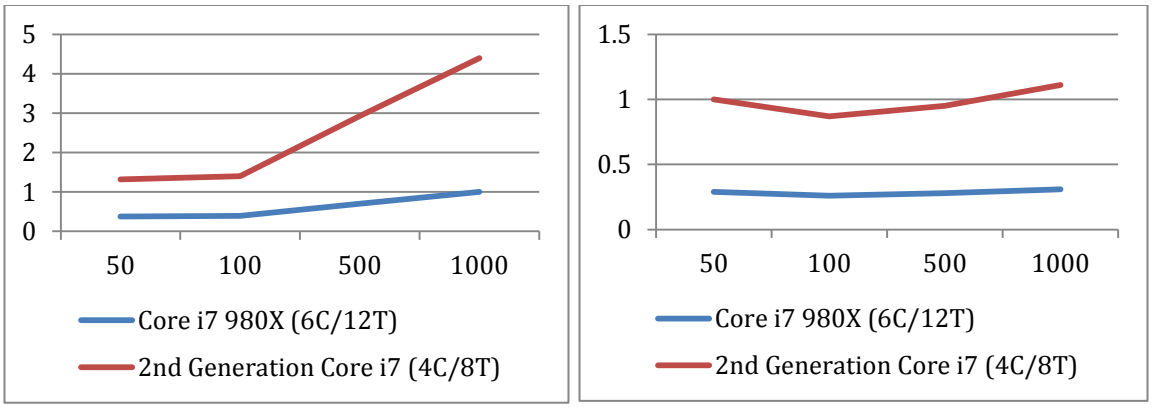


**Figure 6: Computation time per time step in *HRVO Library* (left) and *RVO2 Library* (right) with Benchmark scenario on two different Intel® processors. Number of agents are denoted on the *x-axis* and time in us on the *y*-axis. Increasing the number of CPU cores reduces the computation per time step.**

Figure 6 shows the performance of *HRVO Library* and *RVO2 Library* on the two different platforms. Overall, the computation time per time step is smaller on the Intel® Core™ i7-980X Processor compared to the Intel® Core™ i7-2820QM Processor due to the extra cores available to distribute the computational task, and becomes more pronounced as we increase the number of virtual agents in the simulation. We also find an almost linear increase in the computation time taken per time step as we increase the number of virtual agents in the simulation.

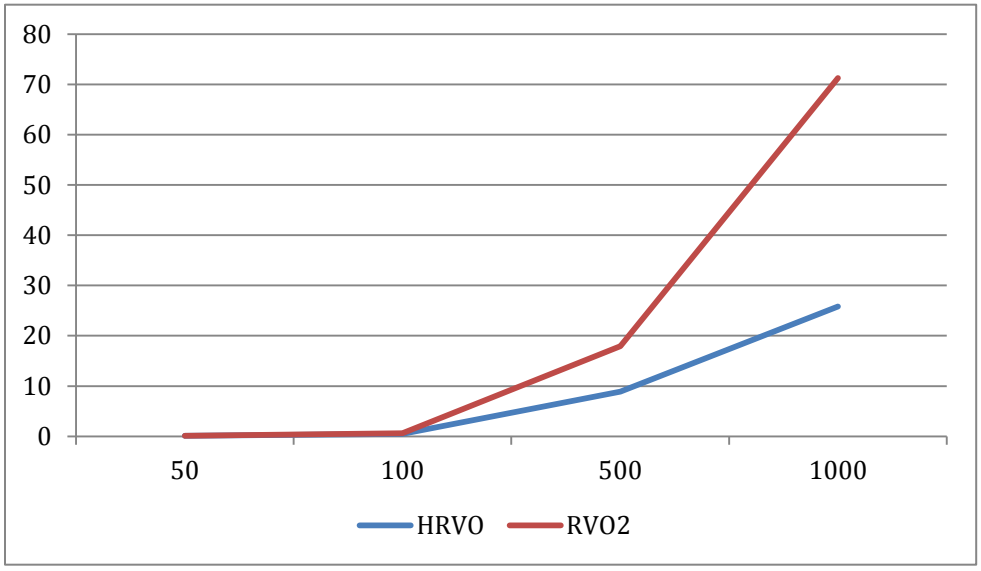


**Figure 7: Average number of collisions per time step for 50, 100, 500 and 1000 agents for *Benchmark* scenario with *HRVO Library* and *RVO2 Library* on Intel® Core™ i7-980X Processor Extreme Edition Processor (6C/12T). On average, *HRVO Library* had fewer collisions compared to *RVO2 Library*.**

Figure 7 shows the average number of collisions per time step of *HRVO Library* and *RVO2 Library*. For 50 and 100 agents, both algorithms have similar performance. However, the number of collisions almost doubles for *RVO2 Library* for 500 and 1000 virtual agents compared to *HRVO Library*.

## G. 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 third-party multi-platform package for *Unity 3*[5] (Unity Technologies, 2010) written in C# and a third-party DLL for the *Unreal Development Kit**[6] (Epic Games, 2010) written in C++ for Microsoft Windows* with *UnrealScript* bindings. A screenshot from the *Unreal Development Kit** integration is shown in

Figure 8.[7] 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*[8] (Mikko Mononen, 2009) to provide local collision avoidance within a navigation mesh, as shown in Figure 9.

**Figure 8: A screenshot of the benchmark scenario for *RVO2 Library* integration with *Unreal Development Kit**, 200 virtual agents navigating in real time between randomly chosen locations at the four corners of the game level (Intel® Core™ i5-760 Processor at 2.8GHz).**

[5] See http://rvo-unity.chezslan.fr/.

[6] As of mid 2011, *Unreal Development Kit** contains an implementation of the reciprocal velocity obstacle approach. Details of the integration of *RVO2 Library* into the game engine are available at http://gamma.cs.unc.edu/RVO2-UDK/.

[7] See http://youtu.be/x8dczNzxM0w for a video.

[8] See http://code.google.com/p/recastnavigation/.

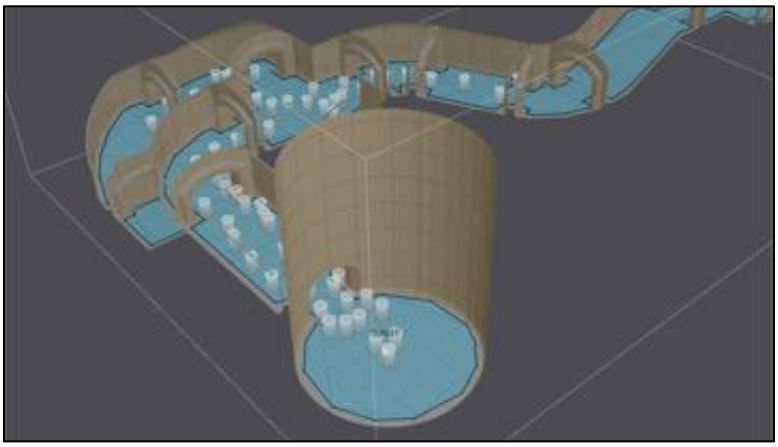

**Figure 9: A screenshot of the *Dungeon* scenario included with *Recast and Detour*, 50 virtual agents navigating in real time from one end of the game level to the other on a navigation mesh (Intel® Core™ i5-760 Processor at 2.8GHz).**

## 4. Performance

The performance of *HRVO Library* and *RVO2 Library* were compared in three scenarios, as follows:

1. *Circle-N:* 100 to 1000 virtual agents are spaced evenly on the perimeter of a circle within the game level. They must navigate to antipodal positions on the circle. The virtual agents will meet and have to negotiate around each other in the center.
2. *Moving Obstacle:* 25 virtual agents are spaced evenly on one side of a rectangular area within the game level. They must navigate to the opposite side of the rectangle while avoiding a moving obstacle that crosses their path. The moving obstacle will not attempt to avoid colliding with the virtual agents itself.
3. *Blocks:* Four groups of 25 virtual agents must navigate from the corners of a square area within the game level to the diagonally opposite corners. Blocking their path are four blocks, around which they must pass.

All calculations were performed on an Intel® Core™ i7-2600 Processor (codename Sandy Bridge), running at 3.40GHz with 8GB of RAM on the Microsoft Windows* 7 SP1 64-bit operating system. Each library was compiled using the Microsoft Visual C++* Compiler, version 10 SP1.

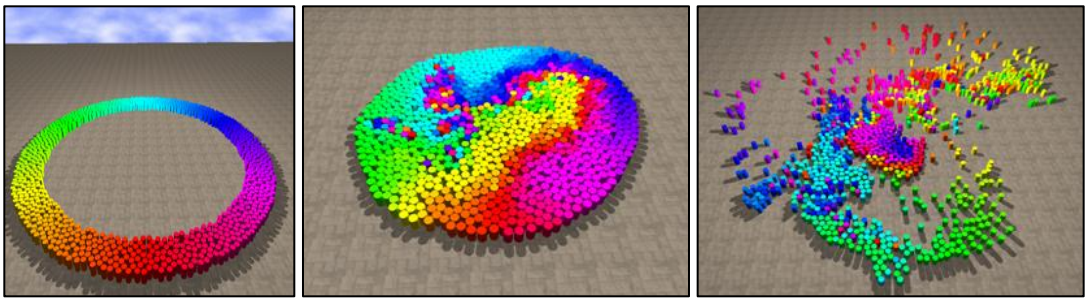

**Figure 10: Screenshots of a simulation of *Circle-1000*, 1000 virtual agents moving simultaneously in real time across a circle to the antipodal positions (Intel® Core™ i7-2600 Processor at 3.4GHz).**

**Table 1: Timing and collisions in simulations of the *Circle-100*, *Moving Obstacle*, and *Blocks* scenarios.**

| | | *HRVO Library* | *RVO2 Library* |
|---|---|---|---|
| *Circle-100* | Computation time (ms) | 93.1 | 29.3 |
| | Collisions per time step | 0.18 | 0.32 |
| *Moving Obstacle* | Computation time (ms) | 26.8 | 17.6 |
| | Collisions per time step | < 0.01 | 0.01 |
| *Blocks* | Computation time (ms) | 458.9 | 157.2 |
| | Collisions per time step | 5.71 | 9.65 |

Table 1 above shows the total computation time and number of collisions per time in each of the three scenarios. On average, through the three scenarios, *RVO2 Library* is around two to three times faster than *HRVO Library* due to the efficiency of the randomized linear programming algorithm. However when counting collisions between virtual agents, there are around half as many when using *HRVO Library* compared to *RVO2 Library*. In practice, *HRVO Library* is far less conservative in terms of the amount of area in velocity space that it forbids, so it is less likely that its algorithm will become infeasible or that some constraints need to be relaxed.

**Table 2: Timing of simulations of increasing numbers of virtual agents moving simultaneously across a circle of increasing circumference in the *Circle-N* scenario.**

| Number of virtual agents ($N$) | Average computation time per time step (μs) | |
|---|---|---|
| | *HRVO Library* | *RVO2 Library* |
| 100 | 88.1 | 31.6 |
| 200 | 145.8 | 51.4 |
| 300 | 195.9 | 78.1 |
| 400 | 240.1 | 97.3 |
| 500 | 305.1 | 118.5 |
| 1000 | 558.5 | 240.7 |

Table 2 above shows the computation time per time step for the *Circle-N* scenario, with $N$ varying from 100 to 1000. The circumference of the circle also varies in

proportion to *N*. In all cases, the computation time is in the order of microseconds with *HRVO Library* and slower than *RVO2 Library*.

**Table 3: Collisions in simulations of increasing numbers of virtual agents moving simultaneously across a circle of fixed circumference in the *Circle-N* scenario.**

| Number of virtual agents (*N*) | Average number of collisions per time step | |
|---|---|---|
| | *HRVO Library* | *RVO2 Library* |
| 100 | 0.18 | 0.32 |
| 200 | 0.93 | 1.35 |
| 300 | 1.93 | 3.31 |
| 400 | 3.05 | 5.61 |
| 500 | 4.36 | 9.40 |
| 1000 | 15.14 | 28.4 |

Table 3 above shows the average number of collisions per time step for the *Circle-N* scenario with *N* varying as before. In this case, the circumference of the circle was fixed to create increasing dense scenarios. As before there were fewer collisions with *HRVO Library* are reported as compared to *RVO2 Library*. However, the number of collisions was extremely low for both methods in these scenarios.

## 5. Conclusion

We have presented the hybrid reciprocal velocity obstacle and optimal reciprocal collision avoidance methods for reciprocal collision avoidance and navigation in video games. We have compared the performance of *HRVO Library* and *RVO2 Library* that implement these methods in C++ and shown they can efficiently simulate groups of 25 to 1000 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.

## 6. Works Cited

de Berg, M., Cheong, O., van Kreveld, M., & Overmars, M. (2008). *Computational Geometry: Algorithms and Applications* (3rd ed.). Berlin, Germany: Springer.

Fiorini, P., & Shiller, Z. (1998). Motion planning in dynamic environments using velocity obstacles. *International Journal of Robotics Research , 17* (7), 760-772.

Guy, S., Chhugani, J., Curtis, S., Dubey, P., Lin, M., & Manocha, D. (2010). PLEdestrians: A least-effort approach to crowd simulation. *ACM SIGGRAPH/Eurographics Symposium on Computer Animation SCA* (pp. 119-128). New York: ACM Press.

Guy, S., Chhugani, J., Kim, C., Satish, N., Dubey, P., Lin, M., et al. (2009). ClearPath: Highly parallel collision avoidance for multi-agent simulation. *ACM SIGGRAPH/Eurographics Symposium on Computer Animation SCA* (pp. 177-187). New York: ACM Press.

Reynolds, C. (1987). Flocks, herds and schools: A distributed behavioral model. *International Conference on Computer Graphics and Interactive Techniques SIGGRAPH* (pp. 25-34). New York: ACM Press.

Snape, J., van den Berg, J., Guy, S., & Manocha, D. (2011). The hybrid reciprocal velocity obstacle. *IEEE Transactions on Robotics , 27* (4), 696-706.

van den Berg, J., Guy, S., Lin, M., & Manocha, D. (2011). Reciprocal n-body collision avoidance. In C. Pradalier, R. Siegwart, & G. Hirzinger (Eds.), *Robotics Research: The 14th International Symposium ISRR* (pp. 3-19). Berlin, Germany: Springer.

van den Berg, J., Lin, M., & Manocha, D. (2008). Reciprocal velocity obstacles for real-time multi-agent navigation. *IEEE International Conference on Robotics and Automation ICRA* (pp. 1928-1935). New York: IEEE Press.

van den Berg, J., Patil, S., Sewall, J., Manocha, D., & Lin, M. (2008). Interactive navigation of multiple agents in crowded environments. *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games I3D* (pp. 139-147). New York: ACM Press.