Simulating Group Formations Using RVO

Martin Funkquist
martinfu@kth.se

Supervisor:
Christopher Peters

Staffan Sandberg
stsand@kth.se

May 25, 2016

Figure 1: From left to right. First figure displays a real scenario of pedestrians walking in groups. Second figure shows one of our simulated scenes which is in front of the KTH building. Last image is our simulated case of the first picture.

Abstract

In this project we are implementing group formations in to the game engine Unity3D. Groups of people move in a certain way when avoiding other groups and obstacles. With the use of the steering package UnitySteer we have made some example scenes using different steering behaviors. We have made an implementation of the RVO2 package in Unity. With this implementation we can simulate various scenarios with agents avoiding each other. Human pedestrians tend to walk in groups. We adapted this behavior in our program to make our simulations as real as possible. Finally we evaluated the performance of our simulation by checking the frames per seconds when increasing the number of agents.

Keywords: Agent, Steering, Obstacle Avoidance, RVO, Groups, Formations, Real-Time, Unity.

1 Introduction

Crowd simulations are widely used in our society today. It ranges from areas such as evacuation emergencies, e.g. people evacuation a burning building. To entertaining purposes in games and movies, where crowds have to be generated and behave like real humans as well. Specifically in games it is important for everything to run in real time, which has been one of our requirements for this project.

Previous research in this area often involve large crowds behaving like fluids. In our project we have a slightly different approach, our focus has been on different steering behaviors and the RVO based avoidance, with group formations involving two to three people avoiding nearby groups. The progress we make is documented on our blog[1]

2 Related Works

Studies in the area of *steering behaviors* has been published in this article [9]. It describes the basics in steering behaviors such as *seek* and *flee* but it also brings up more complex areas such as *path following* and *obstacle avoidance*. The behaviors are based on the laws of physics, using forces applied to the agents to steer them in the right direction. The agents are simulated as particles and a group of agents can be interpreted as a particle system, each agent affecting each other with a force. These kind of steering behaviors is commonly used in games, when only basic steering is needed.

In [9] obstacle avoidance is mentioned. But this kind of obstacle avoidance is limited. It works fine on static objects but when there is multiple agents moving, each at a high speed, this kind of obstacle avoidance does not work very well. Another algorithm for avoiding obstacles called *Reciprocal Velocity Obstacles (RVO)* has thus been produced and is describes in the articles [14] and [3]. This is a velocity-based approach [5] in which the velocities of each agent is used to predict the future position of the agent. It is assumed that the velocity of the agent does not change drastically. What differs the RVO method from the previous velocity-based methods is that the algorithm assumes that each pair of agents is avoiding a collision with each other instead of seeing the other agents as moving obstacles. This approach makes the motion of the agents smoother.

While RVO is a good method for agents avoiding each other it has its flaws. That is when the *Hybrid Reciprocal Velocity Obstacle* (HRVO) [11] approach comes into play. It is a mixture of the RVO and *Velocity Obstacle*(VO). More about this approach in section 3.3.2. A similar approach is the *Optimal Reciprocal Collision Avoidance* [13] which fixes that same problem as the $HRVO$ approach does.

In further studies, the RVO algorithm has been extended to 3D [10]. There has also been research with including the *acceleration* in the velocity obstacle [12].

A lot of research has been done in the area of pedestrian behavior [7] [16] [2] [15] among others. Here we find different approaches to understanding social forces and other human behavior related stuff.

3 Implementation

Our project consist of different parts. At the beginning we wanted to learn the basics of crowd

[1] http://crowdsimulationproject.blogspot.se/

simulations, which is steering behaviors. Then we advanced into areas such as obstacle avoidance where we made use of the RVO algorithm. In the latter we implemented group formations involving two and three agents. These groups walk towards each other in a crosswalk scene avoiding each other.

All of our implementation is made in Unity3D and the code is written solely in C#.

3.1 Steering Behaviors

First we made our own implementation of steering behaviors based on the theory in the article by Craig W. Reynolds [9]. The seek behavior was easily implemented using vector algebra. With the target position and the agents position given we could calculate a vector between these points and make the agent move in that direction. The same idea was used with the flee behavior, but instead of moving towards the target the agent moved the opposite direction away from the target.

Next up was to implement the obstacle avoidance behavior. This was a little bit trickier. We tried a couple of different methods for this. First we tried the method described in [1] which is based on the obstacle avoidance behavior describes in the article by Craig W. Reynolds. This did not work out as we had hoped. The agent moved through the obstacles multiple times.

We continued looking for another algorithm for obstacle avoidance. Then we found an algorithm which was based on ray casting, in this video [2]. It uses ray casting, which is built in into Unity, to find the obstacles. When a ray hits the obstacle the normal of the point where the collision occur is used. This vector is first scaled by a value to create a force vector. Then this force vector is added to the current velocity vector of the agent. Thus the agent will be rotated towards a new direction in which the agent won't collide with the obstacle. A drawing of this is shown in Figure 2.

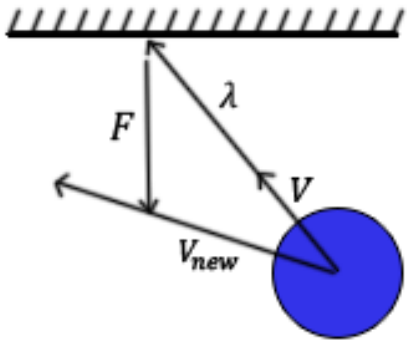

Figure 2: Simple drawing of a ray casing scenario. $\mathbf{v} = velocity$, $\boldsymbol{\lambda} = ray$, $\mathbf{F} = force$, $\mathbf{V}_{new} = new\ velocity$

This method was better than the first one we implemented. But it also had its limitations. For example, if the target was to close to the obstacle then the agent would go in circles around the obstacle object. By reducing the length of the ray cast this error could be minimized but then the agent is more likely to go through the object instead.

When we were done with trying to implement our own steering behaviors we started to experiment the the UnitySteer library [3]. This is a library which is based on the OpenSteer library written in C++ [4]. OpenSteer uses the theory from the thesis by Craig W. Reynolds [9]. In the beginning it was difficult to understand how to use UnitySteer since there was hardly any documentation about it. All the different steering

[2] https://vimeo.com/9304844

[3] https://github.com/ricardojmendez/UnitySteer

[4] http://opensteer.sourceforge.net/

behaviors are divided into different scripts which makes it easy to apply a specific steering behavior to an agent. But some of the scripts did not seem to work when applied to the agent. When there were too many scripts on one agent it all seemed to be just a big mess, since all different behaviors adds a force to the agent.

The obstacle avoidance behavior in UnitySteer were not as we had hoped. It made the agent go in a very stuttering path. When the agent came close to an obstacle the script applied a large force so that the agent almost turned in the opposite direction, which is not a behavior humans usually have. It is as if someone were to walk towards a big obstacle and just before he would collide with it he would turn 170 degrees and walk back a couple of meters. Then turn back again and do the same thing all over again until he have made his way around the obstacle. A video of this behavior can be found here [5].

One more limitation with UnitySteer's obstacle avoidance was that is was not working on other agents. We could not get the agents to avoid each other. Only the static obstacles were avoided.

3.2 Group Formations

At this point in the project we had decided that we wanted to focus on doing group formations with the agents. This is an essential part in simulating pedestrians walking since we humans usually walks in groups of 2 to 3 persons. To implement this we made an abstract base class which handles all the calculations of the formation. What differs between the formations is that they have different relative positions to each other. In each formation we have a leader which all the other agents follow. So basically, each group has a leader who has a target. All the other agents in the group have their target relative to the leaders position. We made a video of our agents changing formations which can be found here [6].

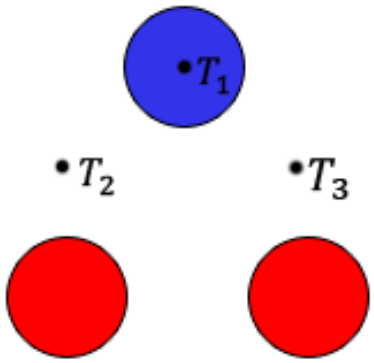

Figure 3: Example with template positions in triangle formation. Blue Circle: Leader, Red Circles: Followers. $\{\mathbf{T}_1, \mathbf{T}_2, \mathbf{T}_3\}$: Template Positions. $\mathbf{T}_2$ and $\mathbf{T}_3$ will be assigned as targets for the followers.

3.3 Reciprocal Velocity Obstacle (RVO)

When we had noted UnitySteer's limitations we wanted to improve our avoidance behaviors. This was the part of our simulations that lacked at this point in the project. RVO is a commonly used algorithm for obstacle avoidance behavior so we decided to go with that.

Our implementation consisted of converting an already implemented RVO library to Unity [7]. At first, we thought that we could rewrite the already existing implementation so that we could use it in Unity. It turned out that it was

[5] https://www.youtube.com/watch?v=j3sDXk4gAF4

[6] https://www.youtube.com/watch?v=W6ApBGcz57o

[7] https://github.com/snape/RVO2-CS

way too much work to do so, since Unity requires everything to inherit from certain classes and it uses its own threading system. We tried to run a scenario but it did not work [8].

We then made a script to take the information from the completed RVO implementation and apply it to our agents. The implementation contained two examples, one of which we made in Unity [9].

In our final scenario we wanted to make a simulation of groups of agents walking towards each other, similar to the situation at a crowded crosswalk where there are a lot of people. In such a scene, if it is studied closely we can see that humans tend to walk in groups. Video can be found here [10]

3.3.1 Theory of RVO

The RVO method is based on velocity obstacles. These are defined as follows: Given an agent A with position $\mathbf{p}_A$ and an moving obstacle B with position $\mathbf{p}_B$. Then the velocity obstacle $VO^A_B(\mathbf{v}_B)$ is defined as the set of all velocities $\mathbf{v}_A$ such that A will collide with B in some moment in time. Further, we denote the a ray $\lambda(\mathbf{p}, \mathbf{v})$ with starting point $\mathbf{p}$ and direction $\mathbf{v}$ as the following:

$$\lambda(\mathbf{p}, \mathbf{v}) = \{\mathbf{p} + t\mathbf{v} | t \geq 0\} \tag{1}$$

With this defined we can conclude that if a ray starting at $\mathbf{p}_A$ is cast in the direction of the relative velocity between A and B, which is $\mathbf{v}_A - \mathbf{v}_B$, and this ray intersects the Minkowski sum of B and $-A$ (which basically is a circle of radius $r_A + r_B$, in case the agents are represented as circles) centered at $\mathbf{p}_B$. Then $\mathbf{v}_A$ belongs to the velocity obstacle of B. We can write this mathematically as follows:

$$VO^A_B(\mathbf{v}_B) = \{\mathbf{v}_A | \lambda(\mathbf{p}_A, \mathbf{v}_A - \mathbf{v}_B) \cap B \oplus -A \neq \emptyset\} \tag{2}$$

What this means is that if $\mathbf{v}_A \in VO^A_B(\mathbf{v}_B)$ then there will be a collision between A and B if B keeps the same velocity. Thus, A has to choose a velocity which is outside of the $VO^A_B(\mathbf{v}_B)$ domain. Figure 4 shows these concepts geometrically.

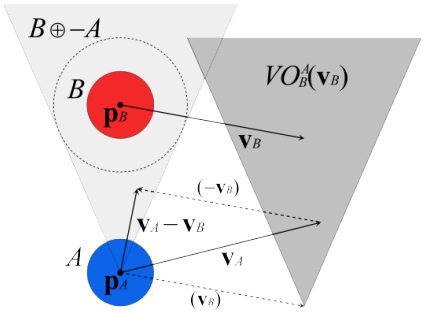

Figure 4: Illustration of the velocity obstacle [14]

However, this approach is not perfect. It can result in oscillations. Assume that two agents A and B are moving toward each other. Their velocities are $\mathbf{v}_A$ and $\mathbf{v}_B$ respectively. Both velocities belong to the velocity obstacle for the other agents, such that $\mathbf{v}_A \in VO^A_B(\mathbf{v}_B)$ and $\mathbf{v}_B \in VO^B_A(\mathbf{v}_A)$. Therefor, these agents will take a new velocity which is outside of the velocity obstacle. Lets call these new velocities $\mathbf{v}'_A$ and $\mathbf{v}'_B$. When the agents adopt these new velocities their old velocities $\mathbf{v}_A$ and $\mathbf{v}_B$ is not in the velocity obstacle anymore. Assume that $\mathbf{v}_A$ and $\mathbf{v}_B$ is the optimal velocities for the agents to reach their goals. Then they will adopt these velocities in the next step and then we are back to the start

[8] https://www.youtube.com/watch?v=157yr85u5gI
[9] https://www.youtube.com/watch?v=W6ApBGcz57o
[10] https://www.youtube.com/watch?v=ylBcDCA1e7U

scenario where both agents velocities belong to the velocity obstacle of the other agent. If this behavior continues it will cause oscillations.

To prevent this problem with the oscillations the *Reciprocal Velocity Obstacle* is introduced. The idea of this is: instead of choosing a velocity outside of the other agent's velocity obstacle, an *average* between the agent's current velocity and a velocity outside of the velocity obstacle is chosen. Mathematically this is defined as follows:

$$RVO^A_B(\mathbf{v}_B, \mathbf{v}_A) = \{\mathbf{v}'_A | 2\mathbf{v}'_A - \mathbf{v}_A \in VO^A_B(\mathbf{v}_B)\} \tag{3}$$

This reciprocal velocity obstacle $RVO^A_B(\mathbf{v}_B, \mathbf{v}_A)$ contains all the velocities that are the average of the current velocity $\mathbf{v}_A$ for agents A and a velocity inside the velocity obstacle $VO^A_B(\mathbf{v}_B)$ of agent B. Geometrically this can be interpreted as the velocity obstacle translated such that its apex (the top of the cone) lies at $\frac{\mathbf{v}_A+\mathbf{v}_B}{2}$.

The reciprocal obstacle avoidance method guarantees that the navigation is collision free, that the agents pass each other on the same side and that the navigation is oscillation free.

Assume that we have n agents $A_1, ..., A_n$ each agent A_i has its current position at $\mathbf{p}_i$, current velocity $\mathbf{v}_i$, a goal position $\mathbf{g}_i$ and a preferred speed v_i^{pref}. There is also obstacles moving at a constant velocity, lets call these $O \in \mathcal{O}$ with current position $\mathbf{p}_O$ and velocity $\mathbf{v}_O$.

With these in mind we create the union of all the neighboring agents RVO. For each agent A_i has the corresponding RVO^i. RVO^i is the *union* of all the RVO generated by the other agents. This can be describes mathematically as follows:

$$RVO^i = \bigcup_{j \neq i} RVO^i_j(\mathbf{v}_j, \mathbf{v}_i, \alpha^i_j) \cup \bigcup_{O \in \mathcal{O}} VO^i_O(\mathbf{v}_O) \tag{4}$$

If we chose a velocity outside of the RVO^i area, then the agent will not be part of a collision. The process of choosing a velocity starts with creating the preferred velocity vector $\mathbf{v}_i^{pref}$, which is the max speed v_max multiplied by the direction vector to the goal position. Then with the preferred velocity vector known we choose the closest vector $\mathbf{v}'_i$ which is outside of the RVO_i region.

Assume that we have an agent moving toward its goal using RVO. At one point, the agent has to move in a direction which differs much from its goal direction. In the presence of a third agent this new velocity can differ even more from the preferred velocity. This means that agents will not always pass each other on the same side and oscillations may arise (known as *reciprocal dances*). These reciprocal dances may take time to resolve which would cause deadlocks.

3.3.2 Hybrid Reciprocal Velocity Obstacle

The *hybrid reciprocal velocity obstacle* is a combination of the RVO and VO. Assume we have two agents A and B. When the velocity of A enters the RVO^A_B we check whether the end point of the velocity vector $\mathbf{v}_A$ of A lies on the right or left side in the area of RVO^A_B. Say that $\mathbf{v}_A$ lies on the right side of the middle of RVO^A_B, then we would like to choose a velocity on the right side of the RVO^A_B. To force the agent to choose a velocity on the right side of the RVO^A_B the opposite side of which the agent should pass on is enlarged by replacing the left edge of the RVO^A_B with the left edge of VO^A_B.

4 Evaluation

Evaluation can be done in many different ways. For example comparing our simulation with real footage from pedestrians walking, to see if they behave similarly. In case the final product is to be used in games and media, user studies should also be applied. Since the end users are those playing the game or watching the movie etc. For them it is important that the behavior of the virtual humans are natural looking.

In our case we started with comparing our simulation with real footage. To make it more understandable in this paper, we have introduced trails behind each agent. Which enables us to see where they have been and make it even more analyzable. The reference footage we used is a video[11], in which trails are used as well. A screen capture from the video can be found in Figure 5.

Figure 5: Footage from real people walking in groups

In Figure 6 the final result of our simulation using RVO2 and group formations. By just observing the trails it looks quite similar to the real footage above.

There is also a suite of tools for evaluating steering behaviors called SteerSuite. Which tests different scenarios and decides how well the steering behavior works. Unfortunately we didn't manage to get this up and running and would be something to look into in the future for later projects.

[11] http://www.mehdimoussaid.com/archives/58

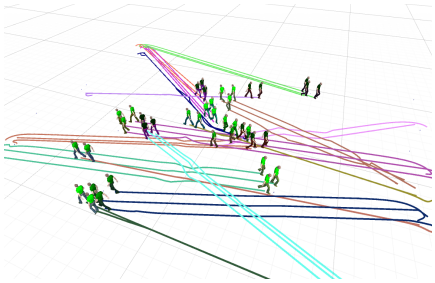

Figure 6: Screenshot from simulation

4.1 Performance

Our project is mostly applicable to games, since it is running in real time. When a user is playing a game he/she has to get feedback immediately for every action that they do, to keep the experience immersive. Therefore we decided to see how the real time performance is affected by the amount of agents in the simulation. Figure 7 shows *frames per second* while we ran the simulation with a increasing *number of agents*.

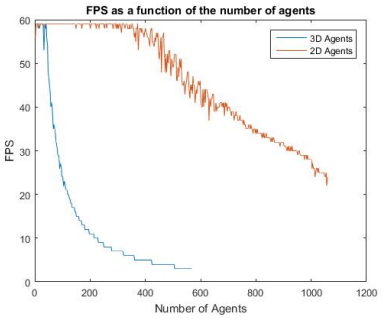

Figure 7: CPU: AMD Phenom II x4 810
GPU: Radeon HD 4650

For first person gaming experiences on computers, a steady frame-rate at 60 fps should be kept, to avoid ruining the experience. If a the fps is lower a decrease in player performance can be detected [4].

The results from our simulation is showing a steady 60 fps for up to 400 2D agents. For 3D agents it was significantly lower, only around 30 agents could be rendered at once to get the same results as with 2D agents. The explanation to this is that 3D agents require allot more processing power from the GPU, so the RVO2 algorithm was not our setback in that case. Therefore the 2D agents gives better results on how the algorithm performed.

We noticed some times that when a lot of agents interact with each other at once, the frame rate dropped slightly. Since each agent had to do even more calculations. This is probably the cause of the dips in Figure 7 for our 2D agents.

In our simulation we used a maximum of 10 agents for each agent to take in account when calculating a new velocity. A high number will decrease performance but give better results, a to low number will increase performance but give worse results. We tried a few and 10 seemed to give good enough results. If a human like behavior should be applied the number should be kept within what is possible for humans to pay attention to at once.

5 Conclusion

In this project we have been through a couple of different stages. In the beginning we started from zero and tried to implement our own type of steering behavior. We did manage to get the *seek, flee* and *arrival* behaviors to work with one agent. But we decided that it would take to long to reach our goal if we did the steering implementations our selves. Therefore we decided to use an existing package for steering behaviors called UnitySteer. Our impression of this package is that it is too limited for what we wanted to accomplish. It had some basic steering behaviors implemented, but they did not work as we wanted them to. Also, there was no API for the package so we decided to discard it.

The method we decided to make our final scenes with was the *Reciprocal Velocity Obstacle* method. This is a state-of-the-art method and is commonly used in video games. The results we got from using this method was outstanding. The agents never walked through each other, although there arose some deadlocks when the number of agents were really large.

5.1 Additional features to consider

Here we will mention a few features that could be considered for the future, to make this model even more human like. One of them is to include so called social forces[8], in which agents take different social norms in to account when deciding their velocities. Each group or agent could have a personal space radius etc, which other agents would avoid if possible. This would also make it possible to start the procedure of avoiding other agents even earlier, in our case it happened quite late and something like this would improve the realism. Another improvement could be to extend the RVO2 library with *acceleration velocity obstacles*, as for now it only uses velocities. By introducing acceleration it would be possible to avoid obstacles that are accelerating. Imagine a car that is just starting to accelerate. With the regular velocity obstacles we would assume that the car is no threat, since its current velocity is

minimal. But assume that the car is accelerating at a high acceleration. Then this prediction would be lousy because the car will increase its velocity each time step. Thus, there may be a possible collision between the car and the agent even though the agent does not see that at the moment. But with only humanoid agents in the scenario this approach will not make a big difference, since human rarely accelerate for long periods.

References

[1] F. Bevilacqua. Understanding steering behaviors: Collision avoidance.

[2] D. C. Brogan and N. L. Johnson. Realistic human walking paths. 2003.

[3] D. Cherry. Rvo collision avoidance in unity 3d.

[4] K. T. Claypool and M. Claypool. On frame rate and player performance in first person shooter games.

[5] P. Fiorini and Z. Shiller. Motion planing in dynamic environment using velocity obstacles.

[6] Q. Gu and Z. Deng. Generating freestyle group formations in agent-based crowd simulations.

[7] L. He, J. Pan, S. Narang, W. Wang, and D. Manocha. Dynamic group behaviors for interactive crowd simulation.

[8] D. Helbing and P. Molnár. Social force model for pedestrian dynamics.

[9] C. W. Reynolds. Steering behaviors for autonomous characters. 1999.

[10] J. Snape and D. Manocha. Navigating multiple simple-airplanes in 3d workspace.

[11] J. Snape, J. van den Berg, S. J. Guy, and D. Manocha. The hybrid reciprocal velocity obstacle.

[12] J. van den, J. Snape, S. J. Guy, and D. Manocha. Reciprocal collision avoidance with acceleration-velocity obstacles. *Robotics and Automation (ICRA), 2011 IEEE International Conference on.*

[13] J. van den Berg, S. J. Guy, M. Lin, and D. Manocha. Reciprocal n-body collision avoidance.

[14] J. van den Berg, M. Lin, and D. Manocha. Reciprocal velocity obstacles for real-time multi-agent navigation.

[15] S. Yi, H. Li, and X. Wang. Understanding pedestrian behaviors from stationary crowd groups. 2015.

[16] B. Zhou, X. Wang, and X. Tang. Understanding collective crowd behaviors: Learning a mixture model of dynamic pedestrian-agents. 2012.