RVO Collision Avoidance in Unity 3D

David Cherry

Abstract

Reciprocal Velocity Obstacles (RVO) method was used to produce collision avoidance for crowd simulations generated using the game engine Unity 3D. Smooth collision avoidance was produced for multiple agents in the same environment by using RVO collision avoidance method. There are three methods for generating crowd simulation using RVO collision avoidance in this experiment. The first method consists of taking user input on the starting position, goal position, and speed of each agent to be simulated and rendered in the virtual environment and the information will be exported to a file. The second method is loading a pre-existing file into Unity for the information to input for the simulation. The third method was to start with only one agent that continuously moves to random goal positions and can be duplicated to produce more agents in his environment. This experiment was combined with another collision avoidance method called predictive forces collision avoidance. An user interace was developed to choose which collision avoidance should be displayed and the specific method for the enviornment to be simulated and rendered in Unity.

1 Introduction

Crowd Simulation consists of producing movement of a large number of entities, or characters within a virtual environment. A few uses of crowd simulation consist of being used to create emergency evacuation simulations, video game crowd AI behavior, and recreating places in the past to show how people would have behaved. The characters used in the virtual environment are known as agents and each one is given their own AI to function in their respective environment. They are given a set path, destination, and specific behaviors based on their unique programming. One major issue that typically must be addressed when simulating multiple AI in the same environment is collision avoidance. Each agent must be able to detect objects or another agent, and then in turn be able to veer off the path in a smooth manner to prevent collision. If there is a case where two agents are both moving towards each other, then they must also behave in a way to avoid collision in an efficient manner.

This summer at the University of Minnesota, I focused on implementing the collision avoidance method called Reciprocal Velocity Obstacles (RVO) in the game engine Unity. Unity is a game engine built in with a powerful rendering engine designed with tools to create an interactive 3D environment [Creighton, 2010]. Unity has a drag and drop environment for creating games and can be programed in JavaScript, C, and Boo. The high level approach we took is outlined in Figure 1. Information on objects in Unity would be taken from the environemnt mean-

ing from User input or a data file with the information. The simulation based on the coding would begin and Unity would render the scene, objects and information so that the User would be able to view models, environments in a virtual environment.

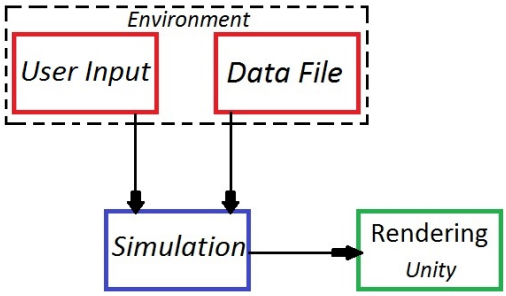

Figure 1: The flow chart or retrieving information from the environment(user input or from a data file), running simulation, and rendering in Unity

Unity 3D is growing in popularity amongst developers and can potentially be used in in virtual environments. Unity is compatible with software such as Mitsuba and blender to allow created models to be placed into the game engine environment. There is a project going on at the University of Minnesota to integrate Unity into a virtual environment to recreate ancient life in Greece. Theyll be able to produce structures, calculate the amount of people that could fit, and recreate people functioning in their environment. If these people, the agents in the virtual world, were walking around the city, then they would need to know how to avoid collision to continue on their path. This is where the RVO for multi-agents method could be applied. Unity has a variety of uses such as gaming, simulations, and even design. Unity has been used for robotic design by creating a functional virtual robot in Unity, then actually creating it based on the Unity model [Mattingly et al., 2012]. Other uses for Unity include application development that can port to iPhone, Android, and PCs.

2 Previous Work

2.1 Crowd Simulation for practical use

Crowd simulation techniques have been used to create emergency situation simulations. A mixture of social sciences and computer implementation of modeling using multi-agent systems with their own programming behavior was used to reproduce emergency situation behavior. One recurring issue is measuring how a group of people would react in a real emergency panic situation. To receive highly accurate results, it would require placing people in real emergency situations. However, making observations from pedestrian crowds could assist by showing the same patterns for reproducing the behavior. A person, that does not know the structure of the building as well, would focus on finding the nearest escape route, would follow the crowd and their velocity would rapidly increase [Almeida et al., 2013]. This scenario is illustrated in Figure 2.

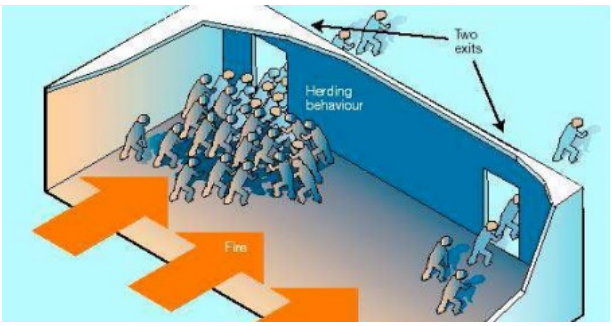

Figure 2: Crowd trying to escape from smoke-filled room [Almeida et al., 2013]

There have been specific tools created to generate environments with emergency evacuation environments. One tool called Collabmap has been created to generate environments, this could be combined with crowd simulation methods to create virtual scenarios, the effectiveness, and how long it would take to evacuate the building

[Ramchurn et al., 2013]. Some crowd simulation software is made for specific circumstances outside of buildings such as an evacuation "from an area under a terrorist bomb attack" [Shendarkar et al., 2006]. The creators of that experiment focused on their own BDI (belief, desire, intention) agent framework they've created.

There are three main reasons for developing computer simulations for crowd behaviors: first to test scientific theories and hypotheses; second, to test design strategies; third, to create phenomena about which to theorize [Almeida et al., 2013]. An understanding of crowd behavior is needed to replicate the behaviors of a real crowd for crowd simulation. Each agent also has various attributes to determine their behavior which consists of state, speed, vision, reaction time, collaboration, insistence, and knowledge [Almeida et al., 2013]. Vision is the aspect that depends on detecting obstacles and other agents. This is where collision avoidance would be applied due to the detection aspect.

The practice of crowd simulation based on cognitive, psychological, and sociological factors stem from the works of Craig W. Reynolds [Reynolds, 1987]. This paper focused on how he implemented behaviors of a flock of birds. His goal was to capture the behavioral aspects consisting of their movements to interactions. Steps are being made to expand upon his ideals on cognitive behaviors to try and govern what a character knows and how knowledge is acquired [Funge et al., 1999]. This can stem back to the knowledge aspect for emergency evacuations because if a character knows the fastest method to escape, then he would not follow the crowd in a panic similar to the situation displayed in Figure 2.

2.2 Methods for Real-Time Multi-Agent Navigation

Reciprocal Velocity Obstacles (RVO) is a concept for local reactive collision avoidance which implicitly assumes that the other agents make a similar collision-avoidance reasoning[van den Berg et al., 2008]. Each agent contains information on all of the other agent's current position, velocity and exact shape. Instead of giving each agent a new velocity to completely avoid collision with the other agents velocity obstacle, RVO introduces a method of taking the average of the current velocity and the velocity outside of the other agent's velocity obstacle. This will allow the agent to choose the velocity closest to its current velocity. Both agents will be able to take the closest velocity to their preferred velocity resulting in both agents each take an equal share of the work to successfully avoid collision. RVO is not only limited to being used in a virtual environment. Collision avoidance is also a fundamental problem in robotics [van den Berg et al., 2011].

The problem of having collision free motion and smooth movement for multiple robotic agents are still prevalent in the world today. Optimal reciprocal Collision Avoidance is just one example of a collision avoidance method that was used for the multiple robotic agent navigation [Snape et al., 2010]. This method and RVO both stem from Velocity Obstacles (VO) to avoid future collision [Fiorini and Shiller, 1998].

3 RVO in Unity

The goal for my project this summer was to implement a fully functional project in unity that could implement reciprocal velocity obstacles (RVO) collision avoidance to multiple agents. I started to do this by first creating an environment that consisted of six cubes and a flat terrain to begin working. In the

scripting, I coded the various formulas from the RVO paper to implement them into Unity [van den Berg et al., 2008].

First, I began learning Unity's mechanics and how to create and move objects. Instead of human models, I began with using cubes as game agents. After figuring out how to manage objects and scripts using the Unity Editor, I started to implement RVO formulas in code.

$$D = |v_{preferred} - v_{test}| \tag{1}$$

Equation 1 contains variable D which represents the magnitude of the preferred velocity minus an arbitrary test velocity. Velocities are sampled to find the best velocity to take for avoiding collision. I found that testing 100 samples of arbitrary velocities was enough to produce a nice motion. This value is added into Equation 2 for the RVO formula.

$$Score = ((w/(\tau + 0.001)) + D) \tag{2}$$

The score has a time returned to it. The smallest time for the score determines which test velocity should be taken.

Equation 2 contains the Score variable in my script. RVO finds the minimum time of collision and then finds the best velocity that is closest to its own to avoid collision. This score function will have a time returned to it and the smallest time returned to score will indicate that the test velocity used for calculating that score is the best velocity to take for collision avoidance. The τ represents the time to collision and contains the closest time to collision for each agent. The w variable is a specific number given to determine the importance on having the agent focus on collision avoidance or the goal position. The higher the w value, the more the object will focus on heading towards the goal. The lower the w value, the more the object will be geared towards collision avoidance. Given the above information, we can find how long the time is to a potential collision as follows. First, we must compute the values for Equations 3, 4, 5 as shown below.

$$a = (|v_{test} - v_{obj}|) * (|v_{test} - v_{obj}|) \tag{3}$$

$$b = 2 * ((|(v_{test} - v_{obj})|) \cdot (P_{cur} * P_{obj})) \tag{4}$$

$$c = |(P_{cur} - P_{obJ})|^2 - (r_{cur} * r_{obj}) \tag{5}$$

Equations 3, 4, 5 each contain information on the current object position (P_{cur}), the approaching object position (P_{obj}), the radius of both objects (r_{cur}, r_{obj}), and the arbitrary test velocity (v_{test}). Object Velocity (v_{obj}) in Equation 3 refers to the approaching objects current velocity while the arbitrary velocity (v_{test}) being tested for collision avoidance. Equations 4 and 5 have information about the current positions of both objects to give accurate information on the time to collision. These values will be placed into the quadratic formula for calculating the time to collision.

$$(t1, t2) = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{6}$$

Where $t1$ and $t2$ are the positive and negative solutions.

$$\tau = min(t1, t2) \tag{7}$$

The min takes in the smallest non-negative value.

TAs a result of the quadratic formula, two times may be produced to represent the time to collision. If both of these values are positive, then the lesser value will be the one stored to the minimum time to collision. If both of these values are not positive or both are negative, then there is a condition set to

not accept the time value because this implies that the objects would not collide. If one value is negative and the other is positive then that means it is currently colliding. The smallest time from the two positive values are stored into τ from Equation 7 to represent the closest time to collision. After coding fully functional RVO method into Unity, I ran the code with cubes show by Figure 3 below.

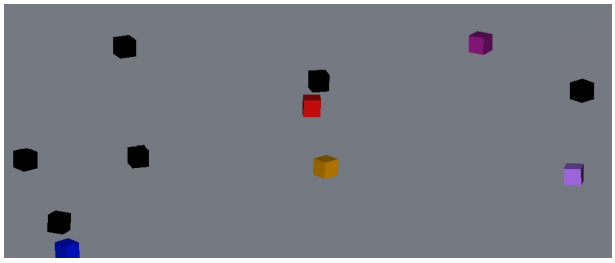

Figure 3: Cubes with random goal positions using RVO collision avoidance

The cubes were given a randomized position and after they reached a small distance away from their goal position, they were given another goal position to navigate to. When two cubes were about to collide, using RVO collision avoidance, they were able to detect each other and avoid collision to continue towards their goal position. The next task assigned was to be able to specify the cube's starting position, ending position, and speed as input then to run that in the game environment while still having RVO applied. I created a GUI interface allowed users to create their own text files with this format displayed in Figure 4.

The user interface that was created in Unity exports a file with this format and takes that information to create the specified objects, then lead them towards their goal position. The next step was to move onto actual animated models instead of using cubes as my agents. The human models were imported inside of unity to replace the cubes. One issue was adjusting the models to face their current position and animating the

day.txt* ×

```
1  Number of Objects, 2
2  Object
3  Starting Position,200,5,250
4  Ending Position,200,5,250
5  Speed,7
6
7  Object
8  Starting Position,400,5,250
9  Ending Position,200,5, 450
10 Speed,7
11
12 Object
13 Starting Position,600,5,230
14 Ending Position,200,5,250
15 Speed,7
16
17 Object
18 Starting Position,315,5,150
19 Ending Position,20,5, 450
20 Speed,7
21
22 Object
23 Starting Position,200,5,250
24 Ending Position,200,5,250
25 Speed,7
26
27 Object
28 Starting Position,400,5,250
29 Ending Position,200,5, 450
30 Speed,7
```

Figure 4: The format for files to be implemented in Unity

models to walk, but this was solved by using a rotation method implemented in Unity. After both issues were resolved, I finally have RVO collision avoidance with human agents. This is shown in Figure 5 below.

The next task was to create a button the user interface that will allow a previously created file to be read and another button that will allow the original random goal position program to run. Implementing the read a file method was similar towards reading the created file which lead to no difficulties. The random goal position assigned to each objects was already created and also integrated into the program. This scene begins with one individual and to duplicate that individual, I have the user press 'T'. The starting scene is

Figure 5: Overhead view of RVO collision avoidance

shown in Figure 6 and the scene with duplicated instances is shown in Figure 7.

Figure 6: Starting with one agent with randomized goal

Figure 7: Duplicate of agents with randomized goals using RVO collision avoidance

The final task was to combine my project with Jassiem Ifill, another DREU student from my lab, who was implementing another crowd simulation technique, predictive force collision avoidance [Karamouzas et al., 2009]. We had to integrate both of our projects into one project that was supported by the same user interface. This gives the user options of observing the RVO collision avoidance method or the predictive force collision avoidance method. After combining our projects, we created a fully functional project with collision avoidance in the same type of environment using human animated agent.

4 Conclusion

The complete project is able to run both reciprocal velocity obstacles collision avoidance and predictive forces collision avoidance. There are differences in how the code operates in both projects; however, both are able to create and run a file with the user specifications for the human agents while being able to avoid collision. They also are able to read a previously made text file as long as it has the correct format for being read. In the RVO method, there is an option to duplicate from the original human agent to add more human agents all with arbitrary starting positions, and goal positions in the same environment with RVO collision avoidance.

One limitation for Unity is the amount of agents that can move and be displayed on the screen while running smoothly. Unity tends to slow down a great deal when to many agents are placed in the same environemnt. For future experiments RVO collision avoidance could be used in a larger custom environment such as a city. Many different agent models and stationary obstacles could be put into place to further test RVO collision avoidance. Another feature to create after generating this environment is a first person user controlled character. This will allow a closer view on how smooth the RVO collision avoidance s while the other agents avoid collision with the user's controlled agent.

5 Acknowledgements

I'd like to thank Dr. Stephen Guy for his guidance, assistance, and methods for RVO,

Unity and the project as a whole. I also would like ot thank Dr. Ioannis Karamouzas for his help with Unity and crowd simulations.

References

[Almeida et al., 2013] Almeida, J. E., Rosseti, R. J., and Coelho, A. L. (2013). Crowd simulation modeling applied to emergency and evacuation simulations using multi-agent systems. *arXiv preprint arXiv:1303.4692*.

[Creighton, 2010] Creighton, R. H. (2010). *Unity 3D Game Development by Example: Beginner's Guide*. Packt Publishing.

[Fiorini and Shiller, 1998] Fiorini, P. and Shiller, Z. (1998). Motion planning in dynamic environments using velocity obstacles. *The International Journal of Robotics Research*, 17(7):760–772.

[Funge et al., 1999] Funge, J., Tu, X., and Terzopoulos, D. (1999). Cognitive modeling: knowledge, reasoning and planning for intelligent characters. In *26th annual conference on Computer graphics and interactive techniques*, pages 29–38.

[Karamouzas et al., 2009] Karamouzas, I., Heil, P., van Beek, P., and Overmars, M. H. (2009). A predictive collision avoidance model for pedestrian simulation. In *Motion in Games*, volume 5884 of *Lecture Notes in Computer Science*, pages 41–52. Springer.

[Mattingly et al., 2012] Mattingly, W. A., Chang, D.-j., Paris, R., Smith, N., Blevins, J., and Ouyang, M. (2012). Robot design using unity for computer games and robotic simulations. In *Computer Games (CGAMES), 2012 17th International Conference on*, pages 56–59. IEEE.

[Ramchurn et al., 2013] Ramchurn, S. D., Huynh, T. D., Venanzi, M., and Shi, B. (2013). Collabmap: crowdsourcing maps for emergency planning.

[Reynolds, 1987] Reynolds, C. W. (1987). Flocks, herds, and schools: A distributed behavioral model. *Computer Graphics*, 21(4):24–34.

[Shendarkar et al., 2006] Shendarkar, A., Vasudevan, K., Lee, S., and Son, Y.-J. (2006). Crowd simulation for emergency response using bdi agent based on virtual reality. In *Simulation Conference, 2006. WSC 06. Proceedings of the Winter*, pages 545–553. IEEE.

[Snape et al., 2010] Snape, J., Guy, S. J., van den Berg, J., and Manocha, D. (2010). Smooth coordination and navigation for multiple differential-drive robots. In *Proc. IEEE RSJ Int. Conf. Intell. Robot. Syst*, pages 1–13.

[van den Berg et al., 2011] van den Berg, J., Guy, S. J., Lin, M., and Manocha, D. (2011). Reciprocal n-body collision avoidance. In *Cédric Pradalier, Roland Siegwart, and Gerhard Hirzinger (eds.), Robotics Research: The 14th International Symposium ISRR*, volume 70 of *Springer Tracts in Advanced Robotics*, pages 3–19. Springer-Verlag.

[van den Berg et al., 2008] van den Berg, J., Lin, M. C., and Manocha, D. (2008). Reciprocal velocity obstacles for real-time multi-agent navigaiton. In *IEEE International Conference on Robotics and Automation (ICRA)*.