An Evaluation of Path Smoothing Algorithms for Pathfinding in Games

Ryan Tuck
Computer Science
Rensselaer Polytechnic Institute
Troy, United States
ryan.d.tuck@gmail.com

Mei Si
Cognitive Science Department
Rensselaer Polytechnic Institute
Troy, United States
sim@rpi.edu

***Abstract*—One of the most common tasks in video games is creating a path for AI agents to follow. Creating smooth, short paths in a time efficient manner is of great importance to video game developers. The most classical and still widely used algorithm is the A* algorithm. Various modifications to A* have been proposed. In this paper, we evaluate five algorithms that are aimed at improving A* through path smoothing using standard pathfinding benchmarks provided by Sturtevant. Our results indicate that Reactive Path Following (RPF), an algorithm proposed by Valve and used in their video game Left 4 Dead, outperforms other algorithms for generated close to optimal path length and can be executed in running time comparable to A* and the other four variations of A*.**

Keywords—pathfinding, reactive path following, path smoothing

I. INTRODUCTION

The video game industry is one of the largest industries in the entertainment business. In 2009, 273 million units were sold worldwide, bringing in $10.5 billion in revenue. 67% of US households play video games [6]. The industry is always looking for new and better technology and algorithms to deliver a better game experience for the player.

In many modern video games, pathfinding is one of the most important problems. Game developers want agents that make use of the shortest possible, natural-looking paths. At the same time, however, constraints on both memory and CPU usage mean that any algorithm has to use limited computer resources. This problem is exacerbated by the fact that path-creation needs to be done in real-time and, since the target the AI agents is following is usually also moving, often needs to be re-pathed. Therefore, having a time efficient pathfinding algorithm is critical. Poor pathfinding can also make AI agents in the game seem too artificial or, even worse, idiotic [12] but this is not within the realm of the discussion in this paper. In this paper, we focus on finding close to optimal paths (in terms of path length) in a reasonable time.

Various algorithms have been proposed for pathfinding. The most classical and still widely used one is the A* algorithm [9]. Various modifications to A* have been proposed. They can be roughly divided into two types. The first type aims to improve path length through path smoothing, including A* Post-Smoothed (A* PS) [15], Theta* [10], Reactive Path Following (RPF) [3], etc. The other type of algorithms for improving A* emphase reducing memory usage (including IDA*[9], TBA* [4], etc).

In this paper, we focus on path smoothing algorithms. While multiple algorithms have been proposed, they have not been formally evaluated using a common benchmark. In this work, we perform an evaluation on the performance of four path smoothing algorithms plus * using Nathan Sturtevant's Pathfinding Benchmarks. We also record their running times. Our goal is to determine which algorithms are able to find close to optimal paths while not spending extensive time.

In particular, we are looking at algorithms which were proposed in recent years and have received considerable successes. The path smoothing algorithms we include in this evaluation are A* PS, Theta*, RPF, and Ramer-Douglas-Peucker (RDP) [5]. RDP is not traditionally associated with pathfinding in games and is usually used in computer graphics, such as level-of-detail and digitization of lines. This algorithm can reduce the size of an existing path by smoothing it. By adding a simple modification, which we will expand in Section II C, RDP is able to serve as a path smoothing algorithm. We believe it will provide a good point of comparison. Similar to RDP, A* PS is also a post-processing algorithm that smooths an existing path. Theta* improves the computation of cost during path creation in A*. Finally, RPF is an online algorithm that smooths the path as the agent follows it.

We implemented all five algorithems then performed the evaluation. The results from the evaluation indicate that Reactive Path Following is the best algorithm for path smoothing. It is able to create paths significantly shorter than A*, A* PS, RDP, and Theta* while running in a comparable amount of time as A*, A* PS, and RDP.

In the next section, we will describe the algorithms we evaluated in this paper in detail and provide the pseudo code we used for implementing them in the evaluation. Then we will provide the details of our evaluation and statistical tests we performed for analyzing the results, followed by discussion and future work.

II. Descriptions of the Algorithms

In this section, we describe A* and the four variations of A* based based pathfinding algorithms used in this paper. They improve A* either by directly modifying the algorithm (Theta*) or by adding a post-processing step on the path generated by A* (A* PS, RDP, and RPF).

Pseudocode is provided for each algorithm. They are taken from the developers of the algorithms. The only modification we have made is to use consistent symbols and terminology across all algorithms through this paper. We implemented the algorithms following the pseudo codes.

A. A*

A* is the basis for all modern pathfinding algorithm. It was created by Hart, Nilsson, and Raphael (1968) as a heuristic solution to finding minimum cost paths on a graph [7]. A* and its variants are the most popular pathfinding algorithms used in video games [13].

Algorithm 1 denotes the pseudo codes for A*. A* works on a graph of vertices (or nodes). With each vertex it maintains two values: the length of the shortest cost path from the start vertex to the current vertex (**Algorithm 1**, Line 3) and a heuristic cost function which provides the estimated distance from the current vertex to the goal vertex (**Algorithm 1**, Line 4). There are different implementations of A*. In this paper, we adapted a version that also stores the parent of each vertex (**Algorithm 1**, Line 5) as this allows path retrieval and is used as a basis for Theta*, which will be introduced in Section II D.

Throughout the pathfinding process, A* maintains two lists (**Algorithm 1**, Line 6). The first is a list (or priority queue) of open vertices that A* has visited but has not expanded. The second is a list of vertices that A* has closed by expanded. At the very beginning, the open list contains the starting vertex and the closed list is empty. As the algorithm runs, these two lists are updated (**Algorithm 1**, Lines 9 & 14). A vertex with the lowest path length plus heuristic cost is removed from the open list (**Algorithm 1**, Line 9) and a vertex is added to the open list if it is connected to the current vertex, has a lower path length, and is not in the closed list (**Algorithm 1**, Lines 15 to 17 and **Algorithm 2**, Line 6). A vertex is added to the closed list if when it has been removed from the open list (**Algorithm 1**, Line 14) and no vertices are removed from the closed list. The algorithm ends when the open list is emptied or the end vertex is reached (**Algorithm 1**, Lines 10 & 12).

Algorithm 1 A* (s_{start}, s_{goal})

```
1:  # open: a list/priority queue of open vertices
2:  # closed: a list of closed vertices
3:  # g(s): the length of current path from the
      start
4:  # h(s): the heuristic length of the from current
      location to the end
5:  # parent(s): the parent of vertex s
6:  open ← closed ← {};
7:  g(s_start) ← 0;
8:  parent(s_start) ← s_start;
9:  open.Insert(s_start,g(s_start) + h(s_start));
10: while open != {} do
11:     s ← open.Pop();
12:     if s = s_goal then
13:             return "Path found.";
14:     closed ← closed.Append(s);
15:     foreach s' in neighbor(s) do
16:             if s' not in closed then
17:                     if s' not in open then
18:                             g(s') ← ∞;
19:                             parent(s') ← NULL;
20:                     # Expands the vertex
21:                     UpdateVertex(s,s',open);
22: return "No path found.";
```

Algorithm 2 UpdateVertex(*s*,*s'*,*open*)

```
1:  # g(s): the length of current path from the
      start
2:  # h(s): the heuristic length of the from current
      location to the end
3:  g_old ← g(s');
4:  # Updates the g* cost
5:  ComputeCost(s,s');
6:  if g(s') < g_old then
7:      if s' in open then
8:              open.Remove(s');
9:      open.Insert(s',g(s') + h(s'));
10: end
```

Algorithm 3 ComputeCost(*s*,*s'*)

```
1:  # g(s): the length of current path from the
      start
2:  # c(s,s'): the straight line cost from s to s'
3:  # parent(s): the parent of vertex s
4:  if g(s) + c(s,s') < g(s') then
5:      parent(s') ← s;
6:      g(s') ← g(s) + c(s,s');
7:  end
```

B. A* with Post-Smoothed Paths (A* PS)

A* with post-smoothed paths (A* PS) is a technique to create smoother paths from A* paths [15]. It was initially created by Thorpe (1984). A* PS works by removing unnecessary vertices from an A*-generated path, which enables this algorithm to make an improvement over A*. For each vertex in the path, A* PS determines whether line-of-sight exists to non-immediate successor vertices (**Algorithm 4,** Lines 3 & 4). If such line-of-sight exists, then it removes any intermediary vertices as they are regarded as unnecessary. This algorithm was improved by Botea, Muller, and Schaeffer [1]. Our implementation in this paper follows the algorithms provided by [1].

Algorithm 4 lists the procedures for A* PS. For every vertex in the path, denoted by $[s_0,...,s_n]$, A* PS checks whether the vertex can be removed. It does so by checking whether its two adjacent vertices can be linked directly (**Algorithm 4**, Lines 4, 7, & 8). For example, assume A* PS first checks whether line-of-sight exists from s_0 to s_2 - whether there is a direct unblocked line connects the two vertices. If it does, then s_1 is removed from the path. The algorithm then proceeeds and checks whether line-of-sight exists from s_0 to s_3. If it does, s_2 is removed. This continues until s_n is reached or a line-of-sight check is failed, at which point the process repeats on the next vertex. The line-of-sight check is implemented from [11].

Algorithm 4 A* PS($[s_0,...,s_n]$)

```
1: k ← 0;
2: t_k ← s_0;
3: foreach i ← 1...n-1 do
3:      # LineOfSight checks whether an unblocked
              line exists between two points
```

```
4:      if not LineOfSight(t_k, s_{i+1}) then
5:              k ← k + 1;
6:              t_k ← s_i;
7: k ← k + 1;
8: t_k ← s_n;
10: return [t_0,...,t_k];
```

C. Ramer-Douglas-Peucker (RDP)

The Ramer-Douglas-Peucker (RDP) algorithm, also known as the split-and-merge algorithm, simplifies a path by recursively removing points and merging line segments [5]. It was created independently by Ramer (1972) and Douglas and Peucker (1973). RDP is not traditionally associated with pathfinding and is usually used in computer graphics. However, by adding a line-of-sight check it serves as a path smoothing algorithm.

RDP works by taking a list of vertices like A* PS does. It draws a line between the first and last point and checks the distance of every point in the list against this line (**Algorithm 5**, Lines 9 to 12). If the point furthest from the line (the outlier) is at a distance smaller than a given threshold ε, all points between the first and last are removed. If the outlier is at a distance greater than the threshold, the path is split into two smaller segments (**Algorithm 5**, Lines 14 & 15). The first segment includes all points between and including the first point and the outlier while the second segment includes all points between the outlier and the remaining points. The algorithm is then recursively called on each line segment (**Algorithm 5**, Lines 16 & 17). The implementation of this algorithm was modified to include a line-of-sight check in addition to checking the ε. Before removal of points, there must also be line-of-sight. The algorithm ends when the size of the list is less than 3 or there are no outliers (**Algorithm 5,** Lines 4, 14, & 22).

Algorithm 5 RDP([s_0,...,s_n], ε)

```
1: # [s_0,...,s_n]: a list of vertices 0 to n
2: first ← s_0
3: last ← s_n
4: if n < 3 then
5:      return [s_0,...,s_n]
6: k ← -1;
7: dist ← 0;
8: foreach i in [0,...,n] do
9:      # findPerpendicularDistance finds the
              perpendicular distance from s_i to the
              line segment first to last
10:     cDist ← findPerpendicularDistance(s_i, first,
              last);
11:     if(cDist>dist){
12:             dist ← cDist;
13:             k ← i;
14: if dist > ε then
15:     l_1 ← [s_0,...,s_{k+1}]
16:     l_2 ← [s_k,...,s_n]
17:     r_1 ← RDP(l_1, ε)
18:     r_2 ← RDP(l_2, ε)
19:     r_s ← r_1 union r_2
20:     return r_s
21: else
22:     return [first, last]
```

D. Theta*

Theta* was created by Daniel, Nash, Koenig, and Felner (2007) to be a pathfinding algorithm capable of finding any-angle path on a grid [10], which means a path that can go through any open space in the map. In contrast, A* only allows a path which goes through the edges of the graph. The basic version of Theta* (which is the one implemented here) will find a path from the start to goal vertex if one exists and will only return unblocked paths [8].

Theta* improves upon A* by improving the ComputeCost function, which is defined in **Algorithm 3** for A* and in **Algorithm 6** for Theta*. In this modification, two paths are considered: the path considered by A* (**Algorithm 6**, Lines 6 to 8) and the straight line path from the start vertex to the parent of vertex s added to the straight line path from the parent of vertex s to vertex s' (**Algorithm 6**, Lines 2 to 4). The second path is guaranteed to be no longer than the first path (due to triangle inequality) but only the first path is guaranteed to be unblocked. In short, if line-of-sight exists from the parent of s to s' then the second path is used. If not, the first path - the one usually used by A* - is used.

Algorithm 6 ComputeCost(s,s')

```
1: if LineOfSight(parent(s),s') then
2:      if g(parent(s)) + c(parent(s),s') < g(s')
              then
3:              parent(s') ← parent(s);
4:              g(s') ← g(parent(s)) +
                        c(parent(s),s');
5: else
6:      if g(s) + c(s,s') < g(s') then
7:              parent(s') ← s;
8:              g(s') ← g(s) + c(s,s');
```

E. Reactive Path Following (RPF)

Reactive Path Following (RPF) was created by Valve (2008) for their game Left 4 Dead [3]. Left 4 Dead is a multi-player focused FPS that has both co-operative and competitive aspects. In the game, the player was faced with defeating and running away from huge hordes of zombie AI agents. This presented an interesting problem. Valve needed a robust algorithm that was able to create smooth paths for their zombie "flocks" to follow. Due to the amount of agents, they also needed the algorithm to be relatively quick to calculate. To these ends, Valve created Reactive Path Following.

RPF modifies how an agent follows an already-constructed path. Any algorithm can be used to construct the initial path, but both this paper and Valve used A* to generate the initial path. Once this path is generated, the agent checks down the path a certain distance - the look-ahead distance (denoted α) (**Algorithm 7**, Lines 3 & 4). The agent then checks whether line-of-sight exists to the look-ahead point (**Algorithm 7**, Line 5). If it doesn't, it decreases the look-ahead distance and re-checks whether line-of-sight exists. This process is repeated until a point is found that has line-of-sight.

Since RPF uses existing algorithms to create the path and doesn't attempt to smooth the path, path-creation and re-pathing costs are as cheap as any existing algorithm. However, RPF does require additional runtime during the game-loop to perform line-of-sight checks. The amount of checks required is heavily dependent on the initial look-ahead distance and how far this distance is decreased. In essence, the cost of smoothing

the path is deferred. In real-world applications such as video games where re-pathing is common this can be advantageous.

Algorithm 7 RPF$(p,[s_1,\ldots,s_n],\alpha)$

```
1: # p: current position of the agent
2: foreach s' in [s_n,...,s_0] do
3:      dist ← Distance(p,s')
4:      if dist < α then
5:              if LineOfSight(p,s') then
6:                      return s'
7: return "Error"
```

III. Evaluation

In order to use a common point of reference, this paper uses Nathan Sturtevant's Pathfinding Benchmarks to evaluate the performance of the various algorithms [14]. Sturtevant provides both maps from commercial games and the length of the optimal paths between various vertices on the maps. These maps provide a variety of real-world pathfinding benchmarks.

Using maps from this library, the game world is represented by a grid of points. It is assumed that the agent is the size of a single point which can only move in cardinal directions (which have a length of 1) or ordinal directions (which have a length of $\sqrt{2}$) and cannot cut corners.

Sometimes, a pathfinding algorithm may generate a path that needs to be changed to follow the grid representation of the world. Bresenham's line algorithm is used to translate points generated by the various algorithms to conform to these constraints. In Fig. 1, we can see an example of Bresenham's line algorithm translating a continous line into a discrete number of points on a grid. In this example, doing this changes the path length from 5.83 to 7.

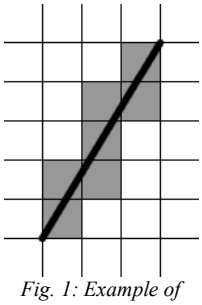

Fig. 1: Example of Bresenham's line algorithm

Three maps from Sturtevant's benchmarkers were used (AR0204SR, AR0600SR, and AR0700SR). We used the versions of the mpas that is to 512x512 in size. These maps are shown as Fig. 2.1, Fig. 2.2, and Fig. 2.3. We chose these maps because they provide a good mix of open spaces and complicated smaller spaces, and thus allow us to evaluate the performances of the path smoothing algorithms in different contexts.

For each of the maps, a navigation mesh was created for the game world using axis-aligned quads [2, 16]. Sturtevant's scenario files were used. These scenario files provide different start and end points on the map along with the optimal path length, though not the path itself. Map AR0204SR has 1263 pairs of start and end points. Map AR0600SR and AR0700SR have 1281. We used all of them in this evaluation.

The algorithms and testing environment were implemented in Python 3.2.3 and Pygame 1.9. Python and its associated libraries are excellent for rapid prototyping and implementation but are relatively slow. We will be more interested in the ratio between timings rather than the exact timing of each algorithm. The program was run on an AMD Phenom(tm) II X4 965 Processor (3.40 GHz) with 4 GB of RAM. The OS was Windows 7 Professional 64-bit.

Fig. 2.1: AR0204SR

Fig. 2.2: AR0600SR

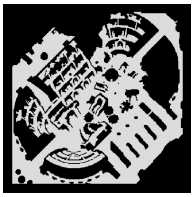

Fig. 2.3: AR0700SR

Fig. 2: Maps for Evaluation

IV. Results

We evaluated the four variations of A* based on two factors: distance of the path and time required to generate the path. In order to provide comparable data between each test, ratios were used. Distance ratios are the ratio of the algorithm path length and optimal path length. The closer the value is to 1, the more optimal the path is. Time ratios are based on the ratio between the algorithm time and the time for A* to create a path. We used A* for providing benchmarks because there is no existing benchmarks for running time.

Fig. 3, 5, and 7 provide box plots of the distance ratios for the three maps respectively. Fig. 4, 6, and 8 show the box plots of time ratios. It can be seen from these plots that RPF provides paths that are both shorter and at comparable times to the other algorithms, that is it has a lower mean ratio (averaged from the 1000+ tests performed on each of the maps) and smaller standard deviation.

We have statistically tested for differences. Table I, II, and III show the mean difference, the standard deviation difference, and the p-value for each algorithm's distance and time ratios compared to RPF. Two-tailed paired Student's t-test was used for calculating the p-value. We conducted these tests for each map separately.

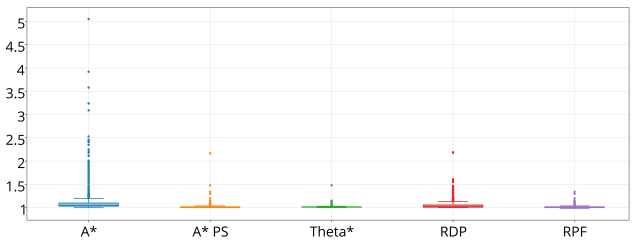

Fig. 3: Distance Ratio on Map AR0204SR

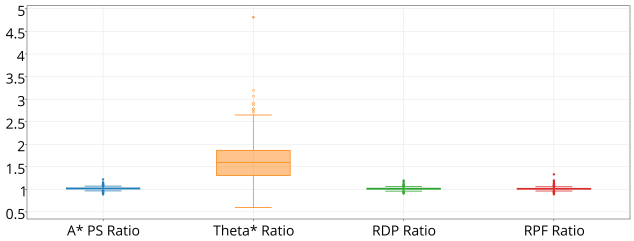

Fig. 4: Time Ratio on Map AR0204SR

TABLE I. RATIO COMPARISON OF RPF ON AR0204SR

Algorithm	Distance Ratio		
	Mean diff.	***Std. dev. diff.***	***p-value***
A*	0.105	0.262	< 0.01
A* PS	0.004	0.038	< 0.01
Theta*	0.002	0.023	< 0.01
RDP	0.038	0.070	< 0.01
	Time Ratio		
A* PS	0.008	0.032	< 0.01
Theta*	0.002	0.028	< 0.01
RDP	0.587	0.383	< 0.01

For all three maps, the distance of the path was significantly less when using RPF as compared to A*, A* PS, and RDP ($p < 0.01$). For AR0204SR and AR0600SR, the reported distance of the path was significantly less when using RPF as compared to Theta*. For each map, there was a significant difference between time required to compute the path between RPF and A*, A* PS, RDP, Theta*.

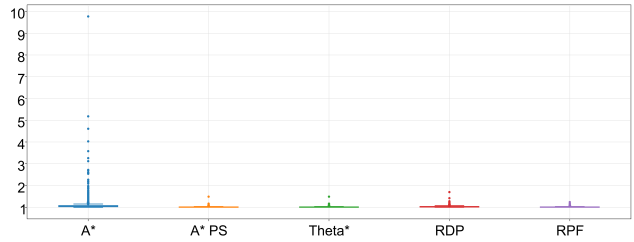

Fig. 5: Distance Ratio on Map AR0600SR

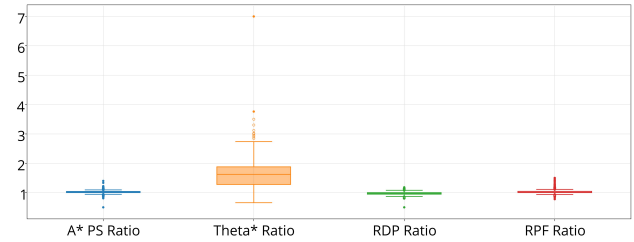

Fig. 6: Time Ratio on Map AR0600SR

TABLE III. RATIO COMPARISON OF RPF ON AR0600SR

Algorithm	Distance Ratio		
	Mean diff.	***Std. dev. diff.***	***p-value***
A*	0.100	0.362	< 0.01
A* PS	0.004	0.019	< 0.01
Theta*	0.002	0.018	< 0.01
RDP	0.022	0.038	< 0.01
	Time Ratio		
A* PS	-0.0081	0.0776	< 0.01
Theta*	0.5863	0.4231	< 0.01
RDP	-0.0512	0.0774	< 0.01

V. DISCUSSION AND FUTURE WORK

Overall, RPF performed the best. RPF was able to consistently return results in comparable time to other algorithms and closer to optimal path length. Theta* was able to provide path lengths that were also close to optimal but took considerably longer to run. A* PS and RDP were able to provide path lengths that were shorter than A* alone but still significantly longer than RPF and Theta*. They were, however, able to do it significantly quicker than RPF. However, the actual runtime of all algorithms is fast. To provide reference, we averaged the running time of each of the algorithms across all maps, which are as follows: A* - 143ms, A* PS - 145 ms, Theta* - 197ms, RDP - 144ms, and RPF - 145ms.

For our tests, we used known benchmarks to evaluate our results. While this provided a common point of comparison, it also put certain constraints on the testing environment. Movement was constrained to cardinal and ordinal directions. It would be interesting to see if less constrained movement resulted in better results.

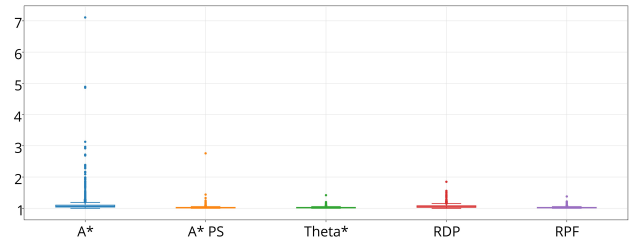

Fig. 7: Distance Ratio on Map AR0700SR

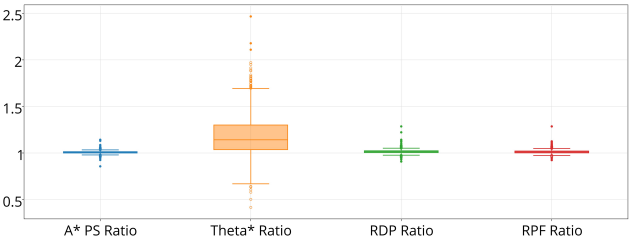

Fig. 8: Time Ratio on Map AR0700SR

TABLE IV. RATIO COMPARISON OF RPF ON AR0700SR

Algorithm	Distance Ratio		
	Mean diff.	*Std. dev. diff.*	*p-value*
A*	0.101	0.300	< 0.01
A* PS	0.004	0.019	< 0.01
Theta*	-0.002	0.021	< 0.01
RDP	0.039	0.058	< 0.01
	Time Ratio		
A* PS	-0.007	0.020	< 0.01
Theta*	0.169	0.208	< 0.01
RDP	0.002	0.023	< 0.01

One of the strengths of post-processing algorithms, such as Reactive Path Following, is that they are mostly independent of the path-creation algorithm. This paper focused only on algorithms that reduced path length however A* still faces problems with memory usage. We are interested in combing these two approaches by running Reactive Path Following on A* variants that aim to reduce memory usage, such as IDA* or TBA*. This would hopefully result in a resulting algorithm that generates short paths in reasonable time and for reasonable memory usage.

Currently, one of the weaknesses of Reactive Path Following is the number of calls to check line-of-sight. In our 2D, grid-based environment, line-of-sight checks are very quick to compute but this may not hold true in more complicated, 3D environments. Further developments could be made to reduce the number of line-of-sight checks needed by Reactive Path Following.

Finally, for our tests, the look-ahead distance used by Reactive Path Following was constant for each map and each scenario file. It is possible that changing the look-ahead distance will affect the performance of Reactive Path Following. This suggests a further area of study in whether and how much changing the look-ahead distance changes the results and whether any gains can be made by dynamically altering the look-ahead distance.

VI. CONCLUSION

In this paper, we implemented five algorithms that are applicable to pathfinding in video games (A*, A* PS, Theta*, RDP, and RPF). We examined how close to optimal their generated paths were and how quickly they were able to compute their paths. We used maps and benchmarks from Sturtevant. Overall, Reactive Path Following is preferred. It returned results that were closer to optimal than A*, A* PS, Theta*, and RDP. Reactive Path Following was also able to return results in a comparable amount of time to A*, A* PS, and RDP.

REFERENCES

[1] A. Botea, M. Muller, & J. Schaeffer, “Near optimal hierarchical path-finding,” in Game Development, 1(1), 2004, pp. 1–22.

[2] M. Booth, “The official Counter-Strike bot,” Turtle Rock Studios, in Game Developer's Conference, 2004.

[3] M. Booth, “The AI systems of Left 4 Dead,” Valve Software, in Artificial Intelligence and Interactive Digital Entertainment Conference at Stanford, 2009.

[4] V. Bulitko, Y. Bjornsson, N. Sturtevant, and R. Lawrence, “Real-time heuristic search for pathfinding in video games” in Artificial Intelligence for Computer Games, 2010.

[5] D. David and T. Peucker, “Algorithms for the reduction of the number of points required to represent a digitized line or its caricature,” Cartographica: The International Journal for Geographic Information and Geovisualization, 10.2, 1973, pp. 112-122.

[6] Entertainment Software Rating Board, “Video game industry statistics,” ESRB, 2013. <http://www.esrb.org/about/video-game-industry-statistics.jsp>.

[7] P.E. Hart, N.J. Nilsson, and B. Raphael, “A formal basis for the heuristic determination of minimum cost paths,” in Systems Science and Cybernetics, 4.2, 1968, pp. 100 - 107.

[8] D. Kenny, A. Nash, & S. Koenig, "Theta*: any-angle path planning on grids," in Journal of Artificial Intelligence Research, 39, 2010, pp. 533-5 79.

[9] I. Millington and J. Funge, Artificial Intelligence for Games. CRC Press, 2009.

[10] A. Nash, K. Daniel, S. Koenig., & A. Felner, “Theta*: Any-angle path planning on grids” in Proceedings of the AAAI Conference on Artificial Intelligence, 2007, pp. 1177–1183.

[11] F. Permadi, “Ray-casting tutorial for game development and other purposes”, 1996. <http://www.permadi.com/tutorial/raycast/index.html>

[12] D. Pottinger & J. Laird, “Game AI: the state of the industrial, part two”, 2000. <http://www.gamasutra.com/features/20001108/laird_01.htm>

[13] B. Stout, “Smart moves: intelligent pathfinding” in Game Developer Magazine, October/November, 1996.

[14] N. Sturtevant, “Benchmarks for grid-based pathfinding” in Transactions on Computational Intelligence and AI in Games, 4, 2012, pp. 144 - 148.

[15] C. Thorpe, “Path relaxation: path planning for a mobile robot.” In Proceedings of the AAAI Conference on Artificial Intelligence, 1984, pp. 318–321.

[16] A. Yahja, A. Stentz, S. Singh, and B. Brummit. “Framed-quadtree path planning for mobile robots operating in sparse environments.” in Proceedings, IEEE Conference on Robotics and Automation, (ICRA), 1998.