

Path-finding with Dijkstra's Algorithm

1 Abstract

As we develop and advance in the Botball competition, the need arises to build more and more intelligent systems. Path-finding isn't only about reducing the repetitive work on programmers, but also allowing robots to make decisions on the fly. As an alternative to techniques such as line-following, or as an aid, path-finding can take advantage of the largely static Botball board layout. This paper seeks to describe a technique and example implementation by which a perfectly optimal, non-intersecting, easily navigable path can be found. In addition, this document outlines surrounding support systems for working with such paths, and putting things into practice.

2 Reasoning

2.1 An Example Problem

Imagine you have a movement library, such as CBCJVM's [5], that allows for simple geometric movements and position tracking¹

[13]. The robot successfully finds a set of blocks with it's camera, but now you have to figure out how to navigate back to your drop-off point. There's a few simple answers, such as reversing the movements that you took to get the blocks returning you to a known point, but all of these are inefficient. If you try to simply move from one point to another, using position tracking, you have no easy way of handling if there is an object in the way of your path.

2.2 How Pathfinding can Help

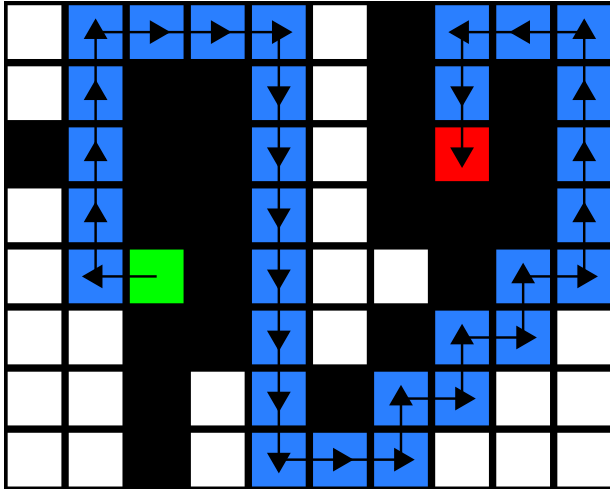
A proper path-finding system already knows the layout of obstacles on the board, either through static (programmer) input, or through dynamically (sensor) inputted information. With this, the system should be able to determine the best way to get from Point A to Point B, avoiding everything else in-between [7]. This can then be distilled into a simple set of actions, such as

1. Rotate 37 degrees clockwise
2. Move 36 cm forward

¹With the right calculations, as long as all your motions happen through a single movement library, you should be able to dynamically determine exactly where you are at any point in time.

3. Rotate 84 degrees counter-clockwise
4. Move 15 cm forward

3 A* (A-Star)



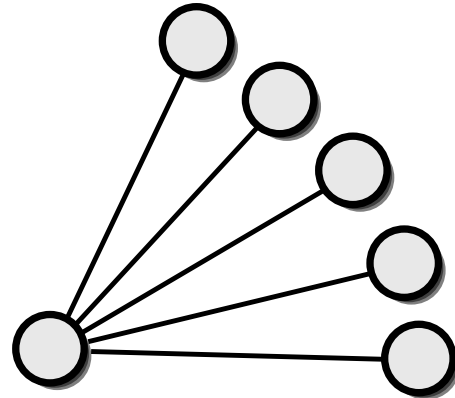
One way of solving such a problem is to turn the board into a grid, and to utilize the simple A* algorithm to then generate a path on the grid. Such a path is then traced and turned into a set of movements, like above. Unfortunately a specific description of the algorithm is beyond the scope of this document, but there is a plethora of places, online and otherwise, to find such a description. [3]

However, such a system works in a non-optimal manner. Our robots do not operate on a grid, and, as a result, the generated paths can end up including far more points than ideal. Additionally, the A* algorithm does not guarantee that the resultant path, even in a grid setting, is the shortest one.

While this is an interesting system to work with, and an even more interesting proof of concept, the more geometrically aligned Dijkstra's algorithm appears a better choice.

²The robot is treated as infinitely small: as though it has no radius. This is not accurate, but it makes working with things easier, and as is pointed out later in this paper, we can make up for this with some polygon expansion.

4 Graph Theory



Let's start by turning the board into the simplest representation we can: a set of line segments. Each of these line segments have two end-points, or nodes, and a group of lines, sharing nodes at vertices form polygons. The board can be represented through a set of inaccessible polygon areas, and otherwise open area.

When you are forming an optimal path, assuming the robot itself is a point-mass², the only nodes included in the path will be the starting point, the ending point, and the vertices of the polygons on the map. This assumption greatly reduces our search-space.

4.1 Line-of-Sight

As obvious as it seems to us that the robot cannot simply pass through pvc-pipe (excluding robots with treads or other advance apparatuses), we must define such in our algorithm. The obvious way of doing such is to declare that from one node, the only other directly accessible nodes are those within the line-of-sight of the current node.

Looking around, there don't seem to be many good published line-of-sight algorithms for our purposes. Most, often for graphical setups, deal with pixel-level, or grid-based systems. [6]

As a result, I attempted, with some success, a naïve technique, derived from ray-tracing systems.

4.1.1 Ray-Tracing

Ray-tracing is a common technique for high quality 3d rendering, and such algorithms lay at the center of multi-million dollar programs, such as Pixar's Renderman software, used in countless Disney-Pixar flicks, as well as a wide variety of other Hollywood blockbusters. [8]

The algorithm is a near brute-force approach at mimicking mother-nature. We can pretend that the sun, instead of sending out continuous particle-waves, sends out countless numbers of rays, which follow rules of refraction. Some of these end up in our view, others don't. The density of returned "rays" per unit area (per pixel) determines the resultant brightness. If rays don't get to a certain area, we end up with shadows (See where we're going?). [11]

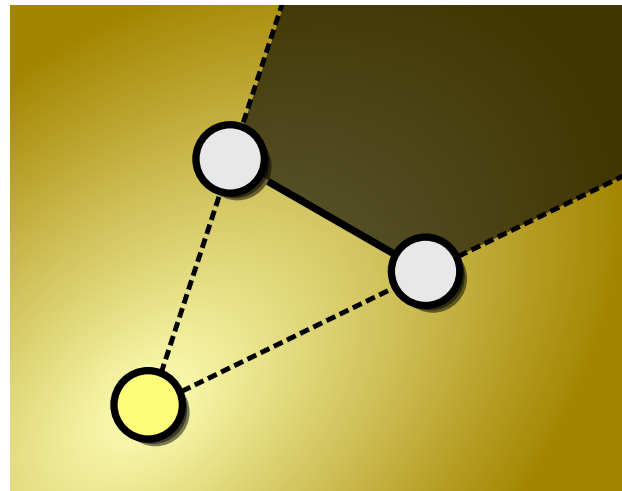
4.1.2 Reverse Ray-Tracing

One might say this implementation is extremely inefficient, and it is. Many of the rays get lost in the abyss of the computer's RAM. Perhaps they enter the world of Tron, and get taken into the games, to be tortured, only to be killed by the evil garbage collector (After all, we're using a high-level managed

language like Java, right? If not, imagine it's the dark lord `dealloc`.)

As a result, there's an alternative strategy, although not always applicable, or as ideal. If, for example, the light-source is located at the same point as the view-point, we can have each of our objects send emit rays back towards the view-point, instead of the other way around, hence the algorithm's name. Along the way, they can collide with other objects and such, but not near as many rays are wasted, and such an algorithm is now far from brute-force. [12] Such an algorithm is typically $O(n)$, where n is the number of objects on the field³.

4.1.3 Lighting up the Board



Imagine the robot is acting as a lamp on the board. If it sends off rays of light, only the ones intersecting with walls, or lines, are stopped, and all other rays continue traveling infinitely.

³This is known as Big-O notation. It relates the execution time of an algorithm to the size of the dataset provided. It is purely about proportionality. $O(n)$ means that the execution time is directly proportional to the data size, n . Big-O notation is useful because it makes the bold statement that performance only matters as a system of scale.

4.1.4 Reverse Ray-Tracing and Line of Sight

Let's have each node on the board draw a line segment to the current node on the board. From there, we can check to see if this line segment drawn intersects any polygons on the board. If it does, then we can conclude that the node is not directly visible or accessible.

4.1.5 Optimizations

Performance wasn't an important goal for our system, but if you need more speed from your system, these a few notes can be of use when optimizing.

- Visibility is mutual, meaning that if one point is visible from the other, the opposite must be true. [12]
- If a board is statically laid out, visibility never changes, and so such results can be cached.

4.2 Dijkstra's Algorithm

Dijkstra's⁴ Algorithm is an extremely popular algorithm, excellent for deducing perfectly optimal paths in a variety of situations, and for its design, it has been proven mathematically optimal in terms of performance, executing in $O(n^2)$ time, where n is the number of vertices, or nodes. It has been used for both the obvious, such as in navigation systems, to the not-so-obvious, such as phone-call routing.

Dijkstra's Algorithm works to reduce the "cost" of travel. For simplicity sake, we only count the cost of translational movement, which parallels the distance the robot travels⁵. It's quite possible, and trivial, to modify the algorithm to factor in the "cost" of rotational movements and the like. However, for our example, whenever we say cost, read "the distance to travel".

4.2.1 Pseudo-code [4]

1. Make a dictionary⁶ relating each node to the current shortest path to that point. For each node, let the beginning undefined path equal infinity⁷, with the exception of the starting node, which you should set the cost of navigation to as zero (Ending where you start takes no time, and thus "costs" nothing).
2. Make a queue⁸ of all the nodes on the graph, excluding the starting one.
3. Let the current node be the starting node.
4. For each node in the queue, calculate the cost of the path through the current node to that node. If this new cost is less than the previous least-cost path to that node, replace it in that dictionary you made.
5. Select the lowest cost path (where the node is in the queue). This is an optimal path. If this path is to the end-node, you can stop. If all the nodes

⁴The name is a Dutch one. Interestingly, it is pronounced "dike-struh" [10]

⁵A sidenote: as Dijkstra's algorithm works in a purely relational manner, the units used don't matter.

⁶A dictionary, or a hash-map or hash-table is a data structure that relates one piece of data to another. Thinking of it in terms of an actual dictionary, you relate the key, the word, with the value, the definition.

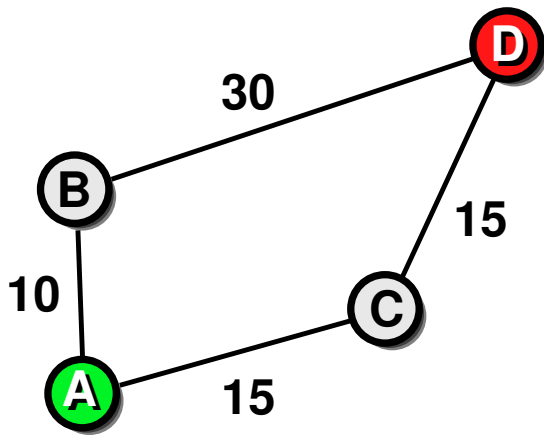
⁷If this isn't possible or easy in your environment, assign it some special constant, such as `null` or Python's `None`, and note that special value for later on, as I did in my Python reference implementation. Work your later numerical checks around this.

⁸For our purposes, this can simply be an array, list, or set of some sort.

have infinite cost, and you have not generated a path to the end-node yet, there is no possible path.

- Remove the final node from this least-cost path from the queue, and make it the new current node. Jump back to step 4.

4.2.2 An Example



Following the above diagram and described algorithm, we can trace the execution of such. Data structures will be shown in a Pythonic manner. Specifically, the dictionary will be expressed in the form `{key=value}` [2], and the list/queue will be expressed as `[a, b, c]` [1].

- Our dictionary starts as `{a=0, b= ∞ , c= ∞ , d= ∞ }`.
- Our queue starts as `[b, c, d]`.
- Our current node starts as `a`.
- The only directly accessible nodes from node A are B and C, each with a cost of 10 and 15 units respectively. Our dictionary becomes `{a=0, b=10, c=15, d= ∞ }`

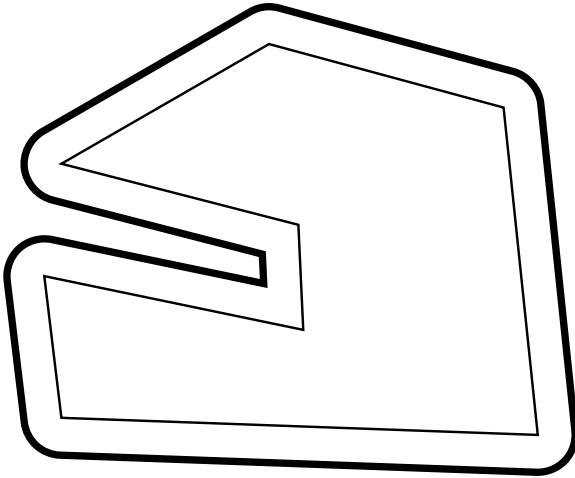
- Our least-cost path from the queue is through node B.
- Our queue becomes `[c, d]` as our current node becomes node B.
- The only directly accessible node from node B that hasn't already been visited (is still in our queue) is node D. The cost to get to node D through node B is 40 units. 40 units is less than ∞ , so our dictionary becomes `{a=0, b=10, c=15, d=40}`.
- Our least-cost path from the queue is through node C.
- Our queue becomes `[d]` as our current node becomes node C.
- The only directly accessible node from node C that hasn't already been visited is node D. The cost to get to node D through node C is 30 units. 30 units is less than 40 units, so our dictionary becomes `{a=0, b=10, c=15, d=30}`.
- We now have our optimal path to node D, our end-node, in the form of `a → c → d`.

5 Existing Problems

5.1 Working With a Radius

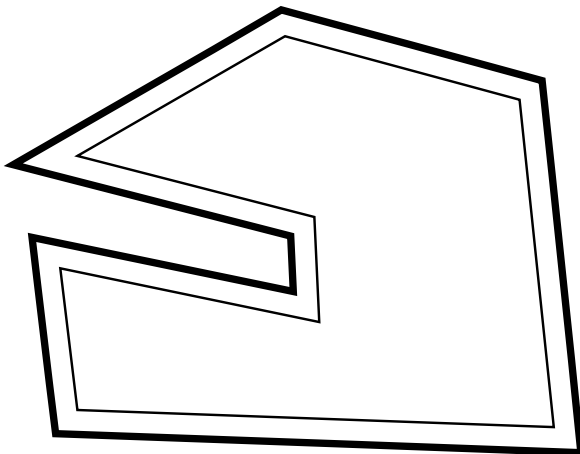
Up this point, we've treated our robot as in infinitely small entity. Unfortunately for us, the vast majority of robots have a non-zero volume. A simple way of doing this is to outset each polygon that composes the board a distance equal to the radius of the robot, and to work with that outset version. There are several general ways of doing this.

5.1.1 Constant Distance



The most accurate way to outset a polygon is to draw a new shape around the polygon, ensuring a constant distance. Unfortunately, the behaviors of this system can be somewhat complex, as seen in the example diagram. Outward-facing vertices form arcs, and so the end result is not a polygon. If one wanted to simulate this system, they could form approximations of these curved areas by circumscribing polygons around them.

5.1.2 Shifting the Edges



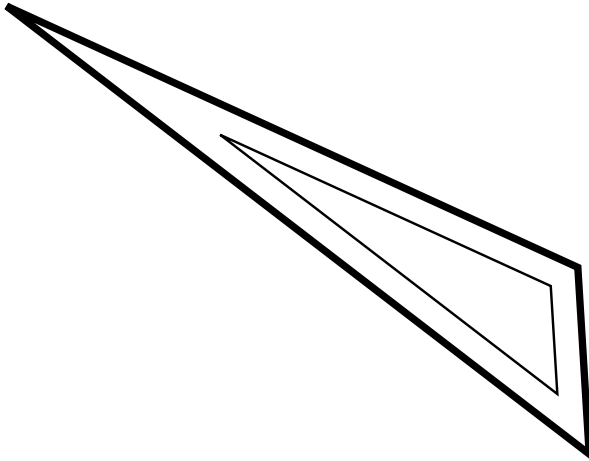
A much simpler approach, is to simply outset the edges, forming new lines, and then to connect them. The naïve algorithm used in our sample implementation to expand each polygon is as follows:

1. For each line segment, find it the slope of a line perpendicular to it (find the negative inverse, $-\frac{\Delta x}{\Delta y}$).
2. Find a point on the original line segment.
3. Using the point and perpendicular slope, derive a line segment extending from a (any) point on the original line segment outward⁹, with a length equal to the desired outset distance.
4. Take the second point of the newly formed line-segment. Form an infinitely extending line that passes through that second point with the slope of the original line segment (a parallel line).
5. Once you have these parallel lines for each edge of the base polygon, find the intersection points¹⁰ for the lines corresponding to consecutive edges on the base polygon. The new polygon can be formed from these intersection points.

A few special handlers should be added to deal with horizontal and vertical lines. Many languages include support for -0 and $\pm\text{inf}$ in their floating-point implementations, and these may be of use.

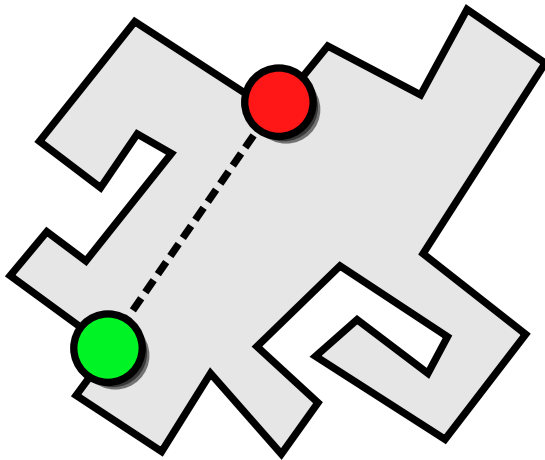
⁹You can easily determine what way is outward if the polygon's points are in a known order, such as clockwise or counter-clockwise

¹⁰To do this, think back to algebra, where you set one equation equal to the other and solved



The disadvantage of such an approach can be seen when one is working with very acute angles. Fortunately for us, most board layouts consist of little more than right angles, and with right and obtuse angles, such a disadvantageous effect is minimized.

5.2 Edge Cases



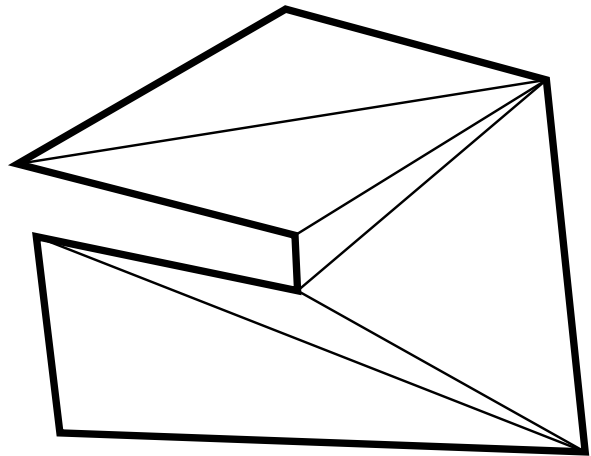
Our line-of-sight algorithm works for the majority of cases. However, when first conceived, an “edge”-case was ignored. If the point of perspective and the viewed point are both points on the same polygon, it is possible that a our line-of-sight algorithm would

fail to catch that the points are not visible.

Simply saying that if two nodes are on the same polygon they are not visible from one another would be an inaccurate statement, so a more intricate solution to the conundrum had to be found.

The system that we ended up using was a rather blunt one.

5.2.1 Polygon Triangulation



Polygon triangulation refers to a common practice in graphics programming by which one takes an arbitrary polygon, convex¹¹ or concave¹², and turns it into a set of triangles. This is done simply because triangles tend to be easier to work with, as far more assumptions can be made about them. [9]

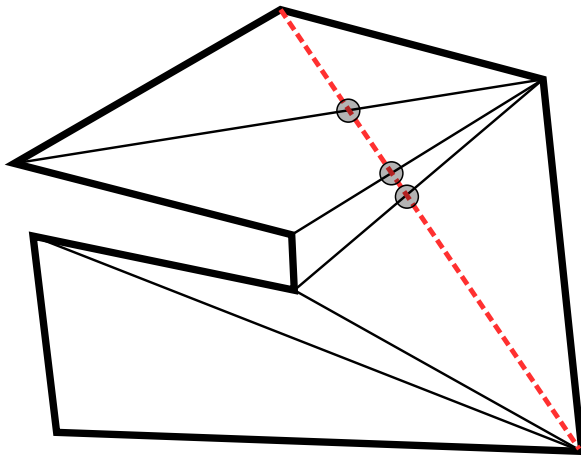
There are a variety of ways of performing such a task, some easy, some complex. Rather than implementing our own system, we managed to procure a piece of Java code to do the task, and we ported the system over to Python.

The Java code was donated under public domain to the project by **asarkar** on the **#xkcd-cs** irc channel on **irc.foonetic.net**. (Thanks, **asarkar**!)

¹¹A polygon that only has outward facing vertices, an example being any *regular* polygon.

¹²A polygon that “caves” in upon itself.

5.2.2 Applying Triangulation to Line-of-Sight



On top of our prior checks, our system does the following for each polygon on the map:

1. Triangulate the polygon.
2. Perform additional intersection checks with the lines formed by triangulation. If there is an intersection, the nodes are not visible.
3. Check to see if our line is the same as one of the lines formed by the triangulation. If so, the nodes are not visible from one another.

6 Conclusion

6.1 Reference Implementation

Our reference implementation is written in Python, or more specifically, in an implicitly statically typed subset of the language,

¹³The Python 2.x and 3.x series are not entirely compatible however they share many similarities, such that if one is careful, they can support both with the same code-base. Python 3.x is meant to replace the 2.x series, but both are currently supported by the Python Software Foundation, and both are in common use.

known as RPython. The code has been specially crafted to run under Jython 2.5 and up, CPython 2.5 and up, and CPython 3.1 and up¹³. Other versions and implementations should work, but simply have not been tested.

The implicitly static nature of the design, and the use but lack of dependence of on Object-Oriented Programming should make porting to other languages such as Java and C trivial. The end implementation, despite it's complexity, is short (less than 1000 lines of code).

The code is available on Github under the GNU GPLv3 at <https://github.com/CBCJVM/python-pathfinding>.

6.2 Closing Thoughts

The entire system was designed in a modular form. If nothing else, this paper demonstrates the powers of encapsulation and subdivision. While each algorithm is not perfect performance-wise, they work well enough for the desired purpose, and more importantly, a superior algorithm could easily be dropped in as a replacement, thanks to the modular design.

As a more personal message, opening this technology to the community should hopefully provide small struggling teams with a powerful tool, and make next year's robots all the more interesting.

Keep Hacking!

7 Works Cited

- [1] 3. *An Informal Introduction to Python — Python v2.7.2 documentation*. URL: <http://docs.python.org/tutorial/introduction.html#lists>.
- [2] 5. *Data Structures — Python v2.7.2 documentation*. URL: <http://docs.python.org/tutorial/datastructure.html#dictionaries>.
- [3] *A* search algorithm — Wikipedia, the free encyclopedia*. URL: https://secure.wikimedia.org/wikipedia/en/wiki/A*_search_algorithm#Process.
- [4] *Dijkstra's algorithm — Wikipedia, the free encyclopedia*. URL: https://secure.wikimedia.org/wikipedia/en/wiki/Dijkstra%27s_algorithm.
- [5] Braden McDorman, Benjamin Woodruff, et al. “CBCJVM: Applications of the Java Virtual Machine with Robotics.” In: *KIPR Global Conference for Educational Robotics* (2010). URL: <https://github.com/CBCJVM/GCER-CBCJVMPaper-2010>.
- [6] Andy McFadden. *Line of Sight Algorithm for Tile Based Games*. Sept. 15, 1999. URL: http://www.gamedev.net/page/resources/_/reference/programming/isometric-and-tile-based-games/298/line-of-sight-algorithm-for-tile-based-games-r729.
- [7] *Pathfinding — Wikipedia, the free encyclopedia*. URL: <https://secure.wikimedia.org/wikipedia/en/wiki/Pathfinding>.
- [8] *Pixar's RenderMan© | quotes*. URL: https://renderman.pixar.com/products/whats_renderman/features.html.
- [9] *Polygon triangulation — Wikipedia, the free encyclopedia*. URL: https://secure.wikimedia.org/wikipedia/en/wiki/Polygon_triangulation.
- [10] *Pronunciation: Dijkstra — MathOverflow*. URL: <http://mathoverflow.net/questions/4381/pronunciation-dijkstra>.
- [11] Paul Rademacher. *Ray Tracing: Graphics for the Masses*. URL: <http://www.cs.unc.edu/~rademach/xroads-RT/RTarticle.html>.
- [12] *Ray tracing (graphics) — Wikipedia, the free encyclopedia*. URL: https://secure.wikimedia.org/wikipedia/en/wiki/Ray_tracing_%28graphics%29#Reversed_direction_of_traversal_of_scene_by_the_rays.
- [13] Benjamin Woodruff. *How CBCJVM Position Tracking Works — Github*. 2011. URL: <https://github.com/CBCJVM/CBCJVM/wiki/Position-Tracking>.

This paper can be found online along with its editing history and its L^AT_EX and SVG sources at <https://github.com/CBCJVM/GCER-Pathfinding-Paper-2011> (See the downloads button for a PDF version). This paper has been made available under the terms of the GNU Free Documentation License. Licensing information is available at the above link.