Nick deWaide - 46744619
Dustin Shaffer - 72716857
Andrew Martin Del Campo - 49527954

Mac-Pan: Non-Deterministic Pac-Man

Introduction: Arcade Pac-Man

Our project is loosely based on Pac-Man. Briefly, our project's goal was to simulate a non-deterministic version of Pac-Man complete with ghost AIs, Pac-Man AI, and an AI for "tile shuffling" which will be explained later.

Pac-Man was an arcade game in which the player must eat all of the dots on the screen while also avoiding four ghosts. The ghosts have unique personalities and will behave in different ways. As the game progresses, the ghosts are released from their "ghost house" and will chase the player in a manner that reflects their predefined personality. For example, the pink ghost, Pinky, tries to predict where Pac-Man will be in the future in order to ambush him. The game's playing field is a tile-based, maze-like board. The board remains static throughout the game and there are also several "energizers" on the board which are big dots that allow Pac-Man to eat the ghosts. Eating an energizer also causes the ghosts to flee from Pac-Man.

Ghost behavior in arcade Pac-Man was entirely deterministic. This means that the ghosts have well-defined patterns of behavior that they will follow. For example, level 1 has four phases. In the first phase, ghosts "scatter" for 7 seconds and then "chase" for 20 seconds. Scattering means that each of the four ghosts starts moving towards their respective corner of the board. The second phase of level 1 is the same as the first phase. The third phase is scattering for 5 seconds, chase for 20 seconds. The final phase is scattering for 5 seconds, then permanently chasing until either Pac-Man has lost all of his lives or he has cleared the board of all dots. It is important to keep in mind that the ghosts' behaviors during both the scatter and chase phases are deterministic. If you are able to simulate the same set of game world circumstances multiple times, the ghosts will behave identically in each scenario. Level 2 also has four phases, but the numbers are slightly tweaked in favor of the ghosts and the ghosts' speeds are increased.

At intersections in the maze, ghosts must make a decision on which direction they should go. In the event that a ghost is equally attracted to more than one direction, it will pick the direction in this order: UP, LEFT, DOWN. The ghost will never break a tie by going right. The fact that there is a tie means at least one of the other options will have an equal or higher precedence than going right.

The takeaway is that Pac-Man was entirely deterministic. Creating an AI that plays the original arcade Pac-Man would be as simple as taking advantage of the quirky subtleties of the

ghosts' behaviors. It is not an interesting problem and can be solved with minimal effort.

Our Problem: Non-Determinism

In order to make the problem of creating AIs relating to Pac-Man *significantly* harder and more interesting, we introduced features of non-determinism to the game world.

> **Tile Shuffling -** This exactly what it sounds like. The board's tiles are literally shuffled around during the game. A stretch of dots can be swapped with a stretch of walls. Whatever is on the tiles being swapped will be swapped. This means that the tiles occupying ghosts can be swapped with other tiles, which effectively becomes the notion of "teleporting ghosts". Originally, Pac-Man's AIs relied on the static nature of the board to make their decisions. Instead of now only being concerned with making a decision at an intersection, any AI must ideally make a decision on every update cycle or as close to every update cycle as possible. Tile shuffling alone transforms our problem into one which is far more computationally-exhaustive.
>
> **Randomized Ghost AIs -** Implementing the original Pac-Man's ghost AIs in a tile-shuffling environment is impossible. As was previously mentioned, the ghosts relied on intersections and fleeing to the board's corners. With the introduction of tile-shuffling, there no longer exists a permanent notion of the board's corners or intersections. We could have put more effort into emulating the original ghost behavior, but then they would have been largely if not wholly deterministic. With that unwanted degree of determinism being added back in, our problem would become easier. Much like the original Pac-Man, our implementation of the ghosts also includes a "scatter" and a "chase" mode and the ghosts have unique speeds. Unlike the original Pac-Man, the ghosts do not have a predefined set of phases. They will randomly and independently of one another decide when to scatter and when to chase. The locations to which they scatter are randomly chosen and the length of time for which they will chase you is randomly chosen.

Why?

Originally, we wanted to create an AI for solving tile-based puzzle games based on a rule set as input. Professor Flerova said that the idea was too general and would be difficult to gauge the effectiveness of the AI. She then suggested Pac-Man as an idea. After discussing the idea with the members of group 13, Professor Flerova proposed the idea of starting with Pac-Man and adding things to it to make the related computations more difficult. We liked the the idea, so we went with it.

We also looked around at other projects centered around Pac-Man, both commercial and academic in nature. What we found was that Ms. Pac-Man incorporated randomness into the ghost AIs, but not quite to the degree we did. Every student project we looked at was

centered around a classic deterministic version of Pac-Man, so we wanted to do something different.

Tile shuffling combined with our randomized ghost AIs make the game of Pac-Man unpredictable and the problem of creating associated AIs vastly more difficult. This jump in difficulty was also a factor in what attracted us to the problem.

What?

We were hoping that we could simulate a non-deterministic Pac-Man game world with AIs that make decisions and respond in real-time. As was previously mentioned, there are not any past Pac-Man based projects that match the problem we created. That being said, we were also really hoping to provide a new, creative hybrid approach of techniques.

Related Work

Our exact problem of Pac-Man with extremely chaotic conditions has not been tackled, so we must discuss similar works.

First, we looked at a student project **[1]** who used AI and machine learning methods for solving the original deterministic version of Pac-Man. The project had three major categories of implementations: simple graph/tree searching, multiagent searching like minimax trees with alpha-beta pruning, and reinforcement learning. This influenced our initial direction with our project in that we first chose to implement basic search algorithms which will be discussed at greater length later.

However, our problem is much different from the deterministic version of Pac-Man. Minimaxing with alpha beta pruning was an unrealistic approach for our problem. The whole point of minimaxing with alpha beta pruning is when you generate a tree once and then prune branches to make searching the tree faster. With the non-determinism we want our AIs to make decisions as fast and as close to real time as possible. Again, we will discuss our exact algorithms later and demonstrate why our creative hybrid approach was needed.

The machine learning methods are great for deterministic Pac-Man, but did not really appeal to us due to the fact that only one of us has taken CS 178.

We also investigated this AI course website from Berkeley **[2]**. The layout of this problem is also a deterministic version of Pac-Man. The breadth of concepts applied to the Berkeley class project is wider than the previously mentioned student project, but they are largely the same. Classification methods, such as the naive Baye's model and perceptrons, were used that would interpret the state of the game board and output a directional decision for Pac-Man. This would have been an interesting approach to apply to our non-deterministic world, but we lacked the machine learning fluency to implement such methods ourselves.

The final related work worth mentioning is this student project **[3]**, also centered around deterministic Pac-Man. This student came up with 8 conditions that could be true or false and made decisions at intersections accordingly. It's a machine learning approach resembling a basic classifier, but what was interesting was the idea of the conditions having continuous weights attached to them. For example, one condition is that "Pac-Man is facing a ghost and the ghost is heading towards Pac-Man," and the weight of the condition is dependent on how far away the ghost is from Pac-Man. This general idea of changing weights is partially what inspired our first creative approach, which can be likened to the generation of a board of "magnetic" or "gravitational" values.

Overall, we did not find a student Pac-Man project with much of any degree of non-determinism. We did, however, borrow combinations of approaches from the solutions to the deterministic version of the game.

Algorithms: Non-AI Involvement

We programmed the engine in Java and the AI scripts and other game features in Lua. The Lua scripts were interpreted and executed on the JVM using LuaJ. LuaJ was the only external tool we used and its sole purpose was to allow the Lua scripts we wrote to be executed by the JVM.

We programmed all of our data structures, system input, displayed UI and rendering, level construction, specific types of bodies which exist in each level (such as the player, enemies, and pellets), delegate function dispatchers, timers, game logic, and feature reloading in Lua.

Algorithms: Basic Search

BFS - The first algorithm we looked at implementing was the breadth-first search. Our tile-based board can be treated as a graph where the tiles are nodes and every pair of adjacent tiles has an edge between it with a weight of 1. A graph with these properties can be breadth-first searched to gather all single-source shortest paths, which is why we chose to use this algorithm.

Our implementation of the breadth-first search is very standard for this kind of problem. We have sets for visited tiles and tiles ready to have their neighbors expanded and then a dictionary mapping each tile to its predecessor. During the search process the dictionary is updating, so for example the tiles adjacent to the starting tile will be mapped to the starting tile in predecessors. Upon mapping the goal node to its first found predecessor, the search will stop and then backwards traverse the dictionary starting at the goal until you're back at the destination. From this information, our algorithm then outputs the best direction to move.

A* - The next algorithm we implemented was the A* search. We implemented this algorithm because it uses a heuristic to perform much better than the breadth-first search. The shell of this algorithm is very similar to breadth-first searching. A* still examines all current adjacent nodes before making a decision, but it will only truly expand the tile with the lowest f score. For every tile, T, there is an assigned f score. The f score is equivalent to the distance from T to the starting node plus the estimated distance from T to the goal node. For the estimated distance to the goal node, we used the Manhattan distance heuristic. The Manhattan distance between a start and goal is absoluteValue(start.row - goal.row) + absoluteValue(start.col - row.col). The Manhattan distance gives us a best-case distance between a start and goal on a grid. The tile-based board for Pac-Man can very much be treated like a grid, so this heuristic was quite useful.

A* also used the sets of visited and ready tiles as well as the predecessors dictionary. When the search is done, you have to loop through the predecessors starting at the goal until you hit the starting tile just like with BFS.

Gravity Map - Our next algorithm is a fusion of concepts that we took from basic searching, multi-agent searching, and even a little bit of learning. Gravity map is an approach that will map gravity values onto a board the same size as the game board. It combines the idea of there being multiple spheres of influence, like in multi-agent searching, but there is also technically no true “path” searching taking place. It’s a kind of searching combined with very simple machine learning elements in that there are calculations which take place and from those calculations a tile is predicted as being the best one to choose.

As a very simple example of the algorithm, think about Pac-Man in a corridor with a dot at the end. To generate the gravity map for Pac-Man, we breadth-first search on every item of interest, which can include dots, energizers, and ghosts. The gravity board itself is the same size as the playing field and is initialized with all zeros. The item of interest, perhaps a dot, has an initial weight that is added to the gravity board, but on every step of the breadth-first search we apply a degeneracy function to that value. The tile with the dot on it might have a value of 1, but the tile next to it might have 0.5 and going further right might decrease it to 0.25, then 0.1250, etc. Basically the further you are from a dot, the smaller its influence on Pac-Man will be.

If you have 300 dots and 5 ghosts on the board, the generation of Pac-Man’s gravity board will have to breadth-first search 305 times. This might seem computationally exhaustive, but since we are doing constant time degeneracy calculations and additions at each step as opposed to dealing with a dictionary of predecessors and finding the exact path, it ends up being a fairly viable way of doing things. Additionally, the ghost AIs can be handled as a group with a single gravity board. Ghosts are only concerned with Pac-Man, so they only need to BFS one time in order to generate their board with this approach.

Upon the generation of the board, the interested party, Pac-Man or a ghost, will look at the tiles currently adjacent to them and pick the direction of the one with the most attractive gravity value. With the gravity approach, we tried to cut out the slow specifics of searching for paths with a general fit solution. Now instead of having 5 ghosts independently searching and pathing to Pac-Man, we handle all of their potential moves with a single computation.

Theoretical Gravity Fields - The Gravity Map approach was not as scalable as we wished for it to be, so we attempted to create a more optimal solution. We based the optimization on the notion that you could represent a gravity field as a system of equations - which in our case is a set of degeneracy-over-distance functions with different distance/weight inputs.

We again needed a BFS approach to obtain a true distance between bodies in our field. Because this is the most computationally expensive process and the most commonly occurring process, we decided it would be a good idea to parallelize these computations in a set of futures (a future is a function which promises a result at a future time) which would run in individual threads. Later, at query time, we could present our problem - our Pac-Man - and then find the best fit value out of all of the degeneracy functions that were determined to be relevant for our Pacman, using the calculated distances and explicit weights.

Unfortunately, the distance processing did not have any ability to eliminate unnecessary body pair distance calculations (such as a pellet to another pellet) because it came before the query. This caused the algorithm to make (n+1) * (n/2) distance calculations (where n is the number of bodies), which were all happening simultaneously on separate threads. This did not scale well, and would immediately cause the JVM to crash upon loading the original Pac-Man level.

Tile Shuffling - Our tile shuffling algorithm was simple. We used the concept of a vertex's degree in graph theory. Each tile was treated as a node. The degrees of each node were calculated to be a number from 0 to 4. Starting at a random lowest-degree node of degree greater than 0, we would carve a short path out of the board and place them in a connected sequence of tiles starting at a random highest-degree node.

The algorithm is not perfect, but it does a fairly good job of ensuring that the shuffled tiles do not result in being walled off from the rest of the map. Being walled off can still happen, but this algorithm does a fairly good job and doesn't require much computation. What we discovered was that the problem of tile shuffling while ensuring all open tiles were still connected and that you did not unfairly trap the player with a ghost is a fairly difficult problem. In fact, it required more computational power than we were willing to take away from the other AI computations.

Summary of our Algorithms Process:

Upon implementing BFS, we realized that it would be an extremely exhausting process to repeat at every single game update. We still used the process of breadth-first searching as part of the setup for our other more creative algorithms, but using it naively would be too hard. It's bad enough with just Pac-Man, let alone also using it on the ghost AIs. Not to mention each level of Pac-Man has about 300 dots, so that's 300 goals for Pac-Man to sift through. Throw all of those goals into a priority queue that is constantly updating and re-sorting the hundreds of goals based on which is closest to Pac-Man and it would have been quite a mess.

In response to realizing simple breadth-first searching could not be the bulk of our search process, we turned to A*. If there was a single pellet at the top left of the board and Pac-Man was in the middle of the board, using BFS would look at just about every single tile on the board. A* is especially good on Pac-Man boards for carving out a single path on average. If BFS was visiting n nodes, then A* is like to only have to visit about log(n) node. A* definitely performed significantly better in most scenarios. The exact performance difference is more complicated and will be visited in the results section, but this was our reasoning behind checking out A*. A* is great for a single source and a single goal, but it did not solve the problem of necessitating a priority queue of all 300 or so dots on the board. This is why we sought a new approach.

The gravity map approach we created ended up being a much better fit for the problem. There was no need for the costly priority queue and there was also no longer a need for direct searching. The gravity *field* approach is theoretically parallelizable and far more viable than the gravity map, but we ran into implementation difficulty after implementation difficulty.

Programming: The Problem

There are many open-source remakes of PacMan; unfortunately, we could not find any which could natively support non-determinism in terms of tile-shuffling, nor could we find any with support for injection of player AI. We would have had to change most of the codebase in order to create support for these features because of how poorly these systems were designed in terms of extension.

Programming: The Solution

We decided to build our own engine. We based the system on feature injection and maintained a minimal core. The system is multithreaded so AI processing can not directly interfere with the game logic or update cycle speed. We followed the SOLID principles of software design so we could have a strong ability to extend the functionality of the engine.

We built everything that we used, excluding the JVM and the LuaJ interpreter. All of the code which is specific to this AI project is written in LuaJ and injected into our Java engine at runtime, so we can actively modify and reload the AI components while the engine is running.

Programming: The Artificial Intelligence

We were able to mix and match many AI approaches over the course of our development. We implemented all of the algorithms that we used.

Search Approaches:

- **BFS** search for a specific goal tile in a tile-based maze.

- **A*** search for a specific goal tile in a tile-based maze using manhattan distance as the heuristic.

Complete Solutions:

- **Gravity Map** model for an entire tile-based board using a degeneracy function over distance (similar to the inverse-square law) and customizable weights for bodies on the tile-based board. The build time is quadratic and the query time is constant.

- **Gravity Field** model for an entire tile-based board based on the same principles as the Gravity Map, but by calculating distances as inputs (in parallel) as a system of equations resolved at query time. The build time is the length of the execution of the longest BFS (linear in terms of tiles) and the query time is linear (in terms of bodies).

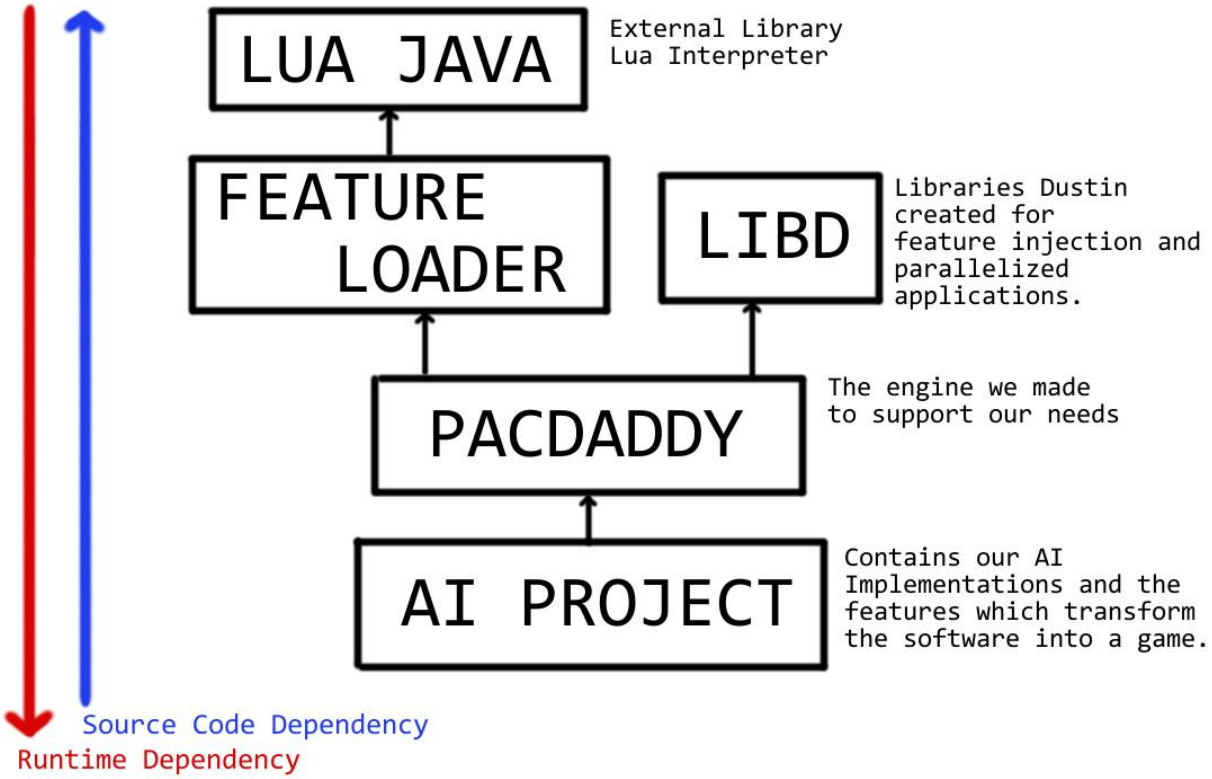
Non-Deterministic AI Elements:

- **Tile Shuffling** for a tile-based board which must:
    - Teleport bodies on the tiles to maintain a non-deterministic traversable environment
    - Maintain a set of traversable hallways so the board can be navigated by moving bodies

  The build time is linear and the run time is theoretically constant, but we messed up.

- **Non-Determinism in Ghost AI** via a random number generator. The ghosts will flee or chase independently at random times, and for a random amount of time. The run time of the RNG is constant, while the Ghost AI is defined by a set of Gravity Maps.

Programming: Software System Design



Work Division

Most of the efforts for this project were exhibited in our off-machine meetings where we actively discussed pros and cons of different approaches, and reasoned about the general implementation of our codebase and AI algorithms.

Most of our code was developed on Dustin's machines - because early on we adopted a work style based on pair programming and frequent friendly meetings, and his home just happened to be the midpoint residence between Nick and Andrew. We are also not the best at resolving version control issues, so we agreed it would be safer to push all major changes from the same machine. Because of our mixed strategy and ignorance, we successfully managed to split the work as evenly as possible - though we all have our individual strengths and weaknesses, and Dustin would sometimes add unnecessary hacks in the wee hours of the night without any prior permission. Overall, the buddy system approach managed to be both a safeguard against bugs and a genuinely fun experience.

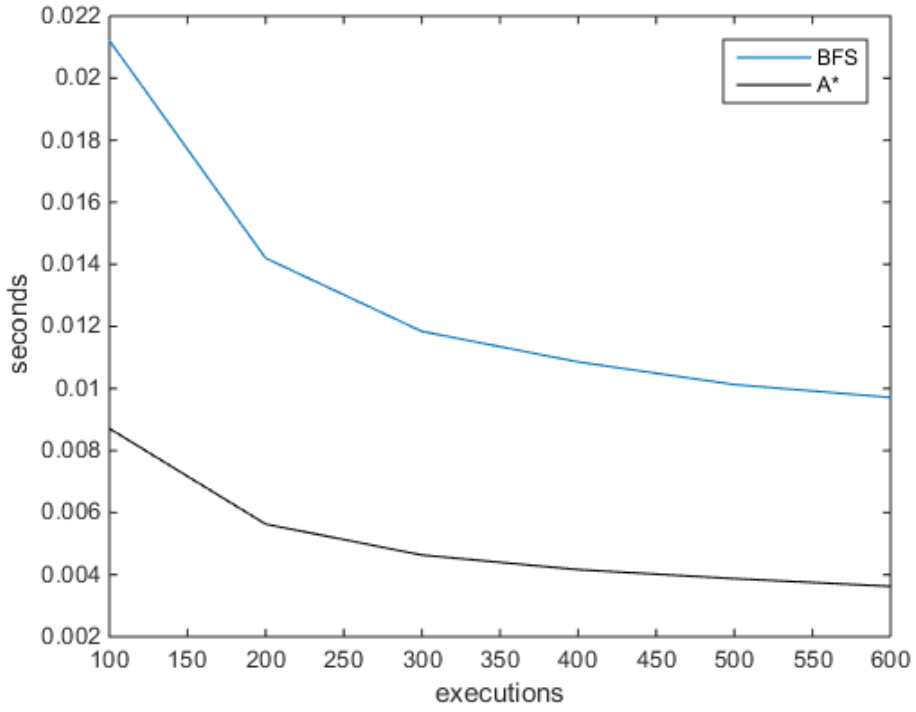
Results

As an overall recap, we first thought simple searching would be sufficient as a starting point in our algorithm. Breadth-first searching worked fine for single starting locations and single goals and A* worked even better. However, our problem introduced a heavy degree of non-determinism which forced us to implement AIs that can make decisions as quickly as possible. Due to the fact that the graph below is from a board with zero enemies and only one pellet for Pac-Man to eat, we were afraid of the scalability of simple searches. On top of that, we also would have had to shove every item of interest into a priority queue for the entity that was interested in those items. Pac-Man would need a priority queue of every single dot and energizer on the board and that queue would constantly be updating as he's moving around. In addition to the priority queue, he would also need separate intelligence for reacting to enemies. It was too slow of an approach.

On top of the lack of scalability and utility we were getting from plain searching, we did not have the option of implementing classic multi-agent methods. For example, generating a minimax tree and pruning it would have been wasteful computation. The actual process of searching through the tree was not the computationally intensive part, but rather building that tree itself. Deliberately applying search logic to the board is what we had to shy away from because of the slow associated data structures for both the search and the prioritizing of goals.

Our gravity map approach was specifically created to avoid the computational cost of the aforementioned data structures.

A* vs BFS



The above graph resulted from a simple maze-like board about 200 tiles in size with Pac-Man, no enemies, no tile-shuffling, and a single pellet on the opposite side of the board from Pac-Man. As expected, A* performed much faster. Breadth-first search was more than likely searching the entire board for at least the first half of the executions. If the pellet was on one corner of the board and Pac-Man is on the opposite end of the board, he will necessarily search the entire board until he passed the midpoint between his starting location and the goal pellet. A* on the other hand scales very nicely, thanks to the Manhattan distance as a heuristic.

Gravity Map Results

The gravity map is a more complete solution than either A* or BFS, so our test conditions included far more pellets on the board and even ghosts for some of them. The provided times and graphs are indicative of how long it took Pac-Man to make his decisions using the gravity map approach. Our primary goal was to get those decision times as low as possible so as to be able to run in real time. We do not have the times it takes for the ghosts to make their decisions because their gravity computations are very simple and, barring slight fluctuations, remain constant.

As can be seen from the maze levels, Pac-Man made his decisions on a board with twice as many tiles as the BFS and A* conditions and 7, 8, and 9 times as many pellets in only twice

as much time. In addition to this, the second and third maze levels both had ghosts on them also utilizing the gravity map method.

For the partially populated level, Pac-Man still made his decisions fast enough to avoid all ghosts and be considered playing in real time. There was only a handful of pellets, but there were five ghosts.

The large room level was a proof of concept that Pac-Man could freely roam outside of a maze with the gravity map and still avoid all enemies, which he did. There were a few pellets on this level, but it was mostly open space. This is where our gravity map approach kind of shines, because instead of a single goal, Pac-Man just “flows” with what’s happening around him. He will cruise around or stay on the same two tiles until a ghost gets too close in which case he will immediately dart off in the other direction. The gravity map approach lends itself to somewhat human-like behavior for this layout.

Our fully populated level is designed after the classic Pac-Man arcade layout. It has all of the dots and all of the ghosts. As can be seen, there were 255 bodies on the screen, so Pac-Man was doing about that many breadth-first searches in the beginning. Of course, as time goes on and more pellets get eaten the number of bodies on the screen go down and therefore the time to generate the gravity map also goes down. Pac-Man does not behave well in the beginning. At an almost 1-second average for making a decision, he’s falling slightly under what we consider to be human-like. We attempted to fix the scalability issue with parallelization, but we were unsuccessful.

Fortunately, our gravity map algorithm is quadratic in its build time. For every tile that has something Pac-Man is interested in, let’s call them t, a BFS must be performed at t to update the gravity map. This all happens every cycle, so $O(tn + tm)$ where t is the set of tiles with dots, energizers or ghosts and n is tiles and m is edges between tiles. Though it failed to meet our expectations, it is a good solution considering the difficulty of the problem and the fact that the query time is constant.

Unfortunately, our gravity map algorithm is slightly too slow in full classic Pac-Man conditions to be considered entirely human-like in its decision making.

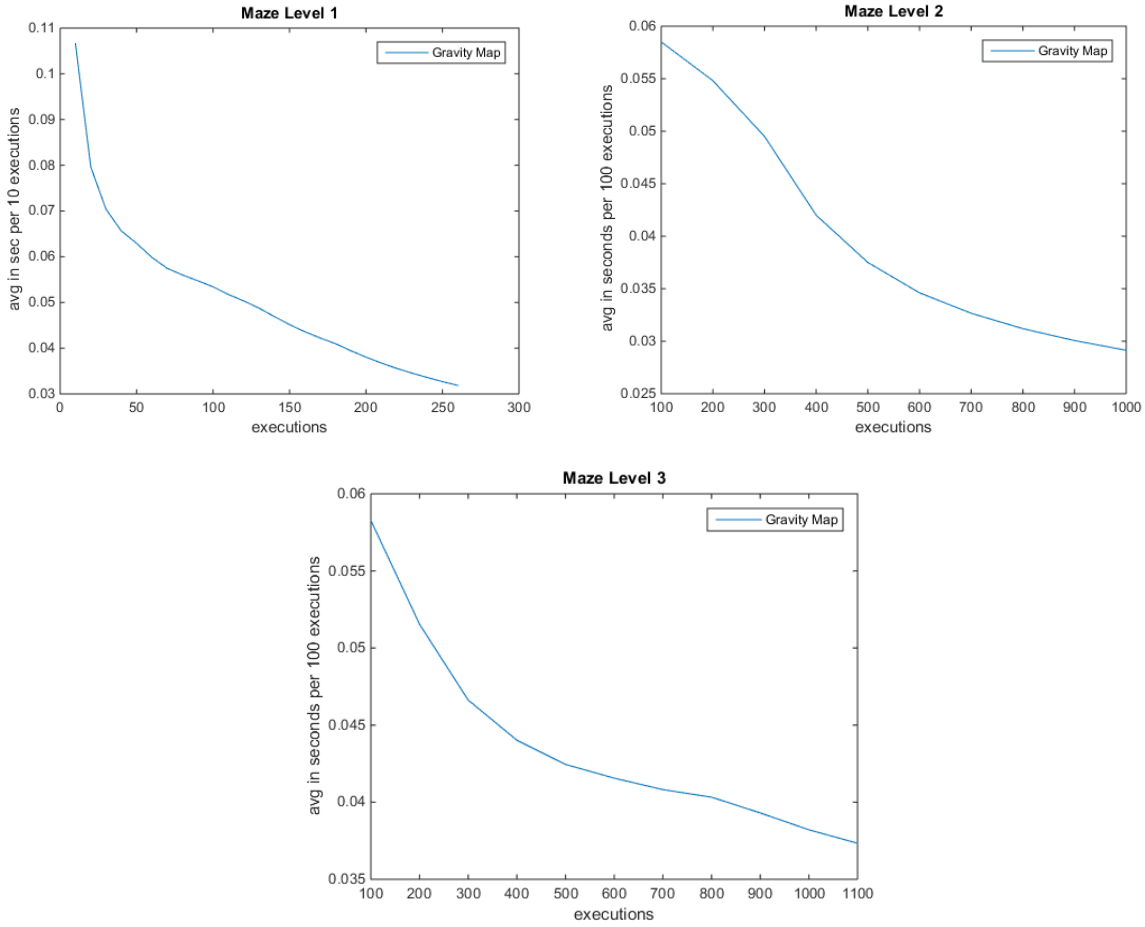
The Gravity Map Approach:

Given that the update speed is 11 updates per second,
Upper bound for a seamless PacMan AI integration is: 0.090909091 seconds per execution

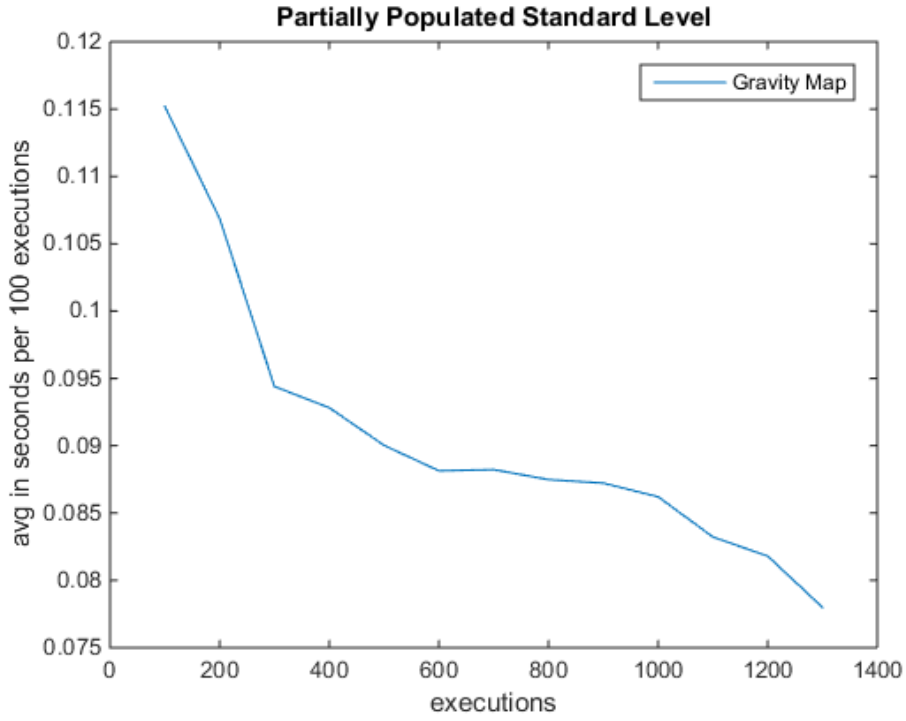
Average execution time in seconds when using the Gravity Map approach:

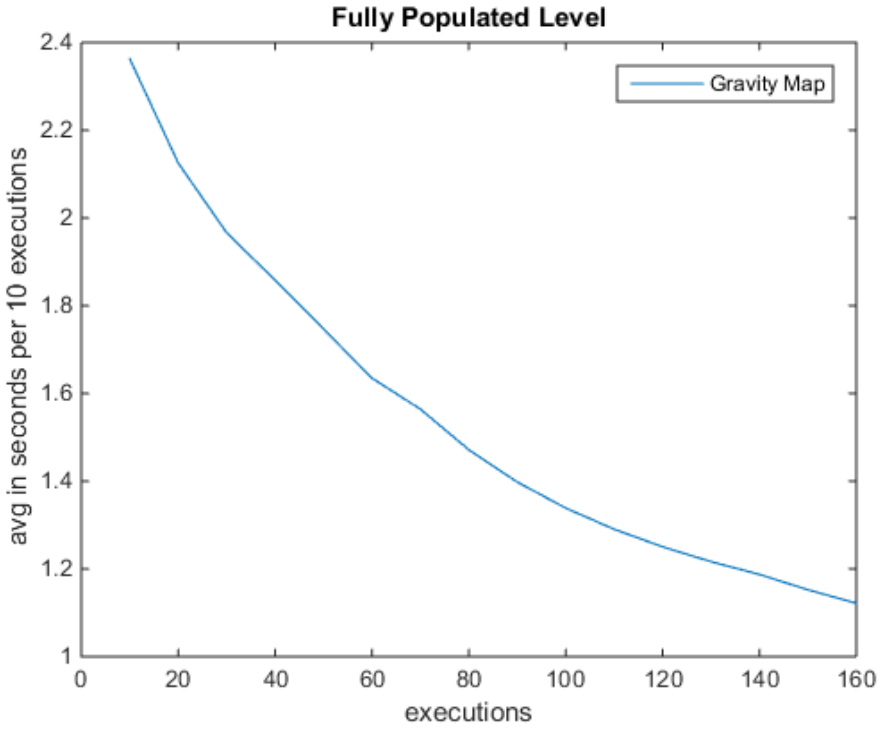
Applied Levels	Tiles	Bodies	Average execution time in seconds
Maze Levels 1, 2, 3	462	7, 8, 9	0.039997778
Partially Populated Base Level	868	15	0.090739506
Large Room	1008	6	0.191105416
Fully Populated Base Level	868	255	0.824644375

Custom Maze Level Comparisons with Varying Body Counts:

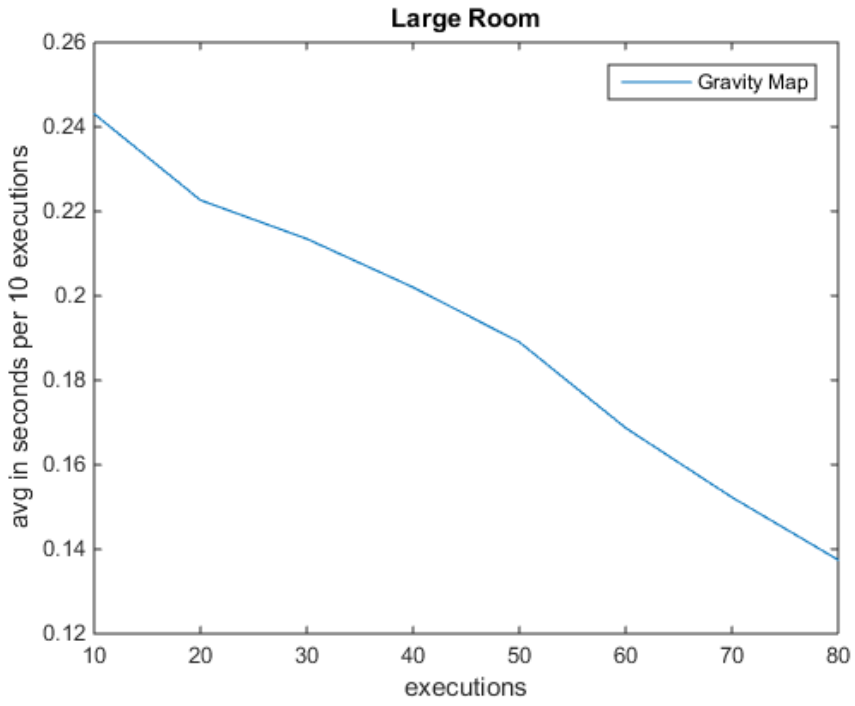


Comparisons for the Standard Pac-Man level:





A large level with no defined walls:



Conclusion

As far as insights, we learned that creating a Pac-Man AI is a really difficult problem and that creating a Pac-Man AI for non-deterministic, chaotic conditions is even harder. We learned that standard search algorithms were not the most particularly useful approach when performance became the most important factor.

If we had to start over and do it all from scratch, we would have allocated more time into creating an efficient priority queue designed specifically for our project. We had to create our own in Lua which worked okay, but it was not satisfactory for the possible BFS/A* approach where Pac-Man has all of the dots and energizers in a priority queue. If we had more time to think about how to tailor a priority queue to these purposes, we probably could have really improved performance by hybridizing the gravity *map* for the ghosts and the gravity *field* for Pac-Man. Obviously, getting the gravity *field* working is the first thing we would implement if we were given another 10 weeks. Our biggest goal was getting Pac-Man to react in time comparable to a human on classic-sized boards and we fell just short of that.

Our group faced a lot of limitations because we built the project from the ground up with essentially no outside libraries. Despite hundreds of crazy runtime bugs and random crashes, we did our best and overcame them through countless hours of pair-programming.

Appendix

Resources:

All of the system components used in our project can be found on GitHub.

Feature Loader: https://github.com/misterdustinface/FeatureLoader
LibD: https://github.com/misterdustinface/LibD
PacDaddy: https://github.com/misterdustinface/PacDaddy
AI Project: https://github.com/misterdustinface/AI-Project-Mac-Pan

External Library:

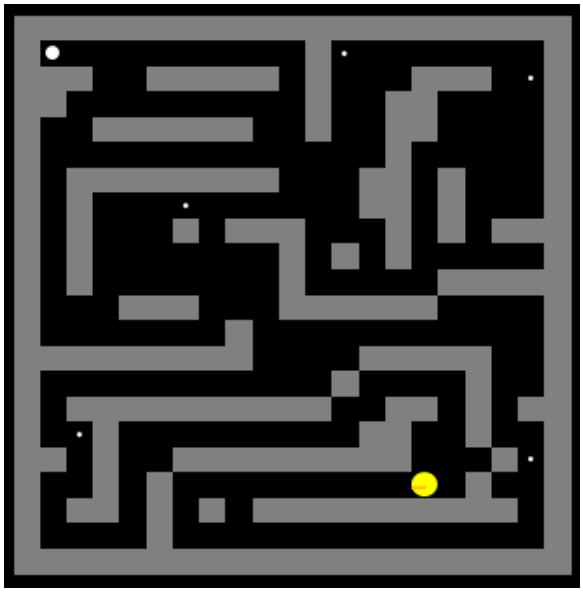
LuaJ: http://luaj.org/luaj.html
LuaJ download: http://sourceforge.net/projects/luaj/
LuaJ readme: http://www.luaj.org/luaj/3.0/README.html

References:

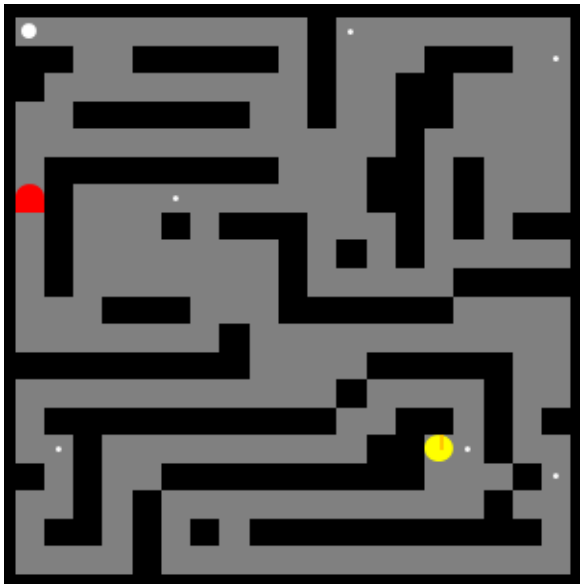
[1] - https://github.com/jcarrillo7/PacMan-AI
[2] - http://ai.berkeley.edu/project_overview.html
[3] - http://yahozna.dyndns.org/projects/pacman/index.html

Images of Levels: (Following Pages)

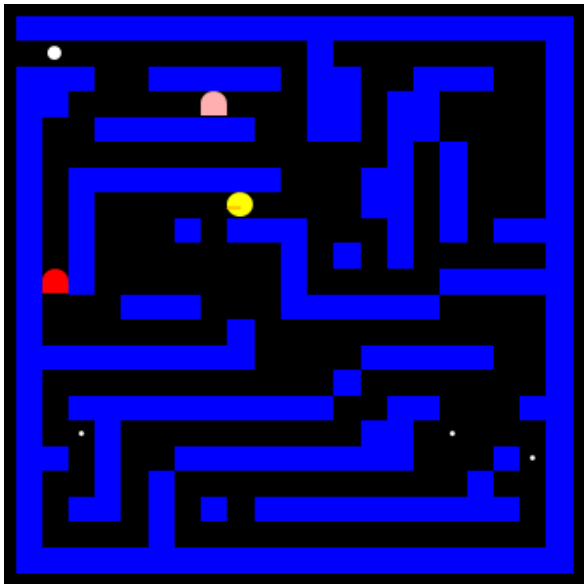
Maze 1



Maze 2

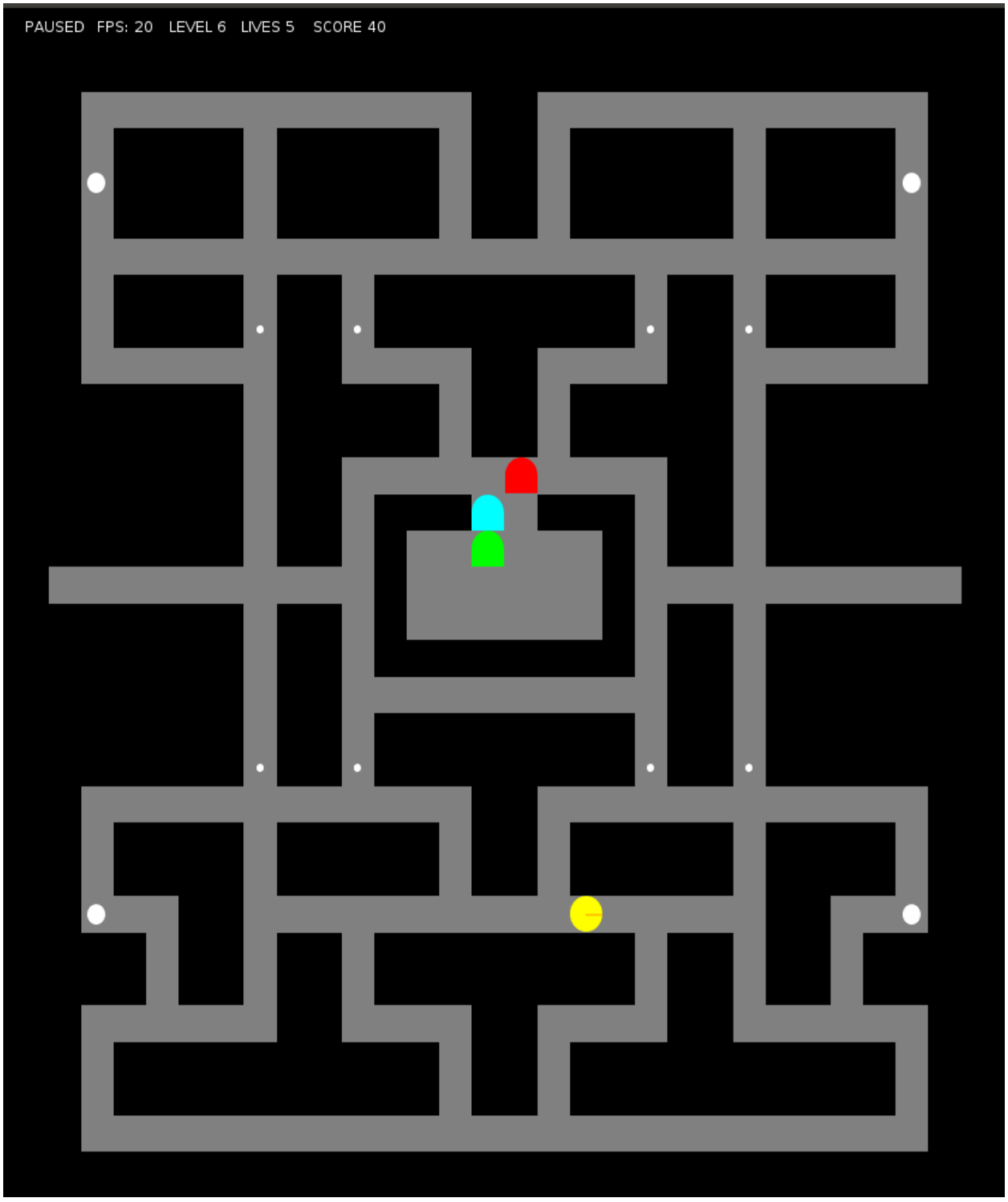


Maze 3 With Active Wall Shuffling

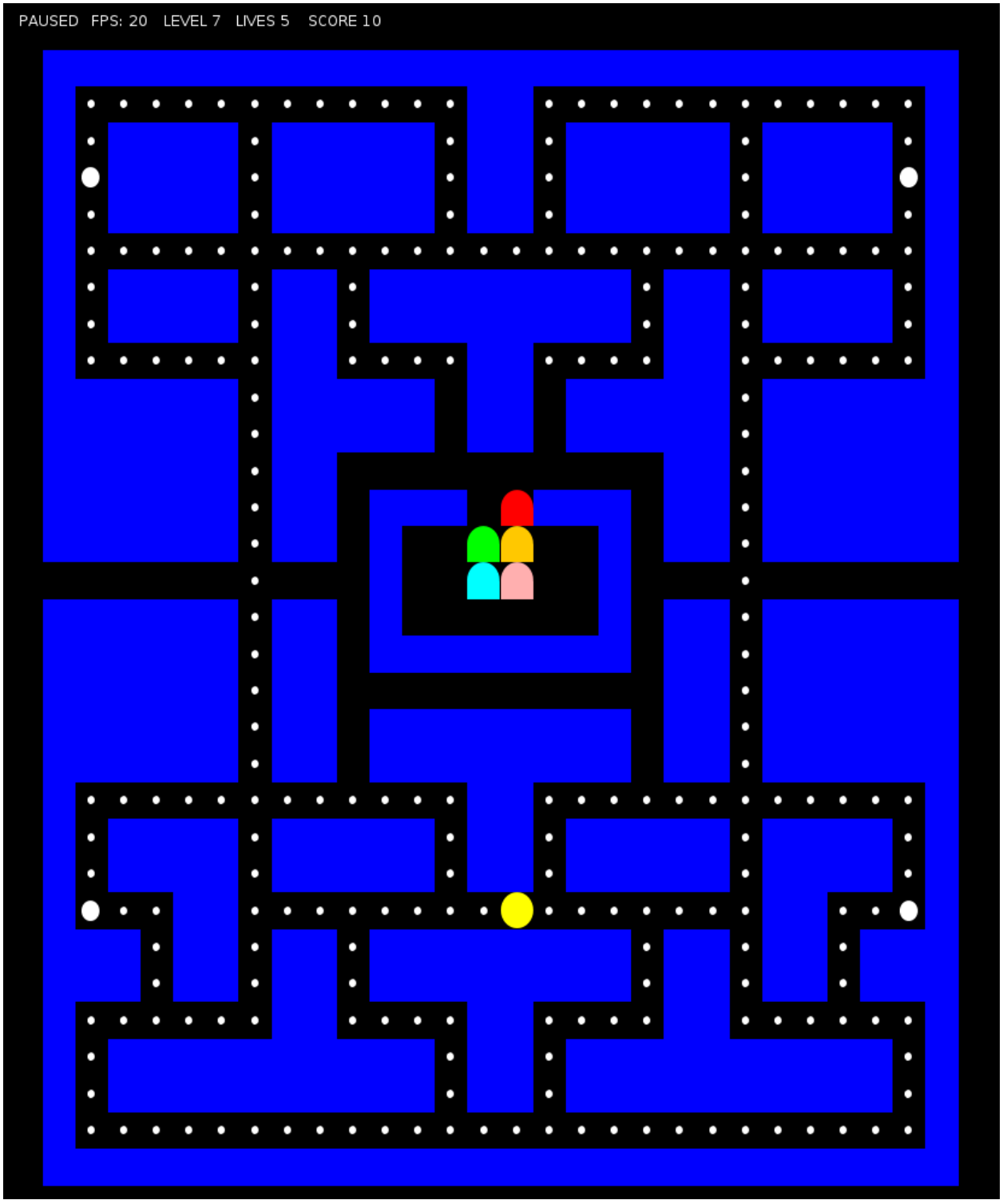


Large Room

Partially Populated Base Level



Fully Populated Base Level



Fully Populated Base Level with Active Wall Shuffling