# Game AI Game Engine

<table>
<thead>
<tr>
<th>Basics: Movers, Agents, and Game Worlds</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mover</td>
<td>1</td>
</tr>
<tr>
<td>Sprite</td>
<td>1</td>
</tr>
<tr>
<td>Decoration</td>
<td>1</td>
</tr>
<tr>
<td>Blocker</td>
<td>1</td>
</tr>
<tr>
<td>Mover</td>
<td>1</td>
</tr>
<tr>
<td>Agent</td>
<td>2</td>
</tr>
<tr>
<td>GameWorld</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Obstacles</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obstacle</td>
<td>6</td>
</tr>
<tr>
<td>ManualObstacle</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Navigators</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Navigator</td>
<td>7</td>
</tr>
<tr>
<td>Navigating Grids</td>
<td>9</td>
</tr>
<tr>
<td>GridNavigator</td>
<td>9</td>
</tr>
<tr>
<td>RandomGridNavigator</td>
<td>9</td>
</tr>
<tr>
<td>GreedyGridNavigator</td>
<td>10</td>
</tr>
<tr>
<td>Navigating Path Networks</td>
<td>10</td>
</tr>
<tr>
<td>PathNetworkNavigator</td>
<td>10</td>
</tr>
<tr>
<td>RandomNavigator</td>
<td>10</td>
</tr>
<tr>
<td>NavMeshNavigator</td>
<td>10</td>
</tr>
<tr>
<td>RandomNavMeshNavigator</td>
<td>10</td>
</tr>
<tr>
<td>AStarNavigator</td>
<td>11</td>
</tr>
<tr>
<td>APSPNavigator</td>
<td>11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Specialized Agents</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generally Useful Agents</td>
<td>11</td>
</tr>
<tr>
<td>GhostAgent</td>
<td>11</td>
</tr>
<tr>
<td>Gatherer</td>
<td>11</td>
</tr>
<tr>
<td>NearestGatherer</td>
<td>12</td>
</tr>
<tr>
<td>StateAgent</td>
<td>12</td>
</tr>
<tr>
<td>VisionAgent</td>
<td>12</td>
</tr>
<tr>
<td>Agents for MOBAs</td>
<td>12</td>
</tr>
<tr>
<td>MOBAAgent</td>
<td>12</td>
</tr>
<tr>
<td>Minion</td>
<td>13</td>
</tr>
<tr>
<td>MyMinion</td>
<td>13</td>
</tr>
<tr>
<td>Hero</td>
<td>13</td>
</tr>
<tr>
<td>MyHero</td>
<td>14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bases and Towers</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>14</td>
</tr>
<tr>
<td>Tower</td>
<td>15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bullets</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bullet</td>
<td>15</td>
</tr>
<tr>
<td>MOBABullet</td>
<td>16</td>
</tr>
<tr>
<td>SmallBullet</td>
<td>...............................................</td>
</tr>
<tr>
<td>BigBullet</td>
<td>..................................................................</td>
</tr>
<tr>
<td>TowerBullet</td>
<td>..................................................................</td>
</tr>
<tr>
<td>BaseBullet</td>
<td>..................................................................</td>
</tr>
<tr>
<td><strong>Specialized Game Worlds</strong></td>
<td>...............................................</td>
</tr>
<tr>
<td>GatedWorld</td>
<td>..................................................................</td>
</tr>
<tr>
<td>Gate</td>
<td>..................................................................</td>
</tr>
<tr>
<td>MOBAWorld</td>
<td>..................................................................</td>
</tr>
<tr>
<td><strong>Agent Behavior Controllers</strong></td>
<td>...............................................</td>
</tr>
<tr>
<td>State Machines</td>
<td>..........................................................</td>
</tr>
<tr>
<td>StateMachine</td>
<td>..................................................................</td>
</tr>
<tr>
<td>State</td>
<td>..................................................................</td>
</tr>
<tr>
<td>Behavior Trees</td>
<td>..........................................................</td>
</tr>
<tr>
<td>BehaviorTree</td>
<td>..................................................................</td>
</tr>
<tr>
<td>BTNNode</td>
<td>..................................................................</td>
</tr>
<tr>
<td>Selector</td>
<td>..................................................................</td>
</tr>
<tr>
<td>Sequence</td>
<td>..................................................................</td>
</tr>
<tr>
<td><strong>Binary Space Partitions</strong></td>
<td>...............................................</td>
</tr>
<tr>
<td>BSPGameWorld</td>
<td>..................................................................</td>
</tr>
<tr>
<td>BSPNode</td>
<td>..................................................................</td>
</tr>
</tbody>
</table>
**Basics: Movers, Agents, and Game Worlds**

This section describes some of the most basic object types, including Agents, Movers, and GameWorlds.

**Thing**

A Thing is the base class for most objects in GAIGE that manifest themselves graphically. It supports the functionality to receive an update from the GameWorld and a notification of a collision

Member functions:

- **update(delta):** The GameWorld calls update on objects each tick of the game clock. Delta is the number of milliseconds since the last tick.
- **Collision(thing):** The GameWorld calls this on a Thing when it collides with another Thing. The parameter is a reference to the thing that just collided.

**Sprite**

Sprite is part of the PyGame package, representing things that can be rendered to screen.

**Decoration**

A Decoration is a Sprite with some convenience code for placing it at a certain position with a certain rotation.

**Blocker**

A blocker is something that impedes the movement of other things. There are no member variables or functions.

**Mover**

A Mover is a Thing and a Sprite. Movers are things that manifest themselves graphically on the screen and are capable of moving around the game world. The Mover class itself does not define how an object moves, which must be provided by specializations, because different things can move in different ways.

Member variables:

- **rect:** A data structure of type pygame.Rect, recording the rectangular bounding box around the sprite.
- **image:** A reference to image that is loaded into the sprite, rotated to reflect the orientation of the Mover.
- **originalImage:** the original image in its original orientation.
- **orientation:** direction agent is facing. Measured in degrees, with 0 indicating “to the right.”
- **world:** A reference to the GameWorld.
• speed: How fast the Mover is moving. Represented as a tuple of the form (horizontal_speed, vertical_speed).
• radius: The distance from the center of the sprite to the farthest point on the bounding box. Radius can be used as a bounding circle.
• owner: The thing that created this Mover.
• alive: A Boolean indicating whether this object is alive or not.

Member functions:
• getRadius(): return the radius around the sprite.
• turnToFace(pos): Changes the orientation of the Mover so that it is facing toward a given (x, y) position.
• turnToAngle(angle): Changes the orientation of the Mover to the given angle, in degrees. 0 = “to the right”
• getLocation(): return the (x, y) position of the center of the sprite.
• getOrientation(): return the angle of the Mover, in degrees.
• getOwner(): return the owner of the Mover.
• isAlive(): returns True if the Mover is still alive.
• die(): Makes the Mover not alive.

Agent

An Agent is a type of Mover that is either controlled by the player or is controlled by AI. Agents know how to move in straight lines, but do not know how to avoid obstacles. Agents also have hitpoints (a measure of health, which, when depleted, causes the Agent to die) and can shoot a weapon.

Because an Agent is not very smart about how it moves, another object, a Navigator, will tell the Agent exactly how to move to get from an origin point to a target destination point in the most efficient manner without running into obstacles.

Member variables:
• moveTarget: Indicates that the Agent should move toward this point, represented as a tuple (x, y). This variable is None when the Agent is not moving, and a non-None value when moving.
• moveOrigin: Indicates the (x, y) point that the Agent started moving from.
• navigator: A reference to a Navigator object that intelligently controls the Agent’s movement.
• firerate: How often the Agent can shoot.
• firetimer: how long since the last time the Agent shot.
• canfire: A Boolean indicating that it is possible for the Agent to shoot.
• hitpoints: The measure of the Agent’s health. When hitpoints is zero, the Agent dies.
• team: A symbol referring to a team that the Agent is a member of.
• distanceTraveled: Agents keep track of how far they have traveled since creation.
Member functions:

- **update(delta)**: Called by the GameWorld every tick. Update() determines if the Agent is moving and if so, moves the sprite in the direction toward moveTarget at the appropriate speed.
- **doneMoving()**: called when the Agent has reached it's moveTarget.
- **collision(thing)**: Called by the GameWorld when the Agent intersects another object that can be collided with. Agents stop moving when a collision occurs.
- **moveToTarget(pos)**: Instructs the Agent to begin moving towards the given (x, y) point in a straight line.
- **setNavigator(navigator)**: Sets the Navigator object.
- **navigateTo(pos)**: Instructs the Agent to move itself from its current location to the given (x, y) point. navigateTo() invokes the Navigator to ensure that the Agent can arrive at the target point in the shortest manner and without colliding with obstacles.
- **shoot()**: Shoots the gun.
- **setTeam(team)**: Sets the Agent’s team designator to the given symbol.
- **getTeam()**: Returns the Agent’s team designator symbol.
- **damage(amount)**: This function is called when the Agent receives damage of the given amount.
- **start()**: Indicates that the Agent is now fully initialized and ready to being operation.
- **stop()**: Indicates that the Agent is no longer operational.
- **stopMoving()**: Instructs the Agent to stop moving and forget its moveTarget.
- **isMoving()**: Indicates whether the Agent is currently moving or not.
- **getMoveTarget()**: Returns the Agent’s target destination point.
- **getHitpoints()**: Returns the number of hitpoints.
- **canFire()**: Returns true if the Agent is able to shoot at this moment.

**GameWorld**

A GameWorld contains all information necessary to render the game world on screen and execute the rules of the game. The GameWorld maintains the game’s internal clock, updating all update-able objects every tick, and determines whether collisions between objects have occurred. Creation of a GameWorld also creates the graphical display for the game.

All games must have one player-controlled Agent. If the size of the game map is larger than the size of the rendering window, the window follows the player’s agent so that it is always visible.

The user can control the player-controlled Agent by clicking the mouse at the point on the screen the Agent should move to. The default behavior of an Agent is to move straight to that location, although this behavior can be overridden with AI Navigators to avoid obstacles. If the user presses the ‘space’ key, the Agent will shoot its weapon. If the user presses the ‘d’ key, the Agent will print out the total distance it has moved since the beginning of the game.
The screenshot below shows the default game world, configured with three obstacles, and a number of randomly placed resources that can be collected by the player. The player-controlled Agent is in the center of the screen, facing to the right.

Member variables:

- **dimensions**: The size of the game map, represented as the tuple (width, height).
- **screen**: The rendering window.
- **background**: The background surface of the screen.
- **agent**: The Agent that is controlled by the player.
- **sprites**: A list of all sprites (player and NPCs), for rendering purposes.
- **npcs**: A list of AI-controlled Agents.
- **bullets**: A list of all Bullets currently in the world.
- **resources**: A list of all Resources currently in the world.
- **movers**: A list of all Movers currently in the world.
- **obstacles**: A list of terrain features that can impede the movement of Movers.
- **points**: A list of points of every Obstacle (Obstacles are polygons), and the four corners of the map. A point is a tuple (x, y).
- **lines**: A list of all lines of every Obstacle (Obstacles are polygons), and the borders of the map. A line is a tuple of the form ((x1, y1), (x2, y2)).
• clock: Elapsed time since the game started.
• destinations: A hash table that indicates points, of the form \((x, y)\), that Agents of different class types can be at without intersecting with obstacles. The hash table is keyed of class type and each entry contains a list of points.
• getPoints(): Return all the points of Obstacles in the map, plus the four corners of the map.
• getLines(): Return all lines on all Obstacles in the map, plus the borders of the map.
• getLinesWithoutBorders(): Return all the lines of all Obstacles in the map, but does not include the lines along borders of the map.
• getObstacles(): Return a list of Obstacles in the map.
• getDimensions(): Return the dimensions of the map, as a tuple \((\text{width}, \text{height})\).
• setPlayerAgent(agent): Set the Agent that is controlled by the player.
• initializeTerrain(polys, color, linewidth, sprite): Makes a number of ManualObstacle objects. polys is a list of polygons \((\text{poly1}, \text{poly2}, \ldots)\) where a polygon is a tuple \(\((x_1, y_1), (x_2, y_2), (x_3, y_3), \ldots\)\). Color indicates the line color of the polygons, given as an \((r, g, b)\) tuple. Linewidth is the thickness of the line, sprite is a string indicating a filename of an image for the resources.
• initializeResources(points, resource): Makes a number of SimpleResources, one located at each \((x, y)\) in the list of points. Resource is a string indicating a filename of an image for the resources.
• run(): Starts the game loop, which is an infinite loop that calls update() on all objects every tick, and checks for collisions every tick.
• drawWorld(): Render the world.
• handleEvents(): Called whenever there is mouse input or a keyboard key press.
• doMouseUp(): Called whenever there is a mouse click. Default behavior is to instruct the player's Agent to navigate to the clicked position.
• doKeydown(key): Called whenever there is a keyboard button press. Default behavior is for 'spacebar' to shoot and 'd' to print out the distance the player's Agent has traveled since the beginning of the game.
• worldCollisionTest(): Check if Movers have collided with each other and with Obstacles.
• update(): Called every tick.
• collision(): Called if something collides with the world (e.g., the borders of the map).
• addBullet(bullet): Called when a Bullet is spawned.
• deleteBullet(bullet): Called when a Bullet disappears from the world.
• addResource(res): Called when a Resource is spawned.
• deleteResource(red): Called when a Resource disappears from the world.
• addNPC(npc): Called when an AI-controlled Agent is spawned.
• deleteNPC(nps): Called when an AI-controlled Agent dies.
• getVisible(position, orientation, viewangle, type): Returns a list of all Movers that are visible from a given position \((x, y)\), with a given orientation in degrees,
and a view angle in degrees around the orientation. If type is not None, then only Movers of the given sub-type are returned.

- **computeFreeLocations(agent):** Computes destination points \((x, y)\) that an Agent can go to that are guaranteed not to intersect with any Obstacles. Only needs to be called once, storing all points in the destinations hash table.
- **getFreeLocations(agent):** Returns a list of points \((x, y)\) that an Agent can be at in the game world that will not intersect with any Obstacles.
- **getNPCs():** Return a list of AI-controlled Agents.
- **getAgent():** Return a reference to the player-controlled Agent.
- **getBullets():** Return a list of references to Bullets currently in the world.

**Obstacles**

Obstacles are map terrain features that impede the movement of Movers. This section describes the Obstacle class and the main sub-class, ManualObstacle.

**Obstacle**

An Obstacle is an object in the game world that impedes the movement of Movers. Obstacles are defined by the polygonal border, and may be filled in with sprites such as trees to given them a more appealing look. Obstacles are both Things and Blockers.

Member variables:

- **points:** A list of all of the points making up the polygon border of the Obstacles. Points are ordered in monotonically increasing or decreasing angle around the center point of the polygon. A point is a tuple, \((x, y)\).
- **pos:** The center point of the polygon as a tuple, \((x, y)\).
- **lines:** A list of all lines that make up the polygon border of the obstacle. A line is a tuple of the form \(((x_1, y_2), (x_2, y_2))\). Each \((x, y)\) is also a point in points.
- **surface:** Each Obstacle is rendered on its own pygame.Surface.
- **rect:** The rectangular bounding box around the Obstacle. A pygame.Rect object.

Member functions:

- **draw():** Draws the Obstacle.
- **getLines():** Returns the list of lines making up the polygon border. A line is a tuple of the form \(((x_1, y_1), (x_2, y_2))\).
- **getPoints():** Returns a list of points of the polygon border. A point is a tuple \((x, y)\).
- **isInPoints(point):** Returns true if the point \((x, y)\) given is a member of the list of polygon points.
- **twoAdjacentPoints(p1, p2):** Returns true if p1 and p2 are points that are polygon points and are adjacent to each other on the polygon.
- **pointInside(point):** Returns true if the given point \((x, y)\) is contained within the polygon.
ManualObstacle

A ManualObstacle is an Obstacle that is created from a list of points. It is the default type of Obstacle used in the GameWorld.

Navigators

Navigators are objects that control the movement of Agents. Think of Agents as being really stupid. Although they manifest themselves as the “characters” in the game world, they do not know how to move (except in straight lines) or how to make decisions. A Navigator is an object that attaches itself to an Agent and helps it to move about the map in an intelligent manner. Think of the Navigator as the “brain” of an Agent that only controls the feet. A Navigator has one primary purpose: to steer the Agent around Obstacles to get to a specified point in the map.

Navigator

The base Navigator class, which must be specialized to intelligently control the Agent. The base Navigator makes no assumptions about whether the game world is grid-based or path network based or what algorithm should be used to move the Agent in the most effective manner while avoiding Obstacles. The default behavior is to move the Agent in a straight line from its current location to the desired target location.

Member variables:

• agent: A reference to the Agent that this Navigator controls.
• world: A reference to the GameWorld object.
• source: The (x, y) point where the agent started navigating from.
• destination: The (x, y) point that the Agent desires to be at.
• path: A list of points ((x1, y1), (x2, y2), (x3, y3), ...) that the Agent should navigate to in order to get to the destination without collision.

Member functions:

• setAgent(agent): Indicates which Agent this Navigator will control.
• setPath(path): Sets the Agent’s path. Path is a list of points (x, y).
• getSource(): Return the point that the Agent is coming from, as a tuple (x, y).
• getDestination(): Return the point that the Agent desires to be, as a tuple (x, y).
• getPath(): Return the list of points that the Agent should travel through to get to the destination.
• setWorld(world): Store the GameWorld object.
• doneMoving(): A callback from the Agent when the Agent has gotten to its moveTarget, which may be a single point in the path or the desired destination.
• checkpoint(): Called when the Agent reaches a point in the path.
• collision(thing): Called when the Agent collides with something.
• smooth(): Updates the path to take any shortcuts, allowing the Agent to get to the destination faster.
• computePath(source, dest): Instruct the Navigator to create a path from the source point to the destination point.
• update(delta): Called every tick. Delta is the number of milliseconds since the last tick.

Using Navigators:
An Agent “outsources” the intelligence about how to move efficiently from one place to another while avoiding collisions to a Navigator object. Think of a Navigator object as an intelligent parasite that sits in the brain of an otherwise stupid Agent. The Agent knows how to pass along all relevant information to the Navigator. In turn, the Navigator makes callbacks to the Agent to make it move and turn.

Navigation starts with an instruction to the Agent to navigateTo() a desired destination. The Agent asks the Navigator to compute a path. The default Navigator just tells the Agent to go straight to the desired destination. However, a more intelligent Navigator such as APSPNavigator or AStarNavigator will perform some computation to create a path, involving a number of intermediate points, called checkpoints. Once the path is created, the Navigator will tell the Agent that its move target is the first checkpoint. This move target is different from the desired destination. The Navigator remembers the desired final destination for the Agent, but the Agent only knows its most immediate next target.
Every tick, the Agent will receive an update() call from the game engine. If the agent is not within ½ radius of its move target (in this example, the intermediate checkpoint), then the Agent will move toward the move target. Navigator will also receive an update() call. Agent will also call smooth() on the Navigator. The purpose of smooth() is to optimize the path based on new information that might have been discovered enroute.

If the agent gets within ½ radius of the move target, it informs the Navigator that it has arrived and requires further instructions. The Agent may be at the final desired destination, or it might have just reached an intermediate checkpoint—it doesn’t know, so the Navigator makes that determination. If the Agent is just at a checkpoint, then it grabs the next checkpoint from the previously computed path and instructs the Agent to go there next. The Navigator calls Checkpoint() on itself in case there are any specialized behaviors that should be performed when a checkpoint is reached. For example, Navigators designed for dynamically changing worlds might check to see if the remaining path has become blocked.

**Navigating Grids**

**GridNavigator**

A Navigator that works on grids. The grid is a 2D array of Booleans where a True indicates navigable space and a False indicates non-navigable space. The construction of the grid is the topic of **Homework 1**.

Member variables:

- grid: a 2D array of Booleans where each cell is True or False indicating whether the corresponding region of the map is navigable or not.
- dimensions: A tuple (columns, rows) indicating the number of columns and rows in the grid.
- cellSize: the physical size of the region of the map that corresponds to each cell in the grid. This number indicates the width and height of a square region, and is automatically set to the agent’s diameter.

Member functions:

- createGrid(world): Instantiates the grid member variable.
- drawGrid(surface): draw the grid on the given pygame surface.

**RandomGridNavigator**

The RandomGridNavigator is a GridNavigator. When asked to compute a path, it starts with the cell that is closest to the Agent’s current (x, y) position and then randomly adds cells to the up, down, left, or right. If the random walk hits the cell that corresponds to the region closest to the desired destination, the path is complete. The random walk terminates after 10 cells are randomly selected. The generated path is then converted into a list of (x, y) points for the Agent to navigate to in order.
**GreedyGridNavigator**
The GreedyGridNavigator is a GridNavigator. When asked to compute a path, it starts with the cell that is closest to the Agent's current \((x, y)\) position and then adds the next adjacent (up, down, left, right) cell that is closest to the desired destination. The path search terminates when the cell is reached that corresponds to the region closest to the desired destination. If the cell closest to the desired point is not reached after 100 cells are added, the greedy search terminates. The generated path is then converted into a list of \((x, y)\) points for the Agent to navigate in order.

**Navigating Path Networks**

**PathNetworkNavigator**
An abstract Navigator specialization for Agents navigating in continuous worlds with path node networks.

Member functions:
- `drawPathNetwork(surface)`: Draw the path network.

**RandomNavigator**
RandomNavigator is a sub-class of PathNetworkNavigator. To compute a path through a path network, the RandomNavigator chooses a random successor from the path network. Random successor points are added to the path until the path node closest to the desired destination is reached or until 100 points have been added to the path.

**NavMeshNavigator**
A Navigator sub-class of PathNetworkNavigator that assumes that the free navigable space of a continuous world has been partitioned into a Navigation Mesh. A Navigation Mesh is a set of convex hulls that describe open space that Agents can move around in. This class uses the Navigation Mesh to automatically generate a path node network. NavMeshNavigator should be considered an abstract class; it does not create the navigation mesh or the path node network, which is the topic of **Homework 2** (see RandomNavMeshNavigator).

Member functions:
- `createPathNetwork(world)`: creates the navigation mesh and the path network. Default behavior of this abstract class is to do nothing.
- `drawNavMesh(surface)`: draw all the convex hulls of the navigation mesh on the given pygame surface.

**RandomNavMeshNavigator**
RandomNavMeshNavigator is a sub-class of NavMeshNavigator. It automatically creates the navigation mesh (**Homework 2**). When a path is computed, it chooses a random successor from the path network. Random successor points are added to the path until the path node closest to the desired destination is reached or until 100 points have been added to the path.
**APSPNavigator**
The APSPNavigator is a NavMeshNavigator that implements an all-pairs shortest-path algorithm to create a path from the path node closest to the Agent to the path node closest to the desired \((x, y)\) point. The Floyd-Warshall APSP algorithm is used to pre-compute the shortest path from every path node in the path network to every other path node in the network. The implementation of Floyd-Warshall is the topic of **Homework 3**.

Member variables:

- **next**: A dictionary of dictionaries that indicates which node to traverse to next to get to a given destination. That is, next\([p1][p2]\) holds the path node point \((x, y)\) that the Agent should traverse to if it is near \(p1\) and wants to get to \(p2\). \(p1\) and \(p2\) are points of the form \((x, y)\).
- **dist**: A dictionary of dictionaries that indicates the path traversal distance between any two path nodes. That is, dist\([p1][p2]\) is the distance along path links between \(p1\) and \(p2\), where \(p1\) and \(p2\) are points of the form \((x, y)\).

**AStarNavigator**
The AStarNavigator is a NavMeshNavigator that uses the A* algorithm to dynamically compute the path along the path network from the node closest to the Agent to the node closest to the desired \((x, y)\) position. The implementation of the A* algorithm is the topic of **Homework 4**.

**Specialized Agents**
This section gives the various specializations of the Agent class.

**Generally Useful Agents**

**GhostAgent**
A GhostAgent is an Agent that cannot collide with Obstacles. Useful for testing and for providing a player-controlled agent for observing fully autonomously played games.

**Gatherer**
A Gatherer is an Agent that, given a list of target \((x, y)\) positions, navigates around the map to each of the given locations. Typically, the list of targets are the \((x, y)\) positions of Resources, but the targets can be any \((x, y)\). A Gatherer visits the targets in the given order.

Member variables:

- **targets**: A list of \((x, y)\) points.
- **score**: An integer indicating the number of targets visited.

Member functions:

- **setTargets(targets)**: Set the list of targets to be visited. targets is a list of points of the form \(((x1, y1), (x2, y2), ...)\).
• addTarget(target): Add an (x, y) point to the list of targets.
• addToScore(points): increase the score variable by the given number of points.

**NearestGatherer**
A NearestGatherer is a Gatherer. Instead of visiting each target location in the proscribed order, NearestGatherer sorts the list of targets such that the first target is the one closest to the Agent, the second target is the next closest to the first target, and so on. If the NearestGatherer is ever forced to stop moving to collect targets, it automatically asks its Navigator to re-plan.

**StateAgent**
A StateAgent is a type of Agent, and also sub-classes from StateMachine. A StateAgent is an agent that implements a StateMachine. The behavior of the Agent is determined by callbacks from the current State object that is running. One can think of a StateAgent as a very dumb Agent that asks other objects—States—to tell it what to do.

Member functions:
• getStateType(): returns the name of the class that is currently executing. Use this function to easily determine what state the Agent is using to control itself.

**VisionAgent**
A VisionAgent is a type of StateAgent. VisionAgents have a vision cone that determines what they are allowed to be aware of. Without a vision cone, agents can be omniscient. A vision cone sweeps out a cone centered on the direction that the Agent is facing. Anything within that vision cone is visible to the Agent. Although a VisionAgent can still query the GameWorld for lists of things (e.g., enemy Agents, Bullets, etc.), the purpose of the VisionAgent is to restrict what the Agent is allowed to know about. Setting the angle of the vision cone to 360-degrees makes everything visible at all times.

Member variables:
• viewangle: The angle, in degrees, of the vision cone, centered on the direction that the agent is facing.
• visible: A list of Movers that are within the Agent’s vision cone. The list is updated every tick.

Member functions:
• getVisible(): Returns the list of things that are visible to the Agent at this particular time.
• getVisibleType(type): Return the list of things of a certain type that are visible to the Agent. type is a class name.

**Agents for MOBAs**
**MOBAAgent**
A MOBAAgent is a VisionAgent specialized for playing MOBA (Multi-user Online Battle Arenas) games. MOBAAgents remember the maximum number of hitpoints they are allowed to have. They also have a level, which is an integer value that increases the
strength of shots fired. If a MOBAAgent collides with an Obstacle, it dies immediately. MOBAAgents also keep track of a list of points in the map that are free from obstruction in the event that it needs to move to an arbitrary point. This is useful if one needs to implement functionality to move to a point nearby to something else. Additionally, MOBAAgents keep track of the Agent that last performed damage to it, which is used to credit other Agents with kills. MOBAAgent is an abstract base class.

Member variables:
- maxHitpoints: The maximum number of hitpoints the agent is allowed to have.
- lastDamagedBy: A reference to the Agent that last did damage to this Agent.
- Level: The strength modifier. Initially the level of an Agent is 0.

Member functions:
- getMaxHitpoints(): Return the maximum number of hitpoints.
- getPossibleDestinations(): Return a list of (x, y) points that the Agent could travel to that will not cause the Agent to intersect with an Obstacle.
- creditKill(killed): Called when this Agent kills another Agent. The default behavior is to do nothing.
- getLevel(): Return the level value.

Minion
A Minion is a MOBAAgent that is relatively weak in strength and doesn’t have any special abilities. Minion is an abstract base class.

MyMinion
A specialization of Minion that is implemented for Homework 5 and Homework 7.

Hero
A Hero is a MOBAAgent that is relatively powerful and has a number of special abilities.Heroes can dodge, which is implemented as instantly jumping a small distance away from its current position. Heroes have a secondary attack ability called an area effect weapon. The area effect attack does damage to all other Agents within a certain radius of the Agent.Heroes level increases each time it kills another Agent; shooting and area effect attacks increase in damage with each increase in level. When a Hero touches a Base from the same team, the Hero’s hitpoints are restored to maximum. Hero is an abstract base class.

Member variables:
- dodgeRate: How often the Hero can use the dodge ability.
- dodgeTimer: The number of ticks since the last time the dodge ability was used.
- candodge: A Boolean indicating that the dodge ability can be used.
- areaEffectDamage: The amount of damage the area effect attack does.
- canareaeffect: A Boolean indicating that the area effect attack can be used.
- areaEffectRate: How often the Hero can use the area effect attack.
- areaEffectTimer: The number of ticks since the last time the area effect attack was used.
Member functions:

- **dodge(angle):** Use the dodge ability. The angle, in degrees, indicates the direction the Hero should jump. If angle is None, then the Hero will dodge in a random direction.
- **areaEffect():** Use the area effect attack ability.
- **canDodge():** Returns True if the dodge ability can be used.
- **canAreaEffect():** Returns True if the area effect attack can be used.

**MyHero**

A specialization of Hero that is implemented for Homework 6 and Homework 7. MyHero also inherits from BehaviorTree, so it can implement behavior logic as either a state machine or via behavior tree.

**Bases and Towers**

Bases and Towers are specific entities used in Multi-User Online Battle Arenas (MOBAs). Bases spawn Minions and cannot be damaged as long as its team has Towers. Towers are less powerful entities that shoot at opponent Minions and Heros.

**Base**

A Base is a Mover, although by default Bases don’t actually move. Bases spawn Minions at a certain rate until the team has a maximum number of Agents allowed. A Hero will be spawned if there are no Heroes on the team. Bases shoot at enemy Agents, choosing to target Minions before Heroes.

Member variables:

- **team:** A symbol indicating the team the Base is on.
- **hitpoints:** The amount of damage the Base can take before it dies.
- **buildRate:** The number of ticks that must transpire before the Base can spawn more Agents.
- **buildTimer:** The number of ticks that have passed since the last Agent was spawned.
- **nav:** A Reference to a Navigator object. This Navigator will be cloned and given to each newly spawned Agent.
- **minionType:** The name of the class of Minions that will be spawned. Should be a sub-class of Minion.
- **heroType:** The name of the class of Hero that will be spawned. Should be a sub-class of Hero.
- **bulletclass:** The name of the class of Bullet that will be used when the Base shoots.
- **firerate:** The number of ticks that must transpire before the Base can shoot.
- **firetimer:** The number of ticks since the last time the Base shot.

Member functions:
• setNavigator(nav): A reference to a Navigator object that will be cloned and given to Agents when they are spawned by the Base.
• getTeam(): Return the symbol that indicates which team this Base is on.
• setTeam(team): Set the symbol that indicates the Base's team.
• spawnNPC(type, angle): Create an Agent of the given type. angle is an indicator of where the Agent should be spawned, relative to the Base’s centerpoint.
• damage(amount): Called when the Base takes damage.
• shoot(): Instructs the Base to fire the weapon.
• getHitpoints(): Return the number of hitpoints the Base has.

Tower

A Tower is a stationary entity that shoots at enemy Agents. Tower is a sub-class of Mover, but Towers don’t typically move. Towers fire at any enemy Agent within range, preferring to shoot Minions before Heroes.

Member variables:

• team: A symbol indicating the team the Base is on.
• hitpoints: The amount of damage the Base can take before it dies.
• bulletclass: The name of the class of Bullet that will be used when the Base shoots.
• firerate: The number of ticks that must transpire before the Base can shoot.
• firetimer: The number of ticks since the last time the Base shot.

Member functions:

• getTeam(): Return the symbol that indicates which team this Base is on.
• setTeam(team): Set the symbol that indicates the Base’s team.
• damage(amount): Called when the Base takes damage.
• shoot(): Instructs the Base to fire the weapon.
• getHitpoints(): Return the number of hitpoints the Base has.

Bullets

This section enumerates the various types of bullets that can be fired by Agents. Bullets are created when Agents call their shoot() functions and travel in a particular direction at a particular speed until they collide with something. If a bullet collides with an Agent, damage is done to the Agent.

Bullet

This is the default type of bullet and base class for all other bullets. It sub-classes Mover.

Member variables:

• damage: The amount of damage that will be dealt to an Agent if the Bullet collides with an Agent.
• distanceTraveled: The total amount of distance traveled by the Bullet since it is created.

Member functions:
• getDamage(): Return the amount of damage this Bullet is capable of doing to Agents.
• Hit(thing): Called when the Bullet collides with something.

**MOBABullet**

MOBABullets are specializations of Bullet for MOBA (Multi-user Online Battle Arena) games. The primary difference is that MOBABullets have a range, meaning that they disappear after a certain amount of distance has been traveled. MOBABullet should be considered an abstract base class.

Member variables:
• range: The distance the Bullet can travel before disappearing.

**SmallBullet**

A specialization of MOBABullet that does relatively little damage. SmallBullets are typically shot by Minion agents.

**BigBullet**

A specialization of MOBABullet that does more damage than SmallBullets. BigBullets are typically shot by Hero agents.

**TowerBullet**

A specialization of MOBABullet are typically shot by Towers.

**BaseBullet**

A specialization of MOBABullet are typically shot by Bases.

**Specialized Game Worlds**

GameWorld specializations add extra rules to the default game play.

**GatedWorld**

A GatedWorld is a GameWorld in which walls (called “gates”) randomly appear around the environment. Gates appear and disappear at regular intervals. There can be a variable maximum number of Gates. A Gate will never appear too close to an Agent so as to not force an unexpected collision. Note: the GameWorld.getLines() is overridden so that it adds the lines down the center of gates to the list of all lines in the world.
If the player presses the 'g' key, one of the gates will be moved to a position close to where the cursor is positioned.

Member variables:

- `gates`: A list of Gate objects, indicating all the gates that are currently present in the world.
- `numGates`: The maximum number of gates that can be in the world at any time.
- `potentialGates`: A list of lines of the form \(((x_1, y_1), (x_2, y_2))\) where a gate could appear.
- `timer`: The number of ticks since the gates were last moved.
- `alarm`: The number of ticks that must pass before gates can be moved.

Member functions:

- `getNumGates()`: Return the maximum number of gates allowed.
- `getGates()`: Returns a list of lines of the form \(((x_1, y_1), (x_2, y_2))\) that indicate where Gates are currently in the world.
- `makePotentialGates()`: A potential gate will be created between every pair of obstacle points (and corners of the map) possible.
- `getLines()`: Returns all of the obstacles lines (and the borders of the map), including a line through the center of each gate. Returns a list of lines, where a line is a tuple of the form \(((x_1, y_1), (x_2, y_2))\).
- `drawPotentialGates()`: Draws a faint gray line every place a gate could appear.
- `addGateAtNearest(point)`: Forces a gate to appear in the place of a potential gate that is nearby to the given \((x, y)\) point. This function is called when the player presses the 'g' key.

**Gate**

A Gate is a helper class for GatedWorld. A Gate is a Thing and a Blocker. Gates are lines over which a number of decorations are positioned to give the Gate an appearance of thickness.

Member variables:

- `decorations`: A list of Decorations that will be rendered.
- `sprites`: A list of pygame.Sprites for the decorations.
- `line`: The line of the Gate, \(((x_1, y_1), (x_2, y_2))\).

Member functions:

- `getLine()`: Return the line as a tuple of the form \(((x_1, y_1), (x_2, y_2))\).
- `draw(surface)`: Draw the decorations on the given surface.
- `isColliding(rect)`: Is the given rectangle (pygame.Rect) intersected by the line of the Gate?

**MOBAWorld**

A MOBAWorld is a GatedWorld specialized for Multi-user Online Battle Arena (MOBA) games. MOBAWorlds keep track of the game score, and have additional functionality to
keep track of Towers and Bases. Additionally, MOBAWorlds are aware that different
Agents are on different teams and provides functions for handling them differently.
Pressing the ‘j’ key will cause the player-controlled Hero to dodge (jump) in a random
direction. Pressing the ‘a’ key will cause the player-controlled Hero to use the area
effect attack.

Member variables:

• bases: A list of Bases in the game.
• towers: A list of Towers in the game.
• score: A dictionary where each entry is the total number of points scored for
each team. Thus score[teamname] is the number of points scored for the given
team.

Member functions:

• addBase(base): Add a new Base to the game.
• deleteBase(base): Remove a Base from the game.
• addTower(tower): Add a new Tower to the game.
• deleteTower(tower): Remove a Tower from the game.
• getBases(): Return a list of all Bases in the game world.
• getBaseForTeam(team): Return the Base for the given team. We assume each
team has a single Base.
• getEnemyBases(myteam): Return a list of all bases that are not a member of the
given team.
• getTowers(): Return a list of all Towers in the game world.
• getTowersForTeam(team): Return a list of all Towers that are part of the given
team.
• getEnemyTowers(myteam): Return a list of all Towers that are not a member of
the given team.
• getNPCsForTeam(team): Return a list of all NPCs (AI-controlled Agents) that are
members of the given team.
• getEnemyNPCs(myteam): Return a list of all NPCs (AI-controlled Agents) that
are not members of the given team.
• damageCaused(damager, damagee, amount): Whenever one Agent—the
damager—causes damage to another Agent—the damagee—then the
corresponding number of points are added to the damager’s score.
• addToScore(team, amount): Add a given number of points to the score of the
given team.
• getScore(team): Return the number of points awarded to the given team.

Agent Behavior Controllers

Agent’s are, by default, not very intelligent. They need something to tell them what to do
and where to go. There are two means of controlling Agent behavior in the game
engine: state machines and behavior trees. StateAgents have StateMachines built into
them. MyHeroes have BehaviorTree controllers built into them. In both cases, Agents “outsources” the decision making to an object that contains behavior control logic. These decision-making objects make callbacks to the Agent to tell it where to move, to shoot, etc.

State Machines

StateMachine

A StateMachine is a model of Agent behavior where different behaviors manifest based on which state the Agent is in. Certain conditions cause the Agent to transition from one state to another. The StateMachine class doesn’t do much except remember what state it is in and provide functionality for changing the state. Each state the machine can be in has a corresponding State object that can run code every tick and make callbacks to the Agent.

Member variables:

- state: The current state of the machine. state is a reference to the State object that is currently executing.
- states: A list of states that the machine knows about. Each element in the list is the name of a class that sub-classes from State.

Member functions:

- changeState(newStateClass, *args): Causes the machine to change to the given state. newStateClass is the name of a class that sub-classes from State. When the state changes, a new object of the given type is created and becomes the active state and starts executing every tick. changeState() can take an arbitrary number of additional arguments, which are collected up and passed into the new State constructor.
- getState(): Return the name of the class of the currently executing state.

State

A State contains a chunk of code that controls the behavior of an Agent via callbacks. Every tick, its execute() function is called. State is an abstract base class.

Member variables:

- agent: A reference to the Agent that this State is currently controlling.
- execute(delta): If the State is the current state, then this function is called every tick. delta is the amount of time since the last tick. By default this function doesn’t do anything and must be overridden.
- enter(oldState): This function is called when a State first starts executing. oldState is the name of the class of the State that was executing just prior to this State taking over control of the Agent. Override this function for any one-time setup of the State.
- exit(): This function is called when the state machine has transitioned to a new state and the current State is no longer going to be running. Override this function to handle any one-time clean up after the State has completed execution.
• parseArgs(args): This function is called by the constructor, which may have gotten an arbitrary number of parameters passed to it. args is a list of parameters. By default, this function doesn’t do anything and should be overwritten if the State can be parameterized upon construction (i.e., when it becomes the current state controlling the Agent).

Example:
Consider the example of an Agent that also inherits from StateMachine. MyAgent knows about two custom States, State1 and State2, and defaults to State1 upon execution. MyAgent has two custom callback methods.

class MyAgent (Agent, StateMachine):
    def __init__(self, image, position, orientation, speed, world):
        Agent.__init__(self, image, position, orientation, speed, world)
        StateMachine.__init__(self, [State1, State2])

    def start (self):
        Agent.start(self)
        self.changeState(State1)

    def callback1 (self):
        print "callback1"

    def callback2 (self, x, y):
        print "callback2", x, y

class State1 (State):
    def execute(self, delta = 0):
        self.agent.callback1()
        if <some condition>:
            self.agent.changeState(State2, arg1, arg2)

class State2 (State):
    def parseArgs (self, args):
        if len(args) >= 2:
            self.arg1 = args[0]
            self.arg2 = args[1]

    def execute (self, delta = 0):
        self.agent.callback2(self.arg1, self.arg2)

When MyAgent starts running, it immediately changes state to State1. This cause State1 to be created and its enter() method to be called. When MyAgent receives its first update from the game engine, it asks State1 what it should do, by calling its execute() method. State1 calls back to the MyAgent with callback1().

If a condition triggers as true, State1 also causes the agent to change state to State2 with two additional arguments. Note that execute() should not do any more work, since the state change will cause State1’s exit() method to be called. State2 is created, State2 interprets the two arguments passed in with its parseArgs() method, and its enter() method is called. When MyAgent receives the next update from the game engine on the next tick, State2 receives the execute() call, which calls back to the MyAgent with callback2.

The flow of function calls is illustrated below:
Behavior Trees

BehaviorTree
A behavior tree is a specification for plan-like branching behaviors for an agent. The BehaviorTree class is a container for the behavior tree, which is a tree of BTNodes. BehaviorTrees know how to build the tree from a specification language and make sure that the root of the tree receives updates every tick. The purpose of the behavior tree is to figure out which BTNode should control the Agent in this tick and for that BTNode to make the appropriate callbacks to the Agent to control its behavior. Unlike a state machine, all nodes are instantiated when the tree is built.

Member variables:
- tree: A pointer to the root BTNode of the behavior tree.
- running: A Boolean indicating whether the tree should receive updates from the game engine.

Member functions:
- buildTree(spec): Creates the structure of the behavior tree, instantiating all nodes. A spec is a tuple or list that contains instructions for which types of nodes to create in the tree and how the tree is structured (i.e., parent-child relationships).
• setTree(root): If a behavior tree has already been pre-constructed, this sets the tree pointer to the given root node.
• printTree(): Print the nodes of the tree to the terminal in depth-first order.
• update(delta): Called every tick. This function calls the execute() function on the root of the tree, which then passes down the execute() calls to the appropriate node to control the Agent.

Tree specification language:
The specification passed into buildTree() tells the BehaviorTree the type of each node in the tree, the child/parent relationships between each node, and any parameters that can be known at build time. The build specification language is a sub-set of the Python language and is as follows:
• A List indicates a child-parent relationship such that the first element of the list is a parent and each subsequent element (2 through N) in the list is a child of the first element.
• If the first element in a List is a class name (a BTNode or a sub-class of a BTNode), then a node of that class type is instantiated.
• If the first element in a List is a Tuple, then the first element of the Tuple must be a class name (a BTNode or a sub-class of a BTNode) and all subsequent elements in the Tuple are parameters to be passed into the node's constructor.
• List elements that are not first in the List can be a class name, a Tuple, or a List. Class names and Tuples indicate leaf node children. A List indicates that a new child node should be created (see above) and that this child in turn has its own children.

For example:
[[Sequence, 1], [[Sequence, 2], (BTNode, 3), (BTNode, 4)], [[Selector, 5], [[Sequence, 6], (BTNode, 7), (BTNode, 8)], (BTNode, 9), (BTNode, 10), BTNode]]
creates the following behavior tree. Arrows denote Sequence nodes and question-marks denote Selector nodes. (Note that one node does not have any parameters.)
how the behavior tree works. One of the responsibilities of a behavior tree is to figure out at every tick which BTNode should be in control of the Agent. There are generally two different types of BTNodes: internal control flow nodes and leaf nodes. Internal control flow nodes determine which leaf nodes receive the final execute() call. Leaf nodes implement the behavior of the Agent through callbacks. A BTNode’s execute() function can return one of three values: True, False, or None. A True response means that the node has successfully completed execution. A False response means that the node has failed to complete its execution or that the node is not applicable to execute in the current world situation. A None response means that the node is executing but will require more ticks to complete execution and thus cannot determine success or failure at this time. The first time a node’s execute() function is called, the node’s enter() function is also called. A behavior tree can be “reset” so that all nodes think they have never been called before.

Member variables:

- **agent**: A reference to the Agent that can be controlled by this node.
- **children**: A list of pointers to objects that are BTNodes or sub-classes of BTNode.
- **current**: The index (position in the list of children) of the BTNode in children that is currently being executed.
- **first**: A Boolean indicating that this is the first time this node has received an execute() call.
- **id**: The unique identifier of the node.

Member functions:

- **parseArgs(args)**: The constructor to a BTNode can receive a number of arguments in a list. These arguments are passed to parseArgs, which must figure out what to do with them. The default behavior is to set id to the list of args.
- **execute(delta)**: Perform a behavior or determine which child should execute. Returns True if the node has successfully completed execution, False if the node is not applicable at this time due to world conditions or has failed to execute successfully, or None if the node should continue to execute next tick. If this is the first time execute() has been called on this node, then enter() is called. The default behavior is to do nothing. Sub-classes should override this function, but always call the base class version immediately.
- **enter()**: Called the first time a node has been asked to execute. The default behavior is to do nothing. Sub-classes should override this function, but always call the base class version immediately.
- **printTree()**: Prints out the node’s id and then calls printTree() on children in depth-first order.
- **reset()**: Restore the node to its original configuration for a new run.
- **setID(id)**: Set the node’s id.
- **getID()**: Return the node’s id.
- **getAgent()**: Return the reference to the Agent this node can control.
- **setAgent()**: Set the Agent that can be controlled via callbacks.
- **getChild(index)**: Return the child node at the given index.
• `addChild(child)`: Add a child to the end of the list of children.
• `getChildren()`: Return the list of children BTNodes.
• `getNumChildren()`: Return the number of children.
• `getCurrentIndex()`: Return the index of the child BTNode that is executing.
• `setCurrentIndex()`: Set the index of the child node that is executing.

Selector
A Selector is a type of BTNode that controls the internal execution flow of a behavior tree. A Selector iterates through its list of children, searching for the first one to return True from its `execute()` function. This class must be completed as part of Homework 6.

Sequence
A Sequence is a type of BTNode that controls the internal execution flow of a behavior tree. A Sequence iterates through each of its children, calling their `execute()` functions. Iteration stops when all children have executed successfully or one of the fails. This class must be completed as part of Homework 6.

Binary Space Partitions
Binary space partitions make the spatial reasoning aspects of the game engine more efficient. Both collision-free navigation and decision-making make heavy use of ray casts. A binary space partition breaks up the environment into small regions so as to determine if there are elements that cannot be intersected by a ray during a ray cast. The regions are arranged in a binary search tree to quickly search for regions relevant to a ray. The overhead cost of the binary search is mitigated by the reduction of the cost of comparing a ray to all possible objects in the world.

BSPGameWorld
A BSPGameWorld is a specialization of GameWorld. The only difference is that BSPGameWorld maintains a pointer to the root of a binary search tree of space partitions. With a BSPGameWorld, one can use `BSPGameWorld.bspRayTraceWorld()` instead of the `rayTraceWorld()` utility function for more efficient ray casting.

If the user presses the ‘p’ key, the game world will draw a border around the leaf space partition that contains the mouse cursor. If the user presses the ‘r’ key, the game world will draw a ray from the player’s avatar and the mouse cursor and highlight the lines of Obstacles that are contained in the same space partitions that the ray also passes through. This highlighted set of lines is the complete set of lines that the ray had to be checked against for intersections.
Member variables:

- bspTree: A pointer to the root of the binary search tree of space partitions.

Member functions:

- buildBSPTree(): Call this to construct the BSP tree.
- drawBSPTree(): Call this to outline the space partition regions.
- bspRayTraceWorld(p1, p2): Returns the (x, y) point at which a ray from p1 (x1, y1) to p2 (x2, y2) intersects with a line of an Obstacle. This function returns None if there are no intersections.
- drawPossibleIntersectionLines(p1, p2): Highlights the lines of Obstacles that pass through leaf space partition regions that the ray between p1 and p2 also passes through. p1 and p2 are points of the form (x, y).
- drawBSPLeafForPoint(pos): Draws the border around a space partition region that encloses the given (x, y) point.

**BSPNode**

A BSPNode is a node in a binary search tree where every node accounts for a rectangular region of the game world map. Node children represent sub-regions. For example, the root node of a binary space partition tree would represent the entire map, and the two children of the root would each account for half of the map. When a ray is
cast, the binary search tree is searched for the set of leaf BSPNodes that account for regions of the map through which the ray passes. BSPNodes also remember which lines from Obstacles pass through the regions. When the BSPGameWorld needs to determine which Obstacle lines might be intersected by the ray, the binary search tree can efficiently eliminate lines that are guaranteed never to intersect with the ray. Some parts of this class must be completed as part of the **optional BSP Homework**.

Member variables:

- children: A list of children BSPNodes. There should never be more than two children in the list.
- box: A pygame.Rect object that represents the rectangular area of the map that this node is responsible for.
- lines: The list of lines from Obstacles that pass through the rectangular area that this node is responsible for.
- id: An unique identifier for this node.

Member functions:

- **setLines(lines)**: Tells the BSPNode which Obstacle lines it needs to track. This function takes in a list of lines where a line is a tuple of the form \([(x_1, y_1), (x_2, y_2)]\).
- **addLine(line)**: Adds a line from an Obstacle to the node. A line is a tuple of the form \([(x_1, y_1), (x_2, y_2)]\).
- **buildTree(lines)**: A recursive function that creates the sub-trees of the binary space partition tree. The given lines are those that pass through the rectangular area of the node, some of which will be passed down to children nodes that are created. Thus the root of the tree receives all lines from all Obstacles. This function takes in a list of lines where a line is a tuple of the form \([(x_1, y_1), (x_2, y_2)]\).
- **drawTree(surface)**: Draws the node and recursively asks its children to draw.
- **findLeafForPoint(point)**: Return the leaf node that contains the given point. If this node is a leaf that contains the point, it returns itself. Otherwise, it recursively calls the function on the node’s children. A point is a tuple of the form \((x, y)\).
- **findPossibleIntersectionLines(line)**: Return a list of all Obstacle lines that pass through leaf space partition regions through which the given line also passes. A line is a tuple of the form \([(x_1, y_1), (x_2, y_2)]\).