1. Introduction

1.1 Overview
The premise behind Robot Arena is that of a 1 vs. 1 fighting game, but unlike most such games, the player’s role in Robot Arena is more strategic than tactical. The inspiration for Robot Arena was the Robot Wars T.V. series, which involved teams of amateur and professional “roboteers” who built real life robots that then fought against each other in televised matches. By bringing this type of gameplay to the digital medium, the hope is that more variation can be realized, by a larger number of people, and at a cheaper cost.

While games involving the player as the designer of his own battle avatar are plentiful, games where the player is also in charge of designing the AI of this battle avatar are less common. Most games place the battle directly under the player’s control, and Robot Arena is unique in that it both avoids this temptation and also utilizes a sophisticated system by which customized AIs are not only designed by the player, but are integrated into the game play itself. This integration is achieved by a resource system, in which the player is charged for both AI components and equipment for the robot, leading to a design trade-off between complex AI and raw equipment power - “brains” vs. “brawn.”

The ultimate goal for Robot Arena is to achieve a fine balance between AI components and equipment power, as well as between the difference equipment components themselves, so as to create a deep strategic game in which it is not immediately obvious what the optimal designs are. Indeed, the ideal is for there to be no globally optimal design, so that players have to tweak their robots specifically for their opponents, the map selections, and the game modes, leading to back-and-forth interactions of evolving complexity.

1.2 Game Play Format
Robot Arena is a two-player game meant to be played hot seat or over the internet. The game play format involves a robot design phase and an actual battle phase. The players are in charge
of the design phase and are observers during the battle phase, making it quite unlike most other
games in which the players are in charge of the battles directly.

**Design phase:** Players take turns during this phase to design their robots. They select from a
list of weapons, items, upgrades, and AI components, each with an associated cost, constrained
by a pool of money that they begin with. Depending on the game play mode used, the player
may start with more or less money. The players are not allowed to see what the other player’s
design is, so as to preclude direct counter-play. Map previews are also given during this phase,
so as to give players a chance to prepare their strategies and to identify strategic points on the
map.

**Battle phase:** Once both players have finished designing their robots, the game enters into the
battle phase. During this phase, the two robots face off against one another, using weapons and
items, moving to locations, and generally acting according to their AI. The last robot standing
wins the battle. There is a timer for situations when a draw appears likely, which can be set in
the options of the game.

**Game Play Modes:** Game play modes that make sense for Robot Arena include standard Best-
Of-X tournaments, in which the players undergo a series of design-battle phases, one after
another, on either the same map or a set of maps, King of The Hill, in which several players
compete with one another to stay undefeated for as long as possible, and other modes that
make sense for a 1 vs. 1 strategy game.

Though not currently planned, a single-player mode involving a human player playing against a
computer robot designer is also possible, and may initially be available simply as a pool of pre-
designed robots that a human player can try his own robots against. Such a pool of pre-
designed robots can also be linked through a Campaign mode, in which the human player
battles against a series of increasingly powerful pre-designed robots, gaining additional money
for each robot his own robot defeats.

The amount of money available is generally preset before the game as an options variable.
Winning a game could also yield additional money in certain game modes, such as King of The
Hill and Campaign mode.

2. Engine

2.1 Unity and Our System

Unity is a Game Development Engine with an integrated graphical environment as the primary
method of development. Unity has integration with many 3D modeling applications such as
Maya, 3DS Max, Blender and others, supports nVidia PhysX and scripting via C# or javascript.
We took advantage of the modular (Component based) architecture of Unity in our game; since our game is based in assets and different type of weapons a Robot can have; therefore, we scripted these assets as independent and most of the time self-contained C# Scripts.

The Unity authoring tool has four main UI modules for developing which is the Scene, Project, Hierarchy and Inspector.

**Scene:** here we construct our game level, that is, we add our 3D models such as the Arena, Walls and the Robots. The Scene allows us to have a visual representation of what the game level look like at runtime.

**Project:** here we store all of our assets such as 3D models, scripts, audio and textures. This Project place is only for storing, here none of the objects are still active during gameplay.

**Hierarchy:** every asset we add into the Scene well be listed in this UI module where we can create parent-child relationships as a Tree structure.

**Inspector:** This is one of the most important UI modules, here we create connections between the Objects in the hierarchy and Scripts and other components such as Rigid Bodies, Collisions, and Transformation (position, rotation...). Therefore, when we attach a script to our Robot will see the script's information in this UI module.

### 2.2 The Scripts

We chose C# as our language due to the similarity to Java and C++. In Unity a C# script derives from MonoBehaviour, that means a script can be attached at authoring time to objects in the scene, but we can also have pure C# classes which can be instantiated at runtime. The MonoBehaviour scripts have some system methods which are very useful, such as Start, Awake and Update. Awake and Start are for initialization and and executed only once, while Update is executed at every tick.

### 2.3 Why Unity

When we envisioned the idea of having some Robots in a battle, we thought that adding the detail of 3D along with nice animations and particles will add this extra dimension to the user experience, and the user will have a much clearer idea of what they are, what they are doing and be more appealed by the game.

#### 2.3.1 Pros and Cons

**Pros:**
- Graphical Authoring tool
- Handling for Graphics
- Physics engine availability
- C# for scripting
- Game oriented API
2.4 Pathfinding

Given the continuous space of the game and the extra dimension of 3D, we decided to interface with an available API for pathfinding based in NavMesh. First we chose what object will be used to scan the environment, in our case we chose a Capsule because our Robots are based in a Capsule Collider, the capsule's radius was very important and it solved one of our previous issues of the Robot hitting/tripping with the wall's corners; so setting a capsules radius much larger than the Robot's fixed that problem. Then we choose the nodes size for the grid (NavMesh) and the number of them, this will give the final density of our grid (NavMesh).

We have now the NavMesh for navigation, but how to use it? The main function for the Robot to move around the terrain is the “MoveTo(Vector3 destination)”, that is, the Controller script will provide a new destination to the Robot, and then MoveTo will make a call to create a new path from its current position to the destination, this search process is based on the A* algorithm, when the path is computed, the Robot will have a callback function that will receive an array of Vector3 positions, and here is the part that gets complicated, for performance and functionality issues you do not want to just move to the 1st point and create another path at every frame, but also is not possible to navigate all these points, otherwise you will move to the destination extremely fast. Other two functions will play an important role in solving this caching and selection of where to move to; then there is the FindPoint( ) function which is called at every 0.2 seconds, what this function does is to calculate how much the Robot has moved and select a new destination (WayPoint) from the cached path, then a state machine for the robot will be in charged in calling a MoveTowards() function to move to the selected waypoint.

We found the pathfinding to be very fast, and moreover it has a path smooth functionality.

3. Assets

3.1 Robot model & animation

Our Robot is based on a 3D model with IK handles (bones) for character animation, it has a total of 7 animations. As a collider the Robot is based on a Capsule collider along with a Character Controller which is similar to a Rigid Body but specialized for characters. Out Robot agent is controlled by 3 high level States Idle, Moving and Attack, and these states control the actual animation of
the 3D model which depend on other states, for example, for the Moving stated we can have Walking and Running which is based on the actual speed of the agent, and for Attack with have other animations depending on the actual current weapon. The agent will always be moving across the path given by the Pathfinding. Our agent stated are in the Robot script, while we have some of the animation controls on a separate script RobotAnimations that constantly is checking for the speed of the robot if it is in Moving state, or stops other animations before it plays the animation for the current state.

3.2 Items model & animation

Combat being the nature of our game, weapons are an integral part. Weapons are an important factor of our game design. A lot of quality time was spent on discussing the characteristics of different weapons. We decided that each weapon should have its own usage in a particular situation and also a cost assigned to it. The weapons have been designed keeping in view various situations they can be used in by the robots and each one serves a different purpose. Weapons serve the purpose of damaging the opponent. We also decided to include some assets from the point of view of protection and healing oneself. This was done in order to have a good game design and game balance. The following are the different assets that our game has:

1. **Blaster Bullets:**

   This weapon costs 20$. Its a weapon with an infinite range. This was designed keeping in view, situations where the enemy is sighted far away. Since this weapon has an infinite range we decided to have a low damage assigned to it. This weapon decreases the health points by 5.
2. **Grenades:**

Grenades cost 55$. They have a medium range and can be used in situations where the enemy is hiding behind an obstacle, like wall. They reduce the health points by 20. But to make it look more realistic the damage is inversely proportional to the distance, i.e. closer the robot is to a grenade more the damage and vice versa.

3. **Power Of Fist:**

The idea for this weapon really required some thinking to come upon. We decided that if a robot finds himself in melee range of his enemy he should be able to exploit that situation with a suitable weapon. This weapon can be used only when really close to an enemy. Since being close to an enemy is a precarious situation we decided to give this weapon the highest damage. It reduces the health by 30. Also, it costs a whooping 75$.

4. **Shield:**

A shield can be used to protect oneself from damage. A shield is a weapon which provides permanent invulnerability. Hence for game balance and fairness we decided to have a limitation on the time for which the shield is active. The shield lasts for 5 seconds per activation. It costs 30$.

5. **Stealth:**

The stealth mode makes a robot invisible to his enemy. We have a cool down time set so it cannot be activated continuously. This is also a weapon from point of view protection. It costs 50$.

6. **Speed Up:**
This asset increases the speed temporarily. Can be used to flea from or chase the enemy. It costs 25$.

7. Repair Station: We decided that though we have assets that provide protection, none increase the health points of a robot. So we included this asset that increases the health of a robot by 50. These are already placed on fixed on locations on the map and need not be bought.

3.3 Sound

Sound and Audio effects are an important part of any game, it gives more depth and gets more attention from the user. Fortunately, adding audio in Unity was very simple since they are treated as a component that can be called from a function at anytime. We added sound effects for all the weapons we implemented and for their ammunition collisions.

3.4 Level design

The level consists of an arena on which the robots combat against each other. The arena has walls which form a maze through which the robots move. The walls act as cover behind which the robots can hide and attack. They make the level look interesting rather than having just an open arena. They help in making decisions such as when to use a blaster bullet versus when to use grenades.

The level also consists of repair stations situated at fixed locations. They help in marking some strategic locations to which the robots can decide to move in certain situations. These things help in making certain strategic choices. For example, if enemy is behind a wall a robot can move towards him till he is in the range of grenades and throw grenades at him still keeping himself concealed. If there is a wall between a robot and his enemy then he need not use the stealth mode and if the health of a robot is below a certain value he can move towards the repair stations.

Hence level design is an important factor in game design as it helps make some important strategic decisions and makes the game play and AI more interesting and realistic.
4. Game Balance

Game balance generally means that no single player has an unfair advantage above other. But game balance in this game means a balance between the weapons and AI components bought by a player. The whole point of this game is to show that a robot with only weapons and no AI or vice versa is unlikely to win the game. Its important to have a well balance between AI and weapons in order to win the game. The game balance is also in terms of the costs associated with the weapons. It should not be possible for a player to buy the strongest AI component and the strongest weapons together as then there would not be any sense to the whole game. Each level design should need the players to have a different combination of AI and weapons to succeed.

Thus, game balance is important characteristic of this game as most of the game play depends on how well the game is balanced in terms of the various costs of the weapons and AI components.

5. AI

5.1 Design Process

The general design philosophy for AI in Robot Arena is that it should be an integral part of the game, and that there should be a trade-off between “brain” and “brawn” in the design phase for robots. To achieve this, we initially planned a system in which different algorithms, enabling different aspects of intelligent behavior, would be available to the player for purchase as AI components, in the same way different weapons and items, enabling different tools for the robot to use, are. Thus, a player might be able to purchase an AI component like A* (renamed to some appropriate futuristic technology) that would grant their robot the ability to quickly plot paths around walls and obstacles, and this would come at the cost of being able to purchase less weapons and items for their robot due to a diminished money pool.

However, this design was deemed to be poor for several reasons. First, just because the player purchased an algorithm does not mean that his robot would be granted an additional behavior, since, for example, A* does not by itself say how to play the game (it requires a heuristic). Thus, there had to be another mechanism for AI design that would govern the actual strategy of the robot. This made AI design an arduous and complicated process. Second, it was not immediately obvious that different algorithmic choices would lead to clear behavioral differences that can be easily observed by the player. For example, BFS and A* plot similar paths through a map when used for finding shortest paths.

To simplify the design process, we therefore scrapped the idea of having players choose between different AI algorithms. Instead, we adopted the methodology of players explicitly defining the in-game behavior of the robot by building behavioral graphs similar to FSMs. The AI-equipment trade-off would now be over the complexity of the behavioral graphs defined, as
more complex graphs required more nodes, and such nodes would have to be purchased by the player.

5.2 Action-Condition Graph

The methodology we settled on for Robot Arena is a simplified variation on the FSM, called an action-condition graph. In an action-condition graph (ACG), the nodes are either action states or transition conditions, and the vertices are the directed links that represent transitions between them. Thus, an ACG is a directed graph, and cycles are to be expected within this graph because presumably the robot would be repeating many actions during the course of a game. A graphic depiction of the components of an ACG can be found below:

![Action-Condition Graph](image)

Note the green and red arrows. Green arrows are “true” transitions while red arrows are “false” transitions. Actions are defined to evaluate automatically to true, and so they can only transition through green arrows and are only allowed one of these. Conditions, meanwhile, can evaluate to true or false for the current game state, and have different transitions based on this evaluation. They are still limited to only one true transition and one false transition, however, and are thus binary conditions.

Action nodes can be chained one after another, in which case they are to be executed in order, one per time step. Conditions can also be chained, but the evaluation of the conditions in a chain will occur during the same time step the condition at the head of the chain is evaluated. In this way, logical combinations like AND and OR can be easily constructed simply by chaining together conditions. For example, a condition A followed by a condition B, with A’s true transition linking to B, B’s true condition linking to the action to be executed, and both conditions’ false transitions linking to the previous action, constitute an AND condition that would require both A and B to be true before the robot would move to the next action.
5.3 Action-Condition Graph Execution

Executing an action-condition graph is equivalent to traversing it, starting from the root. Action and condition nodes are treated differently, however, and the flow of execution assumes that an action node is executed each time step, while its associated conditions (if any) are evaluated immediately afterwards to determine the action node during the next time step. The general algorithm for traversing an ACG can be described as follows:

current node <- root(ACG)
For each time step t
    If robot is not dead, execute action(current node)
    If current node.next is an action node
        Set current node <- current node.next
    Else if current node.next is a condition node
        Traverse current node.next recursively* until the next action node is reached
        Set current node <- the next action node reached

(Traverse = follow the true or false transition according to what the condition node evaluates to in the current game state)

As implied above, each time step an action would be sent to the robot to be executed, which could simply be an idle action if no other action is desired. In practice, ACGs tend to continuously execute an action or action chain until some condition or set of conditions are reached, at which point it would transition to another action and its associated condition(s).

5.4 Action-Condition Example

A more concrete example of an action-condition graph will now be given. The following ACG implements one of the robots we have implemented in the example games. Specifically, it is the blue robot in game1.scene:
Moving through the graph, the robot would first execute the action MoveToEnemy, which simply moves it towards the enemy robot’s location via our pathfinding algorithm. It would then evaluate the condition EnemyInRange: Grenade, which would determine whether the enemy robot is in range of the grenade weapon. If it is, the robot would transition into the ThrowGrenade action, which would be executed until EnemyInRange: Grenade is no longer true.

If and when EnemyInRange: Grenade is no longer true, the next condition, EnemyInRange: Blaster, would be evaluated. If this condition is true, the robot would execute FireBlaster, and then subsequently evaluate EnemyInRange: Grenade, again, for a possible transition back to ThrowGrenade. This is to ensure that grenades, which do more damage, have priority over the blaster weapon.

If EnemyInRange: Blaster is not true, the robot would transition back into moving towards the enemy. Note that because EnemyInRange: Blaster is only ever transitioned to from EnemyInRange: Grenade, it will always be evaluated in a chain with the former, and so any evaluation of EnemyInRange: Blaster can assume EnemyInRange: Grenade to be false during the same time step. Thus, the final transition back to MoveToEnemy occurs when both EnemyInRange: Blaster and EnemyInRange: Grenade are false.
In short, the given example represents a robot that switches between two weapons - the grenade launcher and the blaster - depending on which is in range, prioritizing the grenade launcher when both are in range, and moving towards the enemy when neither are in range.

5.4 Interface Design

In order for the player to design his robot, there must be an interface for it. While equipment design can be achieved relatively easily with a drop-down box interface, a more convenient solution for the ACG of a robot is desirable. Our initial design consisted of a “circuit design” interface (described below in the Future Work section), but due to time and unfamiliarity with developing interfaces in Unity3D, we instead had to settle for a simpler interface.

As of right now, our interface for AI design is C# code. The example robot AI given above can be found in the Awake() method in BlueRobotGraph.cs, and is more or less a direct translation of the graph, with some code-related redundancies. Of course, such an interface cannot be easily used by players to design robot AI, so developing a graphical interface for robot AI design is a priority.

6. Issues We Faced

6.1 Learning curve for Unity

Two of three group members had never used Unity, so we have to watch tutorials, play with examples and get familiar with Unity at the beginning of the project. We constantly were learning new things, browse the API to find useful methods, and more importantly, find the correct way to do the things in Unity, performance wise- and quality wise.

6.2 Repository - SVN

In order to be able to contribute to the project remotely, we use SVN (SourceForge), and we constantly got a few problems with versions. Most of the problems were caused by the fact that some of the files were binary instead of only text based. The problem got worse on the last week to the point that we started to share the changes via email.

6.3 Initial AI Design Not Viable

As stated above, the initial design of having the player pick between different algorithms was not really viable because these algorithms governed behavior that were sufficiently subtle that it was difficult to show the player what the effects of picking one algorithm over another was. Consequently, the action-condition graph method was adopted, since not only was it more understandable for non-experts, but the effects of picking one behavior over another was easier to see.
6.4 User Interface Difficulties in Unity

According to our initial design and idea, we were planning in having user interfaces where the user would be able to buy/select the assets, and more importantly the user interface that allows the user to build the nodes for the AI. Unfortunately, we did not have too much experience in GUI in Unity therefore, we were not able to have this UI system.

7. Future Work

7.1 Graphical Interface

As stated before, the goal for the AI design phase of Robot Arena is ease-of-use. While action-condition graphs are not particularly difficult to understand, their creation present usability issues. The current interface for creating action-condition graphs are suboptimal and should be replaced with a more user friendly and streamlined interface. A “circuit design” analogy was initially envisioned for this user interface, in which the actions and conditions would be “gates” and the links between them the “wires.” The player would be given the impression that he was actually engineering the robot’s brain, and would be allowed to drag and drop components. The constraints between the components (both AI and equipment) should also be made more explicit to the player, such as by not allowing him to add certain actions and conditions involving an item when his robot does not possess said item.

7.2 Sensors & Actions

As the action-condition AI we use to construct AIs in Robot Arena are highly dependent on available sensors and actions, it is necessary to continue adding sensors and actions in order to give players a rich palette from which to build their own robot AIs. A way to streamline the generation of sensors and actions according to object affordances is thus desirable, similar to how it is done in the Sims.

7.3 AI Algorithms

The action-condition graph was adopted in place of other AI algorithms for simplicity of use, but this is not an exclusionary principle. At some point, it would be useful to add back algorithmic choices as robot component “upgrades.” What is important here is to establish meaningful choices, which would require that those algorithms we add produce definite changes in behavior. For example, the choice might be between A* and brute-force Euclidean minimization (which would get stuck behind walls), instead of between A* and BFS, although the latter could also be included if some way to incorporate time performance differences can be devised. Other examples of useful AI algorithms to add include an automatic targeting system that would be able to predict the future destination of opponent so that the robot can shoot in a more accurate direction and avoidance mechanisms like strafing that would allow the robot to dodge certain projectile weapons.
7.4 Items & Weapons

There are more items and weapons components we would have liked to implement. As for weapons we would like more variety of weapons such as missiles, weapons that affect the opponent’s sensors and visibility. One feature we wanted to have integrated is a field of view, so that robots can give the illusion of being exploring the game scene. More interesting power-up that improve/upgrade the sensors and weapons.

7.5 Maps and Map Features

A implementation of different levels is a must on any video game, this would give the illusion of variety to the user and an incentive to keep playing. In this case we would like to have level of different difficulty such as one with more obstacles, traps and different worlds so that the user find the need to update or tweak their Robot’s components and AI.

7.6 Game Modes

Additional game modes are of great help to Robot Arena since it lends itself to different trade-offs in the AI-equipment balance, as well as different strategies from the players. For example, in a King of the Hill mode, the player might be drawn to robot designs that are more general, while in a 1 vs. 1 against an individual player, the player might try to tailor his robot’s design towards that player. It would also be interesting to work in the idea of earning money from wins and gradually upgrading your robot through the course of a tournament or single-player campaign.

8. How To Run Code

8.1 Files submitted

- Entire Unity Project
- Two Windows executable files
- Readme file explaining executables
- Project document

The two Windows executable files we submitted were Game1 and Game2.

Game1 pits the blaster-grenade robot (the blue robot) mentioned above against a defensive robot (the red robot) sporting only a blaster, and which has an AI that makes it run to repair stations whenever its health reaches below a certain point.

Game2 features a more expensive and advanced version of the red robot (the red robot), which now has the ability to invoke speed ups when it runs to the repair stations, and a hunter killer robot (the blue robot) that utilizes stealth and speed up to use its powerful Power of Fist melee weapon to quickly demolish its opponent.
8.2 Where is the Source code?
In the Unity project, so to Assets folder, then Scripts. Here you will find all of our C# scripts.

8.3 How to run it
- We provide two executable files as example (Tested under Windows 7)