|  |
| --- |
| Kim-Jong Fun |
| Project Proposal |
| Super Dragonslayer Robot 9000 |

|  |
| --- |
| 11/13/2015 |

# Purpose (1 - 3 Paragraphs and/or 3 - 5 bullet points)

Super Dragon Slayer Robot 9000 (working title) will be a top-down dungeon crawler, in which you start with minimal means of fighting oncoming monsters, and you must explore rooms to find weapons and power-ups. There will be various power ups which affect your offense, your defense, or your enemies directly.

It will have typical WASD controls to move the character, and Arrow Key controls to use the weapon. Optionally, the player may control the character with an Xbox controller. The goal in regards to these controls is to make them feel as intuitive and natural as possible, to the same effect as games like Super Meat Boy or Shovel Knight, where each hit the player takes is not the control engine’s fault, but his/her own.

It will be visually similar to The Legend of Zelda (NES), with pixelated graphics (by choice, not because of a lack of talent, I swear). The animations will not be extremely complex, and most of it will be managed by key-frame animation as a side effect of drawing the game as it is running.

# Objectives

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **No.** | **Objective Name** | **In-Software Description (What and Where)** | **Owner** | **Difficulty  (1 to 10)** |
| 1 | Collision Detection | The ability to detect when a weapon or projectile collides with the player or an enemy, or if the player/enemy collides with an obstacle | Ilan Segal | 4 |
| There will be 2 kinds of collision detection in my game:   1. **Circle-to-Circle**   This Is, by far, the simpler form of collision detection that this game will feature. To detect if one circular hitbox collides with another, the program will determine two values: The distance between the centres of the two circles, and the combined length of the radii of the two circles.  The distance between the centres, *d*, will be determined using the Pythagorean theorem:  The variable *a* represents the difference in height (position along the y-axis) of the two centres, and *b* represents the difference in length (position along the x-axis).  The combined length of the radii of the circles, *rc*, will be determined using this simple expression:  The variable *r1* is the radius of the first circle, and *r2* is the radius of the second circle.  If , then the circles have collided, or are overlapping.   1. **Box-to-Box**   This form of collision detection will use the positions, widths, and heights of the boxes to determine if they are colliding or overlapping.  Because describing the actual logic would be very complicated and somewhat, the general procedure will be explained solely in words:  The algorithm finds the points on either box which are closest to each other, using the relative positions of the two boxes. Then, the two points are checked, to see if they are within the bounds of the other box. If they are, the boxes have collided or are overlapping. | | | | |
| 2 | A.I. | * Having the enemy monsters behave according to the player’s position relative to them * Giving the enemy monsters the ability to find a path to the player character | Ilan Segal | 9 |
| There will be 2 forms of artificial intelligence integrated into the game: General A.I., and Pathfinding A.I. These types have been distinguished because of differences in complexity and general purpose.   1. **General A.I.**   This form of A.I. is fairly simple: have the enemies’ behaviour depend on how the player’s character is placed, relative to them. This has 2 facets:   * Behaviour changes with proximity   Enemies will have different sequences and actions, depending on how far away from them the player is. Each type of enemy will have a threshold distance which determines which sequence they will perform.  The distance between the entities, *d*, will be determined using the Pythagorean theorem:  The variable *a* represents the difference in height (position along the y-axis) between the player and the enemy, and *b* represents the difference in length (position along the x-axis).   * Enemies can aim weapons at the player   Most or all of the enemy characters will be able to make use of some kind of projectile, so they will need to be able to aim those projectiles properly.  All projectiles will have a speed and an angle, and this A.I. allows the angle to be decided by the enemy, when it fires the projectile. This is done using trigonometry.  This angle, , will be determined by the following equations:  When an enemy projectile is fired, it will be fired at angle , so that it reaches the player if they are within range and do not move.   1. **Pathfinding A.I.**   To implement pathfinding A.I. that is efficient and quick enough to run several times each second, I will use the pathfinding algorithm known as *A\** (pronounced as “A star”). This algorithm combines the priority of finding a good path, with the need to be fast and efficient for use in things like real-time games.  It operates similarly to the *Dijkstra* algorithm, which is guaranteed to find the shortest path, except that it gets the help of a heuristic. A heuristic is simply an equation or algorithm that aids A\* in finding path options, by eliminating clearly bad solutions. (E.g. the heuristic could stop A\* from investigating a path that leads away from the goal, when better options are available. This makes A\* run significantly faster than Dijkstra, which would check for paths in all directions.)  The algorithm will work on a grid, with a Start node (node meaning a coordinate on the grid) and an End node. Each time it cycles, it will examine a set of nodes, which are the nodes closest to the current position. The node with the lowest cost is added to a path, and then the nodes around that node are examined in this way.  The cost of a node is modelled by the following function, where a node is represented with the symbol n:  f(n) is the total cost of a node. This is determined by adding up the exact distance from the node n to the Start node, and the estimated distance from n to the End node.  g(n) is the exact distance from the node n to the Start node. Each time a new node is examined to see if it will be added to the path, this distance value is calculated by seeing how many nodes are between the Start node and the node n.  A crucial factor to consider when determining this value is if diagonals must be taken to get to an examined node (i.e. if the examined node is not vertically or horizontally adjacent to the node on the “frontier” of the path). If this is so, the examined node is given a higher g(n) cost. This is because a diagonal line to a certain x-value or y-value on a graph is always longer than a straight line to that same height or length on a graph. (Think of how an hypotenuse is always longer than either of the legs of a right triangle)  h(n) is the estimated distance from the node n to the End node. This value is determined using a heuristic. It is important to find a good heuristic that balances speed with accuracy.  Because, for my game, path accuracy is not that much of a priority (most enemies will use projectiles and the player’s location will be constantly changing), I will use a simple heuristic that finds the exact diagonal distance between the node n and the End node.  That function is written like so, where the estimated distance is represented by the symbol h:  Δx is the difference in length along the x-axis between the node n and the End node.  Δy is the difference in height along the y-axis between the node n and the End node.  This heuristic is based on the Pythagorean theorem, because, to find the exact diagonal distance between two coordinates, you must imagine that the diagonal as the hypotenuse of a right triangle formed from the two coordinates.  The A\* algorithm recursively searches through the grid of nodes, until one of the nodes being examined is the End node. Once this has happened, the algorithm goes back, takes all the lowest-cost nodes which connect Start to End, and adds them to the path. Because of the way that the algorithm is structured and guided through a heuristic, it is a pathfinder well-suited for a game such as mine.  When implementing this code, I will also implement a form of “error” in the pathfinding, to make it fairer for the player. This will be done by having the enemies calculate a path to the position which the player was at a few seconds ago, once they are close enough. This will give the player more options when it comes to how they deal with enemies, making them feel more like they’re in control. | | | | |
| 3 | Complex Sound System | Sounds effects and background music, which activate and change based on in-game circumstances | Ilan Segal | 3 |
| Establishing this system is simply a matter of gathering sound and music files, and integrating them into the class which they are to be associated with.  The classes would then play/pause/stop sound effects or music in scenarios that the programmer has designed for.  For example, sound effects such as firing a weapon or picking up an item would be associated with the player’s character. Those sounds would be managed and played whenever the code for the player’s character tells it to play. Once the “Model” class (the class which hosts all of the mechanics of the game) detects that the player has activated a class in this way, it will give a command to the class, which tells it to play its music. | | | | |
| 4 | Inventory System | Giving the user the ability to manage their items, which affect the gameplay | Ilan Segal | 6 |
| There are two types of items which will affect the gameplay:   * Passive items   These items will be the more common items, and will affect the user’s experience as soon as they’re picked up. They can affect the player character’s stats (attack, speed, etc.), or the enemies themselves.  This is done by having a C# List of these Passive Items. That is the parent class, of which several items will be programmed as child classes. Each time the player enters a room, their stats and the enemies’ stats are calculated by running through this List of PassiveItems and executing the effect of the different Items within, whatever they might be.  At any time, the user can exchange these items for consumables, such as money or special projectiles. This will be done via GUI system, where the user selects an item and chooses what they want to do with it.  The player can only carry a finite number of passive items before they’re forced to sell or drop some of them to make room for others.   * Active items   These items only take effect when manually activated by the player, and have special effects on the enemies or on the players (temporary invincibility, massive damage, etc.).  Each active item takes a certain amount of time before they are useable again, and some are single-use only.  The player can only carry one of these at a time, and can exchange them just like they can with passive items. | | | | |
| 5 | Particle System | Creating graphical effects with the use of particles | Ilan Segal | 5 |
| Various enemies and items will have particle effects associated with them. For example, a fiery projectile will have special fire particles drawn as it moves.  A particle object is distinguished from similar objects such as projectiles because it does not affect the gameplay in any meaningful way. A particle object mainly for aesthetic purposes.  A particle object is not designed to be alone, but to exist along with many other particles of the same type in large groups, to give illusions of things like fire, smoke, or explosions. They usually have “lifespans,” or a set amount of time from being created until they stop being drawn or are wiped from memory. This, again can create illusions such as dissipating smoke. Some particles will be subject to factors like gravity, falling at different rates.  This will be achieved by creating a parent class, Particle, and having various child classes branching off of them (e.g. FireParticle, DustParticle, etc.). As the entity (enemy/player/projectile) is drawn to the screen, any particles associated with the entity will be drawn with it. The *Draw* method for these particles will be overridden by whatever type of particles they are. | | | | |

# Milestones

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Task #** | **Description** | **Owner** | **Due Date** | **Dependent Task #s** |
| 1 | *Obj. 1 Subtask:* Program detection system | Ilan Segal | Dec 12 |  |
| 2 | *Obj. 1 Subtask:* Design obstacles | Ilan Segal | Dec 13 |  |
| 3 | Objective 1: Collision Detection System | Ilan Segal | Dec 13 | 1, 2 |
| 4 | Design projectile system | Ilan Segal | Dec 14 |  |
| 5 | Program projectiles | Ilan Segal | Dec 17 |  |
| 6 | Integrate projectiles into player character’s code | Ilan Segal | Dec 19 | 5 |
| 7 | Room-to-room movement | Ilan Segal | Dec 21 |  |
| 8 | *Test & Integrate collision system* | Ilan Segal | Dec 22 | 3 |
| 9 | *Test & Integrate projectiles* | Ilan Segal | Dec 22 | 6 |
| 10 | *Test & Integrate room-to-room movement* | Ilan Segal | Dec 23 | 7 |
|  | Milestone 1 complete | Ilan Segal | Dec 23 | 3, 5, 6, 7, 8, 9, 10 |
| 11 | Design enemies, behaviours, and general A.I. | Ilan Segal | Dec 24 |  |
| 12 | *Obj. 2 Subtask:* Research A\* algorithm | Ilan Segal | Dec 25 |  |
| 13 | *Obj. 2 Subtask:* Program enemies | Ilan Segal | Dec 26 |  |
| 14 | *Obj. 2 Subtask:* Integrate A.I. into enemy code | Ilan Segal | Dec 28 | 14 |
| 15 | Objective 2: A.I. | Ilan Segal | Dec 28 | 13, 14, 15 |
| 16 | *Obj. 3 Subtask:* Design passive items | Ilan Segal | Dec 29 |  |
| 17 | *Obj. 3 Subtask:* Design active items | Ilan Segal | Dec 29 |  |
| 18 | *Obj. 3 Subtask:* Program passive items | Ilan Segal | Dec 29 |  |
| 19 | *Obj. 3 Subtask:* Program active items | Ilan Segal | Dec 30 |  |
| 20 | *Obj. 3 Subtask:* Design UI | Ilan Segal | Jan 1 |  |
| 21 | *Obj. 3 Subtask:* Design management system | Ilan Segal | Jan 2 |  |
| 22 | *Obj. 3 Subtask:* Program management system | Ilan Segal | Jan 4 |  |
| 23 | Objective 3: Inventory System | Ilan Segal | Jan 4 | 17, 18, 19, 20, 21, 22, 23 |
| 24 | *Test & integrate enemies* | Ilan Segal | Jan 5 | 14 |
| 25 | *Test & integrate A.I.* | Ilan Segal | Jan 5 | 16 |
| 26 | *Test & integrate items and effects* | Ilan Segal | Jan 6 | 19, 20 |
| 27 | *Test & integrate inventory management* | Ilan Segal | Jan 6 | 24 |
|  | Milestone 2 complete | Ilan Segal | Jan 6 | 12, 16, 24, 25, 26, 27, 28 |
| 28 | *Obj. 4 Subtask:* Design particle effects | Ilan Segal | Jan 7 |  |
| 29 | *Obj. 4 Subtask:* Program particle class | Ilan Segal | Jan 7 |  |
| 30 | *Obj. 4 Subtask:* Implement particle system | Ilan Segal | Jan 8 |  |
| 31 | Objective 4: Particle system | Ilan Segal | Jan 8 | 30, 31, 32 |
| 32 | Gather UI/HUD images | Ilan Segal | Jan 9 |  |
| 33 | Gather background/environment images | Ilan Segal | Jan 9 |  |
| 34 | Gather player/enemy sprites | Ilan Segal | Jan 9 |  |
| 35 | Gather projectile sprites | Ilan Segal | Jan 9 |  |
| 36 | Gather item sprites | Ilan Segal | Jan 9 |  |
| 37 | *Obj. 5 Subtask:* Gather music/sound effects | Ilan Segal | Jan 10 |  |
| 38 | *Obj. 5 Subtask:* Integrate sound system | Ilan Segal | Jan 12 | 39 |
| 39 | Objective 5: Complex sound system | Ilan Segal | Jan 12 | 41 |
| 40 | *Test & integrate particle system* | Ilan Segal | Jan 13 | 33 |
| 41 | *Test & integrate game artwork* | Ilan Segal | Jan 13 | 34, 35, 36, 37, 38 |
| 42 | *Test & integrate complex sound system* | Ilan Segal | Jan 13 | 41 |
|  | Milestone 5 complete | Ilan Segal | Jan 13 | 33, 34-38, 41, 42, 43, 44 |
| 43 | *Final testing & verification* | Ilan Segal | Jan 18 | 3, 15, 23, 31, 39 |
| 44 | **Game complete** | Ilan Segal | Jan 20 | 43 |

**Declaration:**

**I have read the statements regarding cheating in the assignment description. I affirm with my signature that I have worked out my own solution to this assignment, and the code I am handing in is my own.**

