**Names: Joshua Lattke, Rodolfo Moreno Gutierrez**

**Step 1. Problem Identification**

**Identification of needs and symptoms**

* Multiplayer: The game must support a maximum of two players.
* Presence of enemies: The game must include the presence of hostile enemies.
* Interactive Scenario: The game must be set in an interactive scenario.
* Weapon Pickup: Players must have the ability to pick up weapons scattered around the stage.
* Avatar Selection: Players must be able to choose their own avatar to play.
* Enemy Intelligence: Enemies must display intelligent movement behaviors.
* Destructible Scenario: The game scenario must be susceptible to damage or destruction.
* Life Gauge: The game must report the player's health to know when they die.

**Unambiguously concrete identification and definition of the problem**

The problem centers on creating an interactive video game that allows two players to participate in a graph-based experience with 50 edges and 50 vertices. The main goal is to implement fundamental graphical algorithms to enrich the gameplay. These algorithms include width search (BFS) and depth search (DFS) for graph exploration, determination of minimum weight paths using Dijkstra and Floyd-Warshall algorithms, as well as construction of a Minimum Overlay Tree (MST) using the Prim and Kruskal approaches.

**SOFTWARE ENGINEERING PROBLEM SPECIFICATION TABLE**

|  |  |
| --- | --- |
| CUSTOMER | Icesi |
| USER | Gamblers |
| FUNCTIONAL REQUIREMENTS | RF1: Player Selection  RF2: Weapons Pickup  RF3: Indicate Life  RF4: Presence of Enemies  RF5: Enemy Intelligence  RF6: Interactive Stage |
| CONTEXT OF THE PROBLEM | It takes the development of a video game in which it is multiplayer, the developer is allowed to create it as he wishes. |
| NON-FUNCTIONAL REQUIREMENTS | It must have 50 artist and 50 vertices. In addition, it must have 3 graph algorithms. |

|  |  |  |  |
| --- | --- | --- | --- |
| Name or identifier | RF1: Player Selection | | |
| Summary | The system must inform you how many players are going to play. | | |
| Tickets | Ticket name | Type of data | Selection or repetition condition |
| Gamblers | Int | Must enter maximum 2 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |  |
| General activities required to achieve the results | 1. Enter the game. 2. You need to inform how many players are going to play. 3. The number of players is recorded. | | |
| Outcome or post-condition | . Save Number of Players Information | | |
| Outputs | Departure Name | Type of data | Selection or repetition condition |
| ConfirmationPlayers | Boolean |  |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Name or identifier | RF2: Weapon Pickup | | |
| Summary | The player(s) must select a weapon they are on the map | | |
| Tickets | Ticket name | Type of data | Selection or repetition condition |
| Bullet | Int | Bullet Number 10 |
| Weapon | Int | Usage Time: 6 seconds |
| Pointed | String | The mouse must be moved where the user indicates |
| LeftButton | String |  |
|  |  |  |  |
| General activities required to achieve the results | 1. The player(s) must aim 2. With the | | |
| Outcome or post-condition | Eliminate Enemies | | |
| Outputs | Departure Name | Type of data | Selection or repetition condition |
| ConfirmationPlayers | Boolean |  |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Name or identifier | RF3: Indicate Life | | |
| Summary | The player or players will be able to report how much life they have left to die. | | |
| Tickets | Ticket name | Type of data | Selection or repetition condition |
| DamageReceived | Int | You must initialize at the maximum amount available |
| Recovery Time | Float |  |
|  |  |  |
|  |  |  |
|  |  |  |  |
| General activities required to achieve the results | 1. Calculate the damage done by enemies 2. Life Recovery Time 3. Life Fully Recovered | | |
| Outcome or post-condition | 1. Calculate the player's life | | |
| Outputs | Departure Name | Type of data | Selection or repetition condition |
| IndicatetheLife | Int |  |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Name or identifier | RF4: Presence of Enemies | | |
| Summary | Enemies must spawn randomly on the map, you must have a random number of enemies. | | |
| Tickets | Ticket name | Type of data | Selection or repetition condition |
| NumberEnemies | Int | You must initialize at the maximum amount available |
| AletorioMap | Random | Enemies must spawn at any point on the map |
|  |  |  |
|  |  |  |
|  |  |  |  |
| General activities required to achieve the results | 1.Se generates the number of enemies.  2.Se enemies spawn on the map | | |
| Outcome or post-condition | 1. Visualize Enemies | | |
| Outputs | Departure Name | Type of data | Selection or repetition condition |
| NumberEnemies | Int | Enemies spawn |
|  | AletorioMap | Random | Must display enemies on the map |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Name or identifier | RF5: Enemy Intelligence | | |
| Summary | Enemies must efficiently search for the player(s) | | |
| Tickets | Ticket name | Type of data | Selection or repetition condition |
| DamageReceived | Int | You must initialize at the maximum amount available |
| Recovery Time | Float |  |
|  |  |  |
|  |  |  |
|  |  |  |  |
| General activities required to achieve the results | 1. Calculate the damage done by enemies   1. Life Recovery Time 2. Life Fully Recovered | | |
| Outcome or post-condition | 1. Calculate the player's life | | |
| Outputs | Departure Name | Type of data | Selection or repetition condition |
| IndicatetheLife | Int |  |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Name or identifier | RF6: Indestructible Stage | | |
| Summary | There must be certain walls where they can be destroyed on the map in order to continue advancing on the map. | | |
| Tickets | Ticket name | Type of data | Selection or repetition condition |
| Destructible Walls |  | The coordinates must be within the map and must be accessible to the player. |
| ConjuntoCoordinates |  | The coordinates must be within the map and must be accessible to the player. |
|  |  |  |
|  |  |  |
|  |  |  |  |
| General activities required to achieve the results | 1. The player must find the walls and destroy them 2. The player must use the tool to destroy the walls | | |
| Outcome or post-condition | Destructible walls have been destroyed | | |
| Outputs | Departure Name | Type of data | Selection or repetition condition |
| MapModified |  |  |
|  | MapUpdated |  |  |
|  |  |  |  |

Step 2. Collection of Information

Different alternatives are sought in order to find a better solution to our problem. In that case, we looked for applications or systems similar to the development of our video game called skulbum in order to entertain people. The top 3 we found

**1 Call of duty Warzone:** Video game developed for cross-platform as well as on mobile, computer and consoles. This game is an action genre and is also multiplayer, which allows you to play multiple players at the same time.

**2.Gran Turismo:** video game developed for the PlayStation console is a racing genre where you can find several tracks from around the world and cars, which you can drive and they teach you how professional drivers drive on these tracks.

**3. Flappy Bird (2013):** A simple game in which the player must control a bird that flies between pipes. The game is composed of a total of 26 polygons, developed on Android.

In order to have a better development in our video game, we must know what type of structure is going to be used in order to implement the graphs properly.

**DFS Graph Tour**

The algorithm traverses the graph by saving a list of no's, in a save list. The particular thing about it is that the last node that we added to the list is the first one that we are going to see. Which tells us that we're going to walk the graph.

The algorithm consists of putting a first node in an initial list, adding the last node we added to its list and its neighbors.

It can be used in the following functionality:

Depending on the game conditions, you can adjust the enemy's objective on each DFS run. For example, in one stealth level, the enemy might search for the player stealthily, while, in another level, they might aggressively pursue them.

**Minimum Weight Roads (Dijkstra, Floyd-Warshall)**

The Floyd-Warshall algorithm is able to find the shortest path between each pair of vertices in a graph by comparing all possible paths between these pairs. This is done using an array that stores the minimum distances between vertices, and is iteratively updated.

Gradual iterations:

The algorithm iterates to calculate the shortest distance between all vertex pairs. In each iteration, a set of intermediate vertices, represented by the index 'k', is considered, and all possible paths between vertices 'i' and 'j' passing through 'k' are evaluated. The estimation of the shortest distance is gradually improved.

Recursion and definition of MinimumPath(i, j, k):

To calculate the minimum path from 'i' to 'j' through the vertex 'k', a recursive function called MinimumPath(i, j, k) is used. This function compares two options: the minimum path from 'i' to 'j' through 'k-1' (without using the vertex 'k') or the minimum path from 'i' to 'k' plus the minimum path from 'k' to 'j'. This decision is made for each value of 'k' iteratively.

The basis of the recursion is found in MinimoPath(i, j, 0), which represents the weight of the direct edge between 'i' and 'j'. As the value of 'k' increases in each iteration, the algorithm calculates longer routes and optimizes the paths.

Completion Process:

The algorithm continues to perform these comparisons and updates until each vertex has been considered as a possible intermediate point ('k' reaches 'n', where 'n' is the total number of vertices). At this point, we will have found the shortest path between all the vertex pairs using some intermediate vertex.

Advantages of the Floyd-Warshall algorithm:

Find the shortest path between all vertex pairs in a single step.

Handle graphs with negative weights as long as there are no negative cycles.

Useful for determining the existence of negative cycles in a weighted graph.

In a video game, we can apply by connecting the game's locations.

Take a tour of where the player has traveled

**Minimum Cover Tree -MST- (Prim, Kruskal)**

Prim's algorithm is for finding a minimal covering tree in a connected, undirected graph whose edges are labeled.

What the algorithm does is find a subset of artists that form a tree with all vertices, where the weight of all the edges in the minimum possible tree. If, in the case of the network, it is not connected, find the minimum covering tree for the components that do not form part of the connected network.

The algorithm performs the following steps:

1.Initialize a tree with a single, arbitrarily chosen vertex of the graph.

2. Increase the tree, on the one hand. We call the side the junction between two vertices: from the possible joins that can connect the tree to the vertices that are not yet in the tree, find the side with the shortest distance and join it to the tree.

3. Repeat step 2 (until all vertices belong to the tree)

It can be applied as follows:

1. Level Generation: An MST can be used to generate levels or scenarios in your game. For example, you can model important locations in the level as nodes and the connections between them as edges. Then, by applying an MST algorithm, you'll get a level structure that connects all the locations efficiently.
2. Road or highway design: If a game involves building roads, trails, or routes, you can use an MST to design efficient routes between important locations in the game, such as cities or points of interest. The MST could represent the necessary network of roads or trails.

Use the kruskal algorithm to find the short path between the rooms and make this the path of the game.

**Step 3. Search for Creative Solutions**

As we have stated before, in the collection of information we can obtain different alternatives, which are the following:

Alternative 1: Call of duty Warzone.

Call of Duty: Warzone is a free-to-play first-person shooter video game released on March 10, 2020 for PlayStation 4, PlayStation 5, Xbox One, Xbox Series X|S, and Microsoft Windows



Alternative 2 Gran Turismo.

Gran Turismo is a Japanese racing simulation video game series developed by Polyphony Digital. This video game saga has been developed exclusively for PlayStation consoles.

Image containing table, phone

Auto-generated description

Alternative 3: Flappy Bird (2013)

Flappy Bird was a mobile game developed in Hanoi by Vietnamese developer Nguyen Hà Đông and published by . GEARS Studios, a small independent game developer, also based in Vietnam

Image containing Text

Auto-generated description

Alternative 4: Skulbum developed in java.

A video game developed with javaFx in java language, in which it can be played with 2 players, aims to eliminate all the enemies that are found and the way to eliminate it is with power-ups and bombs

**Step 4. Transitioning from Ideas to Preliminary Designs**

We discarded alternatives 1 and 2 because we have the ability to program games on multiplatform or on PlayStation console, alternative 3 is also rejected for a similar reason and that is that we do not have the ability to program in Kotlin because of this alternative 4 is selected, because of the development in java and the capacity we have to do our project.

**Alternative 4. Skulbum development done in java**

-For the approach of our project we must use the java language with data structure already mentioned above in the information gathering part.

**Step 5. Evaluation and Selection of the Best Solution**

**System Flow Diagram:**

* Criterion A. Aesthetics. The solution is pleasing to the user's eye:
  + [2] It has a good design.
  + [1] It's not pleasant.
* Criterion B. Completeness. It complies with the required functionalities.
  + [3] It fulfills all the functionalities that users need.
  + [2] It fulfills some, but not all, of the required functionalities.
  + [1] It does not meet the requested functionalities.
* Criterion C. Accuracy. The solution fulfills the functionalities using the relevant data structures.
  + [3] It uses all relevant data structures.
  + [2] It uses some, but not all, of them.
  + [1] It does not employ the relevant structures.
* Criterion D. Portability. The system can be used on a variety of devices.
  + [2] It can be accessed from many devices.
  + [1] It is very limited and can be accessed from few devices.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Criterion A | Criterion B | Criterion C | Criterion D | Total |
| Alternative 4 Development of Skulbum | 2 | 3 | 3 | 1 | 9 |

The best solution was Skulbum's Alternative 4

**Defining the TAD Tables Structures to Use**

|  |
| --- |
| TAD Graph |
| < abstract object > G = (V, E)  • Note that "V" is a set of vertices.  • "E" is a set of edges.  • E ⊆ (V x V), which are ordered pairs of vertices. |
| { inv: < ∀ (vi, vj) ∈ E, (vj, vi) ∈ E >} |
| Primitive operations :  Constructor:  createGraph → Graph  Modifiers:  addVertex Graph X Element → Graph  addEdge Graph X Element X Element → Graph  removeVertex Graph X Element → Graph  removeEdge Graph X Element X Element → Graph |

|  |
| --- |
| **createGraph** |
| **creates an empty Graph { pre: TRUE } { post: Graph = { nill } }** |
|  |

|  |
| --- |
| **addVertex** |
| **adds a vertex to the graph { pre: Graph ( G ) } { post: element ∈ Graph ( G ) }** |

|  |
| --- |
| **addEdge** |
| **Given a graph, add a relationship between two elements (vertices ) of that graph. { pre: Graph( G ) ≠ ∅ ∧ V 1 ∈ V vertices ∧ V 2 ∈ V vertices } { post: (V1, , V2) ∈ E(edges) ∨ (V2, , V1,) ∈ E(edges)}** |

|  |
| --- |
| **removeVertex** |
| **Removes a given element from the graph { pre: v ∈ V ∧ V ≠ ∅ } { post: v ∉ V ∧ v ∉ G }** |

|  |
| --- |
| **removeEdge** |
| **Eliminate a relationship between two elements (vertices) of the graph. { pre: (V1, , V2) ∈ E(edges) ∨ (V2, , V1,) ∈ E(edges) } { post: (V1, , V2) ∉ E(edges) ∧ (V2, , V1,) ∉ E (edges) }** |

|  |
| --- |
| **isContained** |
| **checks whether an element is found or not in the graph { pre: G ≠ ∅} { post: True: if the element ∈ G ∨ False: if the element ∉ G }** |

**Pseudocode:**

**DFS**

class MatrixAdjacency {

boolean[][] matrixAdyacency

int numedges

ArrayList<T> vertex

MatrixAdyacency(int numedges) {

this.numedges = numedges

this.arrazAdyacency = new boolean[numedges][numedges]

this.vertex = new ArrayList<>()

}

addVertex(T vertex) {

try {

vertex.add(vertex)

} catch (Exception e) {

print("It's full")

}

}

addEdge(int origin, int destiny) {

if (Origin >= 0 & Origin < Numedges && Destiny >= 0 && Destiny < Numedges) {

matrixAdyacency[origin][destiny] = true

} else {

print("Vertex indexes are invalid.")

}

}

showMatrix() {

String msg = ""

for (int i = 0; i < numedges; i++) {

for (int j = 0; j < numedges; j++) {

msg += (matrixAdyacency[i][j] ? "1 " : "0 ")

}

msg += "\n"

}

return msg

}

toString() {

String msg = ""

for (int i=0; i < vertex.size(); i++) {

msg += (i+1) + ": " + vertex.get(i) + "\n"

}

return msg

}

seeConex(start, destiny) {

boolean[] visited = new boolean[numedges]

DFS(start, visited)

return visited[destiny]

}

hardlyConnect() {

for (int i = 0; i < vertex.size(); i++) {

for (int j = 0; j < vertex.size(); j++) {

if (i != j) {

if (!seeConex(i, j)) {

return false

}

}

}

}

Return True

}

DFS(Vertex, Visited) {

visited[vertex] = true

for (int i = 0; i < numedges; i++) {

if (matrixAdyacency[vertex][i] & !visited[i]) {

DFS(I, visited)

}

}

}

}

**Prim**

class PrimAlgorithm<T> {

Map<T, Integer> indexes

List<Edge<T>>[] adjacencyList

PrimAlgorithm(int vertices) {

indexes = new HashMap<>()

adjacencyList = new ArrayList[vertices]

for (int i = 0; i < vertices; i++) {

adjacencyList[i] = new ArrayList<>()

}

}

addEdge(vertex1, vertex2, weight) {

int index1 = getIndex(vertex1)

int index2 = getIndex(vertex2)

adjacencyList[index1].add(new Edge<>(vertex1, vertex2, weight))

adjacencyList[index2].add(new Edge<>(vertex2, vertex1, weight))

}

minimumSpanningTree() {

List<Edge<T>> result = new ArrayList<>()

boolean[] visited = new boolean[indexes.size()]

PriorityQueue<Edge<T>> pq = new PriorityQueue<>(Comparator.comparingInt(e -> e.getWeight()))

pq.add(new Edge<>(null, indexes.keySet().iterator().next(), 0))

while (!pq.isEmpty()) {

Edge<T> edge = pq.poll()

T currentVertex = edge.getVertex2()

if (visited[getIndex(currentVertex)]) continue

visited[getIndex(currentVertex)] = true

result.add(edge)

for (Edge<T> neighbor : adjacencyList[getIndex(currentVertex)]) {

if (!visited[getIndex(neighbor.getVertex2())]) {

pq.add(neighbor)

}

}

}

Return Result

}

getIndex(vertex) {

if (!indexes.containsKey(vertex)) {

int index = indexes.size()

indexes.put(vertex, index)

}

return indexes.get(vertex)

}

}

**Kruskal**

class KruskalAlgorithm<T> {

Map<T, Integer> indexes

List<Edge<T>> edges

int[] parent

KruskalAlgorithm(List<Edge<T>> edges) {

this.edges = edges

this.indexes = new HashMap<>()

}

minimumSpanningTree() {

List<Edge<T>> result = new ArrayList<>()

sort(edges)

parent = new int[indexes.size()]

for (int i = 0; i < indexes.size(); i++) {

parent[i] = i

}

for (Edge<T> edge : edges) {

int index1 = getIndex(edge.getVertex1())

int index2 = getIndex(edge.getVertex2())

if (find(index1) != find(index2)) {

result.add(edge)

union(index1, index2)

}

}

Return Result

}

getIndex(vertex) {

if (!indexes.containsKey(vertex)) {

int index = indexes.size()

indexes.put(vertex, index)

}

return indexes.get(vertex)

}

find(i) {

while (i != parent[i]) {

parent[i] = parent[parent[i]]

i = parent[i]

}

Return i

}

union(x, y) {

int rootX = find(x)

int rootY = find(y)

parent[rootX] = rootY

}

}

**Unit Testing**

**Step 6. Preparation of Reports and Specifications**

\*\*Time Complexity Analysis for Operations:\*\*

DFS:

Add vertex (addVertex): O(1)

Add Edge (addEdge): O(1)

See Matrix: O(V^2) where V is the number of vertices.

See if two vertices are connected (seeConex): In the worst-case scenario, O(V + E) where V is the number of vertices and E is the number of edges.

Check Full Connectivity (hardlyConnect): O(V^3) in the worst case.

Prim:

Add Edge (addEdge): O(1)

Minimum Spanning Tree (minimumSpanningTree): In the worst-case scenario, O((V + E) log V) where V is the number of vertices and E is the number of edges.

Kruskal:

Initialization: O(V log V)

Minimum Spanning Tree: In the worst-case scenario, O(E log E) where E is the number of edges.

These analyses provide insight into how the execution time increases as the number of vertices (V) and edges (E) in the graph increases. The complexities mentioned are the worst possible and may vary in practice depending on the structure of the graph and the implementation of the algorithm.

**Class DFS**

**Testing for DFS:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Scenario** | **Input Values** | **Expected result** |
| **seeConex** | **Verify if two vertices are connected in a graph via DFS** | **Initial vertices, graph with defined connections** | **Returns true/false depending on connection** |
| **hardlyConnect** | **Verify if a graph is fully connected via DFS** | **Graph with multiple vertices and defined connections** | **Returns true if connected, false if not connected** |
| **seeConex (2)** | **Verify Connection Between Two Vertices in a Disconnected Graph** | **Unconnected vertices, graph with limited connections** | **Returns false** |
| **hardlyConnect (2)** | **Verify Connectivity on a Disconnected Graph** | **Graph with isolated vertices and no connections** | **Returns false** |

**Tests for Prim:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Scenario** | **Input Values** | **Expected result** |
| **addEdge** | **Adding an Edge to the Adjacency List in the Prim Algorithm** | **Vertices and Edge Weight** | **Correctly added edge** |
| **minimumSpanningTree** | **Finding the Minimum Spanning Tree in a Graph Using the Prim Algorithm** | **Graph with defined vertices and edges** | **Minimum Spanning Tree Correctly Found** |
| **minimumSpanningTree (2)** | **Finding the Minimum Spanning Tree in a Large Graph** | **Large graph with multiple vertices and edges** | **Minimum Spanning Tree Correctly Found** |

**Tests for Kruskal:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Scenario** | **Input Values** | **Expected result** |
| **minimumSpanningTree** | **Finding the Minimum Spanning Tree in a Graph Using the Kruskal Algorithm** | **Graph with defined vertices and edges** | **Minimum Spanning Tree Correctly Found** |
| **minimumSpanningTree (2)** | **Finding the Minimum Spanning Tree in a Disconnected Graph** | **Disconnected Graph with Multiple Components** | **Minimum Spanning Tree Correctly Found** |
| **minimumSpanningTree (3)** | **Finding the Minimum Spanning Tree in a Large Graph** | **Large graph with multiple vertices and edges** | **Minimum Spanning Tree Correctly Found** |

**Step 7. Design Implementation**

Programming Language Implementation

Ready to deploy:

Subroutine Specification

|  |  |
| --- | --- |
| Name |  |
| Description |  |
| Entrance |  |
| Return |  |
|  |  |

Construction

Writing code in Java programming language