SCRAP SALVAGE GAME

DAT 602 Assessment project

Abstract

A summary of planned screens and menus for scrap salvager game to be built for DAT602 assessment.

Johan Hobbs

Brief Description

The player clicks tiles to move around – when the player moves onto a square with an item on it, they can pick the item up. Most items are “Scrap” that can be converted to materials.

The player can use the materials to craft machines or parts, and may sell machines or parts or scrap to an NPC market place, and use the funds to buy different materials or other supplies.

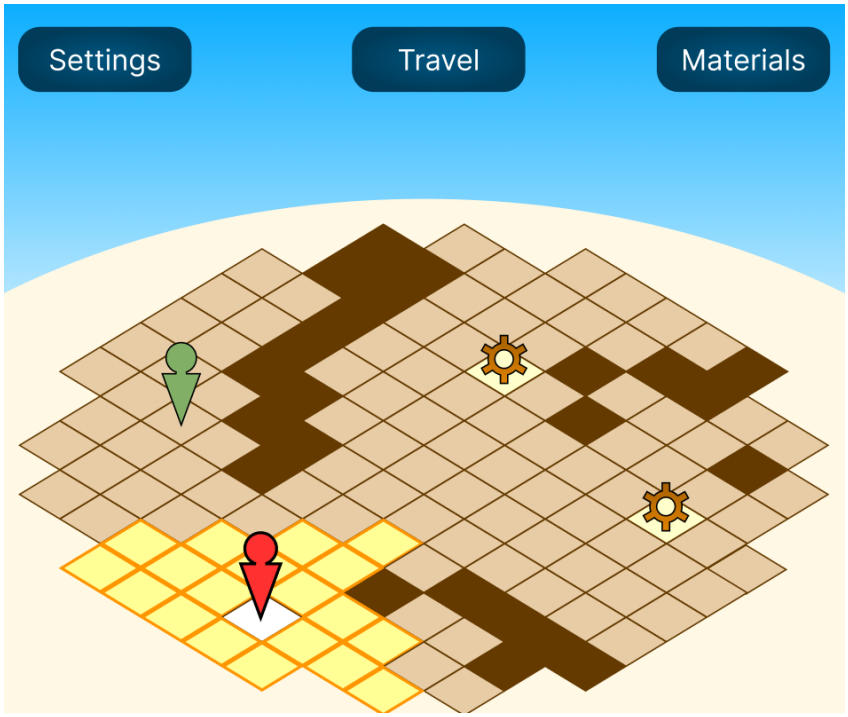
Main screens

Login Screen

Player selects their account, enters a password.

If the player enters both their account name and password correctly, they see a list of the characters they can log into the game as.

Outdoor Zone



An interactive screen that allows the player to navigate around a “zone”.

The grid, and all visual assets are placeholders now, and are subject to change.

The “sky” part of the map is intended to be an aesthetically pleasing way of reserving a certain amount of screen space for UI elements.

Impassable dark tiles may be either built into the map. The player can not move onto these, but may move around them, and with the required powerups, might be able to “hop” over them.

An icon representing the player is displayed on the map, and tiles that the player is able to move to are highlighted visually. (A red arrow with a circular head is here being used to represent the player.)

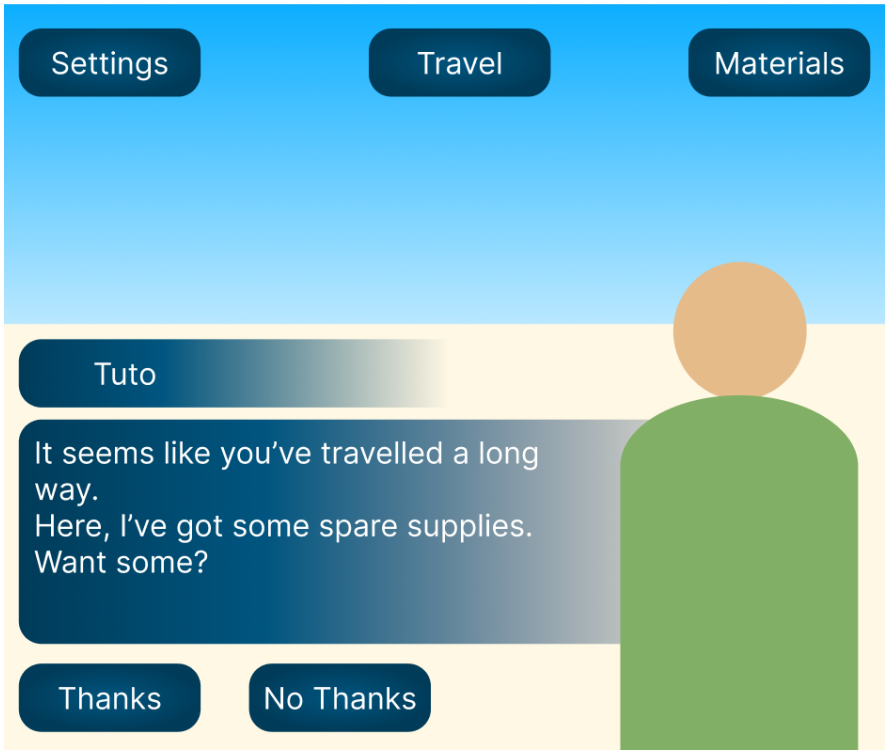
When the player has moved, it ends the player’s turn, and any entities controlled by the system can move.

The player can move to a “legal” square by clicking on it.

In the example, a system entity (NPC) is represented by a green arrow with a round head. The player may interact with an NPC by moving to an adjacent tile, and the NPC may interact with a player by moving to them.

Items that can be picked up are displayed on the map as well (represented in the example by “cogwheel” images) – the player or an NPC can pick an item up by moving to an item square.

NPC interaction



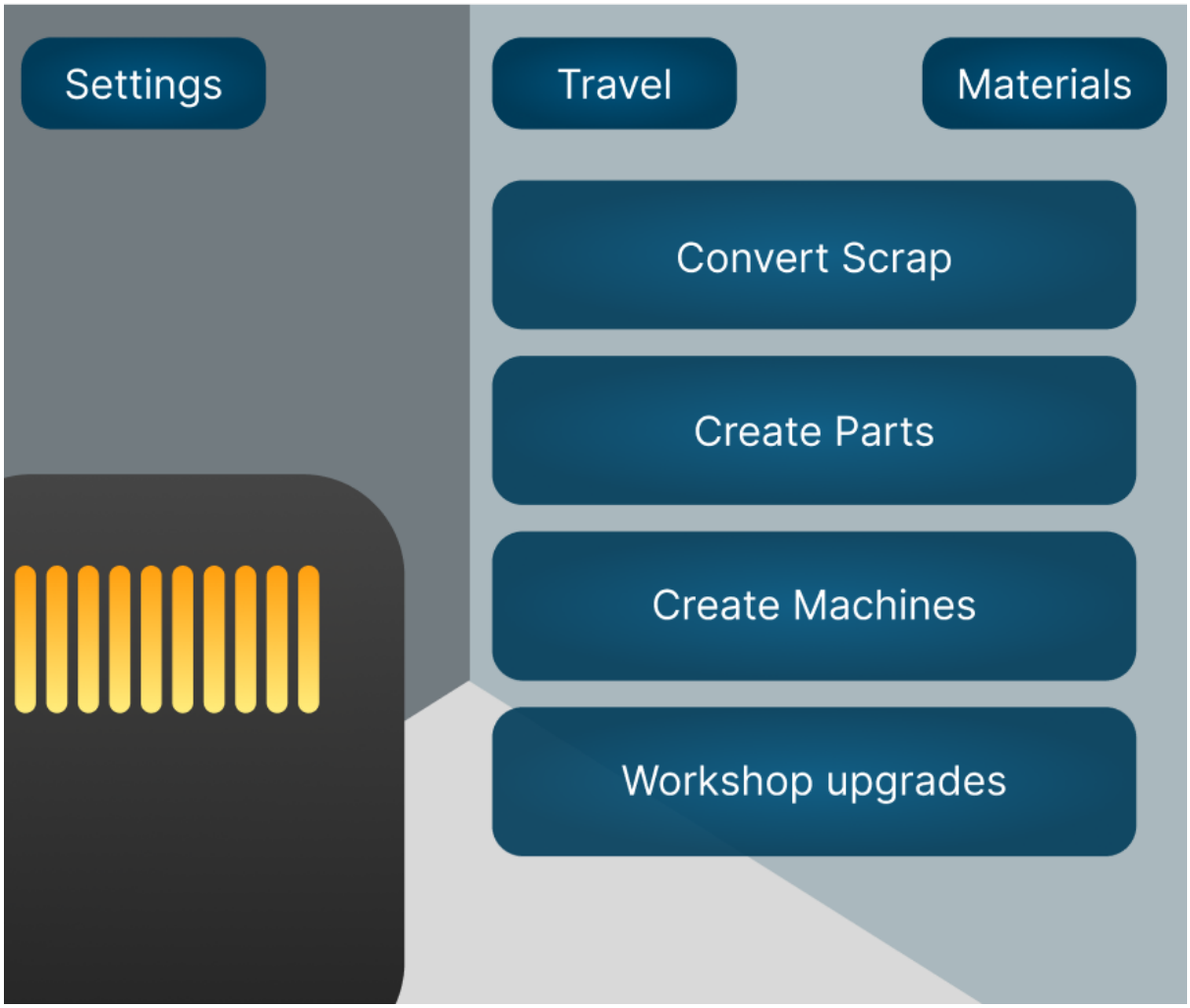
While interacting with an NPC, the NPC portrait should appear, and the NPC’s name should appear above their dialogue box.

Buttons below the dialogue box give the players options to respond.

A player's dialogue options may have different effects depending on the interaction – for example, in this interaction, the NPC "Tuto" might give the player some resources if the player says "Thanks".

Each NPC might have a table of potential interactions, including NPC dialogue, available player replies, and the different things that might happen if a player selects a particular option. (For example, the player may gain or lose resources, currency, or NPC friendship value, or gain an item)

Workshop Zone



The workshop allows the player to create things from materials that they can obtain by gathering, by trading with NPCs or by

Travel Menu

A Menu that will allow the player to travel from their current location to another location – for example, from the workshop to the outdoor zone.

Settings Menu

Adjust things like window colour (Dark mode vs light mode?) difficulty

Materials menu

A list of all the materials the player currently possesses, including scrap, machine parts, complete machines, and tools. (Menu name should possibly be changed to reflect this)

Relational Database Tables

See scrap-salvager-game-diagram.pdf for a relational diagram.

[TO DO: further analysis for each table will be added under each table's title]

Player

Every account that can play the game is a "player".

playerID: Primary key – unique, integer, not null. Serves as a unique identifier within the database and can't be altered.

username: unique, varchar, not null – the account's human-readable identifier for the player, and as such, is a compulsary field that must be unique.

password: varchar, not null. Is used to verify each user. It's best never to store a password in plaintext, so the actual stored value will be hashed.

Character

This table represents a player character in the game

characterID: Primary key – unique, integer, not null. Serves as a unique identifier for each character within the database and as such, can't be altered.

name: The character's name. Displayed in dialogue and parts of the user interface.

playerID: The unique identifier of the character's owner – a foreign key linking the Character table to the database's Player table.

mapID: The unique identifier of the map the character is currently on – this indicates the zone the character is in

currency: How much in-game currency the player is holding right now. Updated if the player purchases or sells an item.

xPosition: The x-coordinate of the player's current location in the current zone. If the current map has no dimensions, the value will be set to zero.

yPosition: The y-coordinate of the player's current location in the current zone. If the current map has no dimensions, the value will be set to zero.

xPosition and yPosition will be used jointly to determine where the character is on the map represented by **mapID**, and where the character can and cannot move to. This will be done by checking the map dimensions, **height** and **width** and the positions of other entities that have a relationship with the map the character is on.

[TODO: update doc, diagram so characters get a sprite like the NPCs and items]

Map

mapID: Primary key – unique, integer, not null. Serves as a unique identifier within the database and can't be altered.

name: The name of the map. This will be shown in the UI

mapTypeID: Foreign key linking to

height: the height of the map. Characters, NPCs, and tiles cannot legally move beyond the map boundary, so the **xPosition** should not be greater than the map height.

width: the width of the map. Characters, NPCs and tiles cannot legally move beyond the map boundary, so the **yPosition** of any character, NPC, or tile should be less than the map width.

The height and width cannot be null, but a map can be zero by zero – the "workshop" would be one such map.

[TO DO: Decide where to store the "default" tile type and background for each map. Maybe this should be stored under "Map type" instead of under map.]

Tile

Tiles are static entities that are placed on the map and cannot move

tileID: Primary key – unique, integer, not null. Serves as a unique identifier within the database and can't be altered.

name: The name of the tile in a human-readable format. Not shown in the UI, it's just to make queries a little easier.

passibility: [TODO: Think of a better name] the property of a tile that determines how the tile affects the movement of a character or NPC. For example, deep mud might slow a player down – a deep hole might be able to be jumped over, but a character or NPC cannot stop while in the middle of moving over one, and a character or NPC may be unable to move over a tall wall.

image: The image displayed in a tile's location.

Tile_Map

This is a singular instance of a tile on a map.

mapID: Joint primary key AND foreign key used to connect this entry to an entry on the Map table.

tileID: Joint primary key AND foreign key used to connect this entry to an entry on the Tile table, used to access the properties of the tile, such as

xCoordinate: the x position of the tile on the map.

yCoordinate: the y position of the tile on the map.

[TODO: Two tiles cannot occupy the same x position if they also share a y position – is there a way to enforce this using database rules?]

Item

itemID: The primary key, unique not null. You get the idea, it's a primary key.

name: The name of the item – shown on the in-game UI

category: The category that the item falls into – might be used for item sorting, such as crafting materials, survival items. As well as QoL purposes, that might affect NPC interactions/trade offers. See the "Behavior" entry under the NPC table.

description: Description of the item that may appear when examining an item in the UI.

value: The value of the item – the basic amount of in-game currency that it would normally be worth, either when purchasing or selling the item. While NPCs might make trade offers that value items higher or lower than their actual value, this will always be derived from the database value.

icon: An icon representing the item. This may be displayed in the UI, and will be displayed when an item is physically present on the map.

NPC

npcID: Primary key

name: The name of an NPC

behavior: A trait that influences this NPCs behaviour [TODO: This would be better as a foreign key referencing an external table. This still might happen but in order to make the overall database structure simpler while I'm figuring it out, I'm keeping it simple.] For example, an NPC with the behavior "Avaricious" might offer low prices on items and sell them higher, and seek out the highest value items that appear on the map – an NPC with the behavior "Hungry" might offer higher value items if the player offers them items in the "Food" category, and seek out those items first if they appear on the map – an NPC with the behavior "Fashionable" might seek to collect unique items within the "clothing" category that they do not possess yet, and seek them out first if they appear on the map – or an NPC with the behavior "Frugal" might be more cautious with their purchases.

currency: The amount of currency an NPC is carrying. They cannot offer more than the currency they are holding when trading with a player.

currentMap: Foreign key of the current map the NPC is on.

portrait: Portrait of an NPC that shows when the player is interacting with an NPC

sprite: Smaller image of the NPC as they appear on the map.

[TODO: Add X and Y positions, how did I forget that??]

[TODO: Do I need an additional 1-2 tables to make a template for NPC interactions in order to have each NPC have multiple possible interactions with dialogue, player choice, and multiple possible outcomes according to each player choice? Is this too complex – should complex events be pushed back to a later version in favor of more generic "trade" interactions?]

Character_NPC

npcID:

haracterID

amicability

Character_Item

characterID

itemID

quantity

NPC_Item

npcID: Joint foreign key with itemID that connects this to the NPC table.

itemID:Joint foreign key with npcID that connects this table to the item table.

quantity: the number of this item that the specified NPC is carrying

Recipe

[TODO: Does this table need to exist? Since all this table does is point from the Recipe_Item table back to the item table, would it be better to have an Item_Item table and skip this step altogether? The only functionality I can think of for the Recipe table being distinct from the item table is if you somehow wanted to have multiple different recipes for the same item, which is not currently planned.]

Recipe_Item

[TODO]

Needs to point to both the item being crafted and the materials used to craft the item.

In other words, this is a join table where both sides reference the same table – one item may have many ingredients, and each ingredient may be used as an ingredient for many items.

CRUD Events

[Write stuff here]