SIT102 Introduction to Programming

Credit Task 7.2: Space Game (part 2)

Overview

In part 1 of the Space Game task we got started making a small game. In this task we're going to extend what you have already made to make use of arrays for managing multiple power-ups. This will be a challenging task where you build out the functionality of this program. You will need to put together all of the aspects of this course, and use your problem solving and critical thinking skills to work out how to make use of the tools available to you.

Submission Details

Use the instructions on the following pages to build additional features in the lost in space game.

Submit the following files to OnTrack.

- Your program code
- A screen shot of your program running
- A screencast demonstrating your program in action

The focus of this task is on the declaration and use of dynamic arrays (using the vector class in C++), and pulling together all of our programming knowledge to build a larger program.

Instructions

Let’s continue on with our Lost In Space game. At this stage you should have a working HUD, with the ship, and a single power-up. Now that we have dynamic arrays we can extend this to handle working with multiple values.

The idea will be to randomly create power-ups for the player to collect.

Here is the diagram of the structs and enums involved by the end of this task.

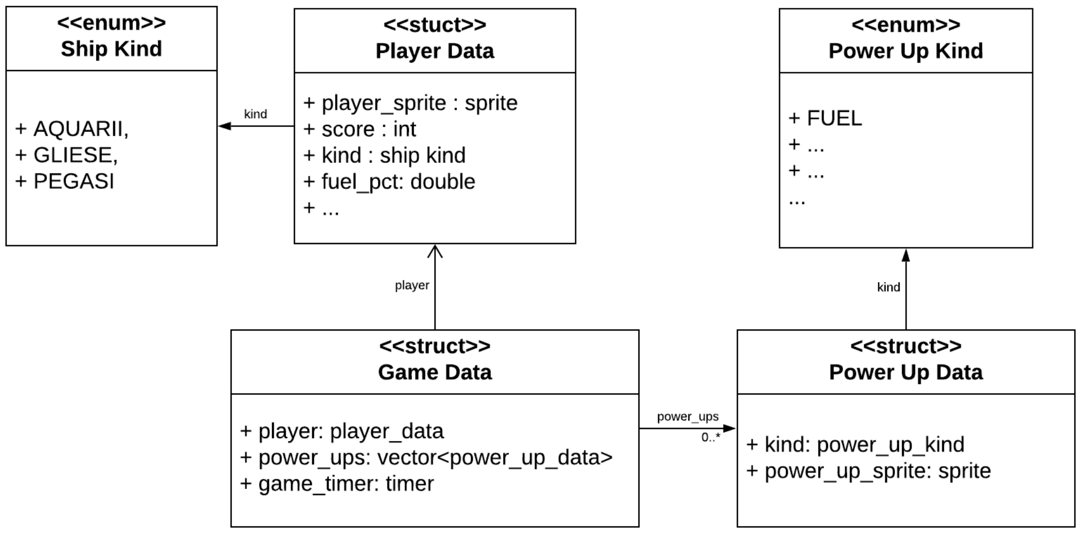

Figure: Diagram of structs and enums in code

The associated functions and procedures are shown in the following structure chart.

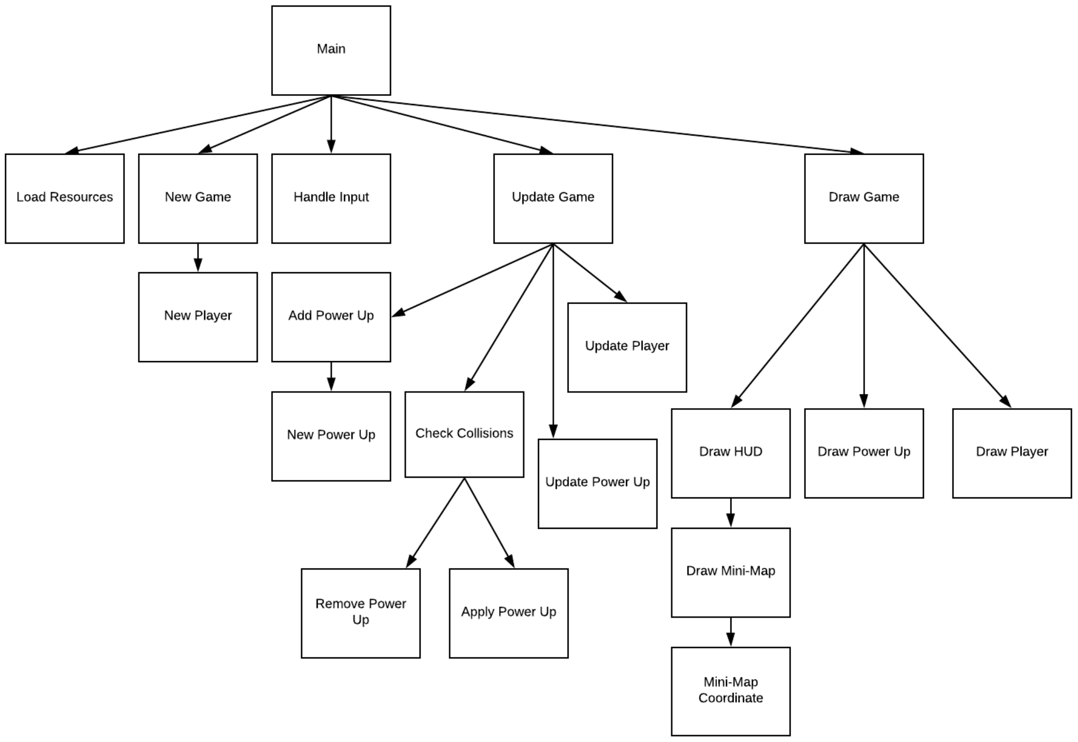

Figure: Structure chart for lost in space

This task will require you to develop a number of extensions to the Lost In Space program. This is likely to be a challenging tasks as it will require you to combine together everything you have learnt so far. You will need to use your problem solving skills and your understanding of how programming works in order to put this all together.

Focus on working in small iterations. Get one small change compiling, then run to test it, and make fixes as needed before moving on to the next part. I'd suggest keeping regular backups, so if things go totally off track you can go back to something that works.

The extensions will mostly involve creating and working with the `game_data` struct. This will represent the overall game itself, and will give us the context around which we will have the code to make this into a game.

To help manage your code you will need to create a new **lost_in_space.cpp** file, and matching header file. This will be used to manage the code related to the overall game, while the existing files will allow you to work with the player and power-ups.

Make sure to add a header guard in the **lost_in_space.h** header file. It will also need to include both the player and the power-up header files, as the game will use both of these. Also `#include <vector>` and add `using std::vector` to get access to the vector for your game.

In the following section I have provided notes to indicate how I would like the code to be orgaised. This is a design, so some things may not work out as expected. Let us know if some part of the design seems to be overly complex when you go to implement it. In these cases, we can discuss your proposed solutions.

As a starter, here is how I would go about starting to implement this myself (see notes on the following pages for the details for each part of this).

1. Start by creating the `game_data` struct and getting the program to work with this at a very basic level. So change main to include it, then create `new_game` to set this up. For example, you can still have `main` call `draw_player`, but now send it the player from the `game_data` value.

   Remove the sample power-up that was there before - just get this working with the player to start.

   Change the code, compile, fix, run, etc. Make sure you have the program running before you move on to the next step.

2. Next I'd add the `update_game` and `draw_game` procedures. We already have this working in `main` so it is just a matter of moving code out of main and into these procedures.

   Don't add any new functionality to these parts yet, just have it work with the player to start. At the end of this step you want `main` to be much simpler. It will just be creating the game, and using `update_game` and `draw_game` in its event loop.

   Change the code, compile, fix, run, etc. Make sure you have the program running before you move on to the next step.

3. With most of the code moved out of `main` and into `update_game` and `draw_game`, the next step would be to start to add in the ability to include power-ups in the game.

   Keep this simple. Focus on `draw_game` to start, and implement `add_power_up`.

   In `draw_game` have it loop though and draw all of the power-ups.

   In `add_power_up` code this to create and add the power-up as indicated in the notes that follow. For the moment you can adjust the values of the constants so that this will randomly positioning the power-up on the screen. That way you can see the power-ups when they are created, and the code should be easy to update as you only need to change the constant values once you want them to appear in the larger game space.

   Have `main` add two (or more) power-ups when the program starts as a test. You should be able to run this and see the power-ups on the screen. If nothing appears - add some code to print out where you created the power-up, and maybe some more code to show that it is drawing the power-up sprites.

   Change the code, compile, fix, run, etc. Make sure you have the program running before you move on to the next step.

4. Now that you have power-ups drawing, we need to update them so that they move.

   Have `update_game` update all of the power-ups. You should have the code to update one power-up, so it should be easy to get this to work with many. When you test this you should see the power-up move a little.

   Change the code… etc. etc.

5. The remaining steps would be:

   - Changing `update_game` to randomly add power-ups to the game. Then remove the test power-ups from `main`.
   - Drawing the minimap. At this point you would change the constants so that `add_power_up` uses a larger than screen game space.
   - Checking collisions between the player and the power-ups. I'd make this a few steps. 1) Get the collision tests working first but just printing to the terminal on hit, 2) add in the code to remove the power-up from the vector, then 3) add in `apply_power_up`.

   These can really be done in any order. Once you have the foundation in place you should be able to extend the game in a number of directions.

When you are done, submit your code, a screenshot, and your screencast (as a comment) of the game working to OnTrack.

Task Discussion

For this task you need to discuss at least the following with your tutor:

- Explain how how the vector is used to manage the power-ups in the game.
- Reflect on how functions and procedures help simplify the tasks of creating larger programs.
- Explain how functions and procedures work together with the vector and loops to simplify processing of values in arrays.
- Reflect on the iterative development process, how did it help you locate issues as you went?
- What could you add next? How would you approach this?

Notes on LostInSpace Design

Data for the game will be organised around three structs, and two enums:

- The **game_data** struct with be used to manage all of the game's data, as listed below. (This is a **new**)
    - A **player** (using the `player_data` type)
    - A `vector<power_up_data>` to store the **power_ups**
- The **player_data** struct will organise the data for the player. This will be accessible through the `game_data` which has a player. Each player has a sprite for its visual representation, a score, kind, fuel percentage, and other features you have already added. (This already exists, and should not need to change)
- The **power_up_data** struct organises the data for a power-up. This will be accessible via the game data's list of power-ups. Each power-up has its kind, and sprite. (This already exists, and should not need to change)

Functionally, the code should be organised into the following functions and procedures as shown in the Structure Chart. Create a new **lost_in_space.cpp** and **lost_in_space.h** files to manage the code related to the new game struct.

- **load_resources** to load the game resources. (No change)
- **new_game** will create and return a new game value. (This is new)

  Create a **new_game** function in the *lost_in_space.cpp* file and header. Use this to initialise and return a new **game_data** value.

  Implement this as **game_data new_game()**

  It will only need to initialise the `player`, as we start with no Power-Ups.
- **new_player** initalises the player at the start of the game. (No change)
- **handle_input** will read user actions and update the player. (No change)

- **update_game** will update the dynamic aspects of the game. This will include periodically creating power ups, checking collisions between the player and a power up, updating the player and updating power ups.

  Implement this as **void update_game(game_data &game)**

  It will update the player, and update *all* of the power ups. It will also need to check collisions, and periodically add new power ups to the game at random locations using `add_power_up`.

  Hint: You can use the rnd(int) function to generate a random number. Remember you already have a function to create a new power up at a given location. So call **new_power_up** and pass it two random values, then add the resulting data to the **power_ups** vector in the game you are initialising (using the vector's **push_back** method).

  Hint: The rnd() function can be used in an if condition to give some code a random chance of running. Using this version of `rnd` makes it easy to give that chance a % chance of running by testing if the value returned is < that percent. For example, `rnd() < 0.05` will be true 5% of the time.

- **add_power_up** creates a random power up, and adds this into the game. (This is new)

  Implement this as **void add_power_up(game_data &game)**

  Create a new power_ups and position it within a space ranging from -1500 to +1500 for both x and y (using the `new_power_up` function). Add the new power-up in the `power_ups` vector in the `game`.

  The range for the x and y values gives us a space larger than the screen for the player to search. The minimum and maximum should be coded as constants in your program.

- **new_power_up** is used to create a power up at a specified location. The type is randomly selected. (No change)

- **check_collisions** will check if the player has hit a power up. When this happens the power up is removed from the game, and it has an effect on the player. (This is new) Implement this as **void check_collisions(game_data &game)**

  This will loop backwards for all power ups in the game (i.e. from last to first), to check each power up against the player. Perform the check using the sprite_collision function of SplashKit to check the player sprite with the power up sprite.

  When a collision occurs, call `apply_power_up` and `remove_power_up` for the identified power up.

- **apply_power_up** will apply the effect of the power up to the player. (This is new)

  Play a sound effect (something positive, a "you got it" sound). Then apply the effect of the power up itself.

  - Have a **fuel** power up add 25% to the player's `fuel_pct` to a maximum of 100%.
  - Adjust your other values as appropriate.

- **remove_power_up** will remove a power up from the game. (This is new) Implement this as **void remove_power_up(game_data &game, int index)**

  Remove the power up at the indicated index from the game's power ups vector.

- **update_power_up** will update the power-up's sprite. (No change)

- **update_player** will update the player's sprite. (Some modifications)

  In addition to updating the player's sprite. Also add code to reduce the `fuel_pct` by a small amount each update. You can calculate the percentage to remove based on update being called 60 times a second. So if 100% fuel lasts one minute, you can remove 1/(60*60) = 0.000278 from the `fuel_pct` each update.

  Add additional code to update anything related to the power-ups you have added.

  Hint: If you want to display a power up for a period of time, an each easy option would be to add a counter to the player_data struct to say how long it has left. Then you can subtract one each update, until it gets to 0. If that value is > 0 then the update is active.

- **draw_game** will draw the game. (This is new)

  Implement this as **void draw_game(const game_data &game)**

  Move the code from main that does the drawing into this procedure… have it clear the screen, draw the hud, the player, draw *all* of the power ups, and refresh the screen.

- **draw_hud** will draw the heads up display (Needs additions)

  Should be mostly as existing, with additional code to draw the mini map. For this, `draw_hud` will need access to the power_ups vector. It can then pass this to `draw_mini_map` in order to show where the power-ups are.

- **draw_mini_map** will draw a scaled down map. (This is new)

  Implement this as **void draw_mini_map(const vector<power_up_data> &power_ups)**

  Loop through each power up and use `mini_map_coordinate` and draw_pixel to draw a colored pixel onto the black mini-map.

  You can use rgba_color to create a color with some transparency. This way multiple power ups at the same location will appear brighter then a single power up.

- **mini_map_coordinate** will get the mini map coordinate for a given power up. (This is new)

  Implement this as **point_2d mini_map_coordinate(const power_up_data &power_up)**

  This will create a Point 2D value to represent the location of the `power_up` on the mini map.

  - The mini-map is 100x100 pixels, and the overall space of the game is 3000x3000 pixels.
  - Power ups positioned from -1500,-1500.

  So to calculate the mini-map coordinate you need to:

  - Subtract the offset from the top left (i.e. add 1500 as the offset was -1500) from the x and y values of the power-up sprite to find its position relative to the top-left most position in the game. (The `sprite_x` function can be used to access the x value of the sprite.)
  - You can then get the proportion of this value to the overall game area, multiply this by 100 (the width/height of the minimap), and add the offset to the location of your minimap on the screen.

  This gives:

  - Mini Map X = (sprite x - top left offset) / total width * mini map width + mini map x
  - Mini Map Y = (sprite y - top left offset) / total height * mini map height + mini map y

  For example: What is the mini-map coordinate of a power-up at 615,-324 and the minimap is drawn at 10,50?

  - Mini-map x is (615 - -1500) / 3000 * 100 + 10 = 80.5
  - Mini-map y is (-324 - -1500) / 3000 * 100 + 50 = 89.2

  You can use the point_at function to convert the x, y values into a `point_2d` value to return.

- **draw_power_up** draws the power up sprite. (No change)

- **draw_player** draws the player sprite. (No change)

- **main** will coordinate the overall game. (This needs updating)

  This should now create a new game then loop until the window is closed, calling `handle_input`, `update_game`, and `draw_game`.