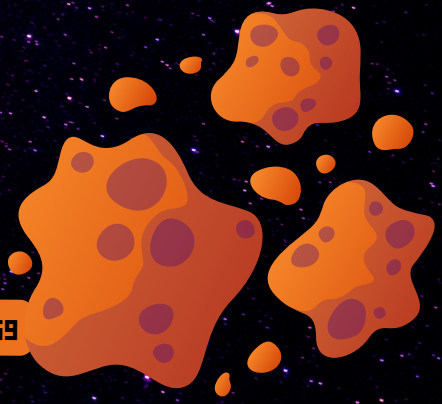
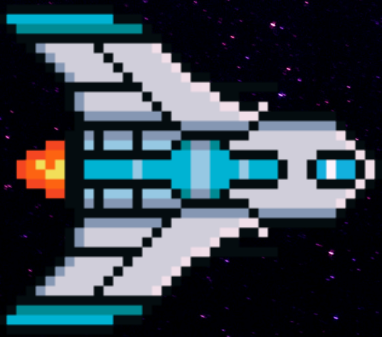
ASTEROID DASH

PROGRAMMING ASSIGNMENT 2



DEADLINE: FRIDAY, 15/11/2024 AT 23:59:59



Hacettepe University - Computer Engineering

BBM203 Software Practicum I - Fall 2024

ASTEROID DASH

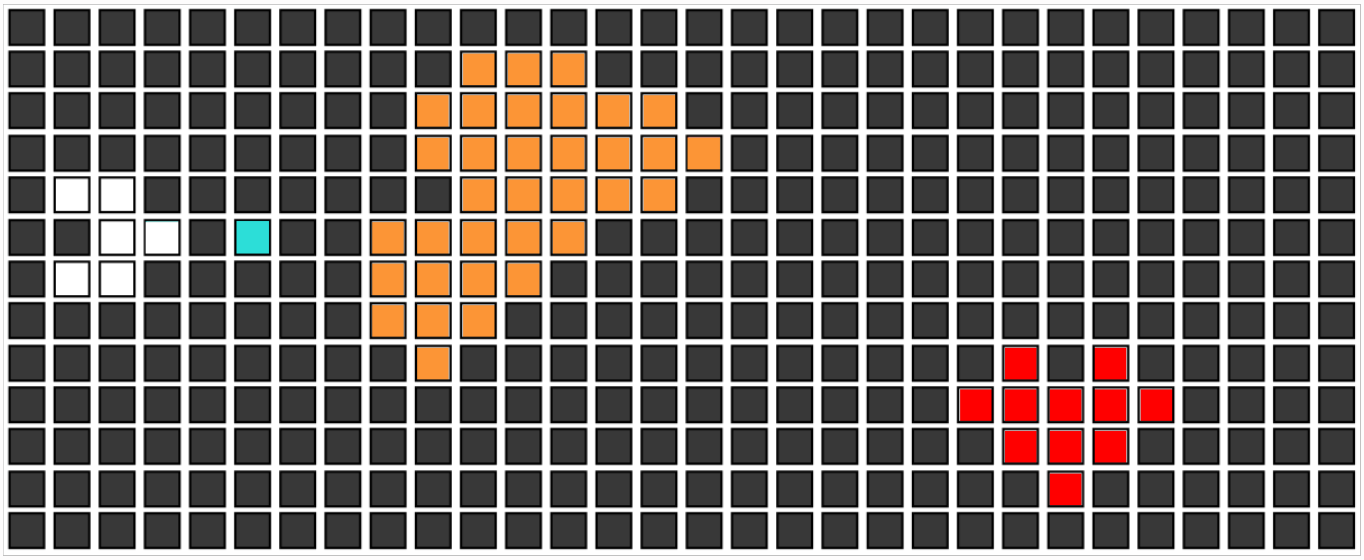
Topics: Linked Lists, Dynamic Memory Allocation, Matrices, File I/O

Course Instructors: Assoc. Prof. Dr. Adnan ÖZSOY, Asst. Prof. Dr. Engin DEMİR, Assoc. Prof. Dr. Hacer YALIM KELEŞ
TAs: M. Aslı TAŞGETİREN, S. Meryem TAŞYÜREK, **Asst. Prof. Dr. Selma DİLEK***, **Alperen ÇAKIN***
Programming Language: C++11 - **You MUST USE this starter code**
Due Date: **Friday, 15/11/2024 (23:59:59)**

Asteroid Dash

The Next Viral Asteroid-Smashing Score-Stashing Game for the Masses

Welcome to **Asteroid Dash**, where your coding skills will be tested in a fast-paced space adventure! In this assignment, you will develop an arcade-style game where players navigate through a treacherous asteroid belt. With asteroids to avoid, power-ups to collect, and strategic shooting mechanisms, this task will challenge your understanding of dynamic arrays, memory management, and object-oriented design in C++. As elite programmers from **HUBBM** and **HUAIN**, you are tasked with building the engine behind **Asteroid Dash**, a high-octane game where players must guide their spacecraft through waves of incoming asteroids. At your disposal are the powerful tools of object-oriented programming and dynamic memory management, which you'll need to successfully manage space objects, collision detection, and time-based gameplay.



Asteroid Dash takes place in a grid-based space environment where asteroids appear from the right and move leftward toward the player's spacecraft. The spacecraft can shoot to defend itself, but with limited ammo, players must tactically manage their resources while dodging incoming asteroids. Power-ups that replenish ammo or restore lives add a dynamic element to the game. Each step in the game updates the grid, moving asteroids, rotating them as they are damaged, and reflecting the player's actions. This assignment emphasizes linked lists and essential C++ concepts such as file I/O, class design, and memory management, providing students with practical experience in developing a real-time, event-driven game. **Will you survive the asteroid belt and rise to the challenge?**

1 Reading the Input Data and Initializing the Game

In this section, we outline game initialization via provided inputs. Pay close attention to dynamic memory allocation requirements, as they are crucial for full credit.

1.1 Input Files and Command Line Arguments

An input file containing details about the game's `space_grid`, structured as a 2D $rows \times cols$ matrix, will be supplied in DAT format via the *first command line argument*. Your task is to parse this file within your program to set up the `space_grid` attribute within the `AsteroidDash` class. The content of a sample input file given as `space_grid.dat` with 10 rows and 20 columns is illustrated on the right. The file consists of rows of digits, with individual digits separated by a single space, and each row ending with a newline character.

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

The second input file, in DAT format and supplied as the *second command line argument*, contains the game's **celestial objects** represented as instances of the `CelestialObject` class, which have 2D $height \times width$ shape matrices, given within square brackets [ ] for asteroids and curly braces { } for power-ups.

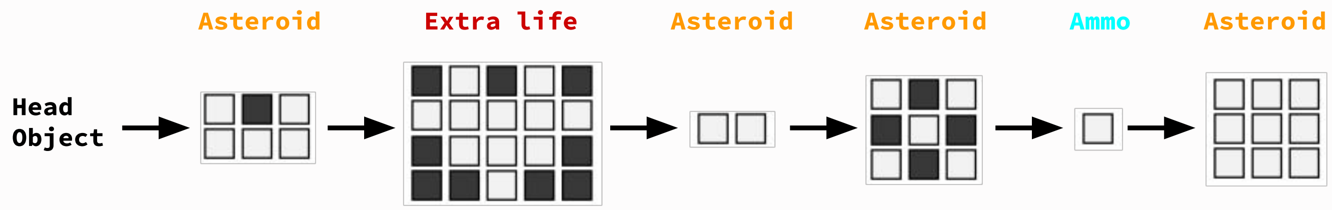
Each celestial object also includes metadata lines below its shape indicating its properties:

- **`s:`** indicates the **starting row** in the space grid where the object's top left corner will appear. Objects will always enter from the rightmost (last) column of the grid.
- **`t:`** specifies the **tick** or **time step** in the game when the object should begin entering the grid from the right.
- **`e:`** (for power-ups only) defines the **effect** the power-up will have on the player: either **`life`** for adding an extra life or **`ammo`** for replenishing the player's ammunition.

Your program should use this file to generate a **linked list** of the game's celestial objects in the `AsteroidDash` class. An excerpt from an example file, `celestial_objects.dat`, is displayed on the right. Celestial objects are placed in the order they are meant to appear in-game (from the right side of the space grid moving towards the left) in their initial unrotated state. **Note that multiple objects may enter the space grid simultaneously, but to simplify the task, they will never be positioned to collide with each other, even if any of them rotate after being shot**.

```
[101
111]
s:0
t:5

{01010
11111
01110
00100}
s:5
t:10
e:life

[11]
s:5
t:15

[101
010
101]
s:7
t:15

{1}
s:4
t:23
e:ammo

[111
111
111]
s:2
t:25
```

It is crucial to recognize that the $height$ and $width$ of individual objects may vary, necessitating dynamic memory allocation for all celestial objects and their rotations. An illustration of the celestial objects from this sample input is provided below:



The input file, in DAT format and supplied as the *third command line argument*, contains details about the **player's spacecraft**, represented as an instance of the `Player` class. A sample excerpt from `player.dat` is shown on the right. The file includes both the spacecraft's initial position and its shape, formatted as follows:

```
3 0
1 1 0
0 1 1
1 1 0
```

- The first line provides the **initial position** of the player's spacecraft on the space grid, indicating the starting `row` and `column` where the top-left corner of the spacecraft will appear.
- The subsequent lines define the spacecraft's **2D shape** as a matrix of 1s and 0s, where each `1` represents an occupied (full) cell of the spacecraft and each `0` represents an empty cell.

This file should be read by your program to create a `Player` instance and set the player's spacecraft in its initial position on the space grid.

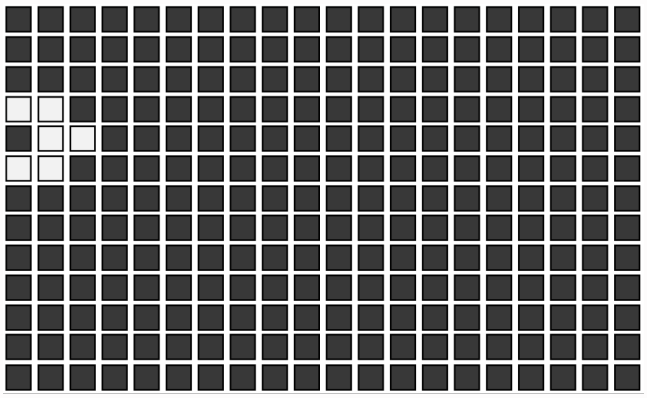
The input file containing the game's **commands**, represented as strings, will be supplied in `DAT` format as the *fourth command line argument*. Your program is tasked with interpreting this file's contents to facilitate gameplay, an operation to be implemented within the `GameController` class. An excerpt from a typical input file, named `commands.dat`, is displayed to the right for reference.

```
MOVE_UP
MOVE_DOWN
MOVE_RIGHT
MOVE_LEFT
SHOOT
NOP
```

The *fifth command line argument* designates the filename of a text file where the **leaderboard** data is stored (further details will follow in the subsequent sections of the assignment instructions). Lastly, the *sixth command line argument* determines the **current player's name** for leaderboard identification purposes.

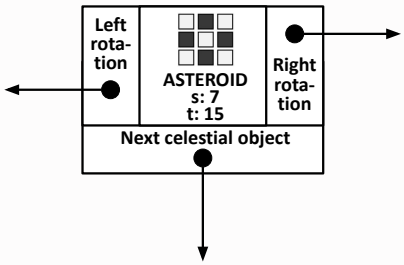
1.2 Initializing the Grid

The game's `space_grid` is managed as a dynamically allocated 2D integer matrix, where **zeros** (0's) represent unoccupied cells (illustrated as black squares) and **ones** (1's) represent occupied cells (illustrated as white squares). The player's spacecraft should be positioned on the grid according to its shape, **with the top-left corner placed at the specified start row and column**. You may assume that the grid's dimensions are sufficient to accommodate the player's spacecraft without exceeding boundaries.

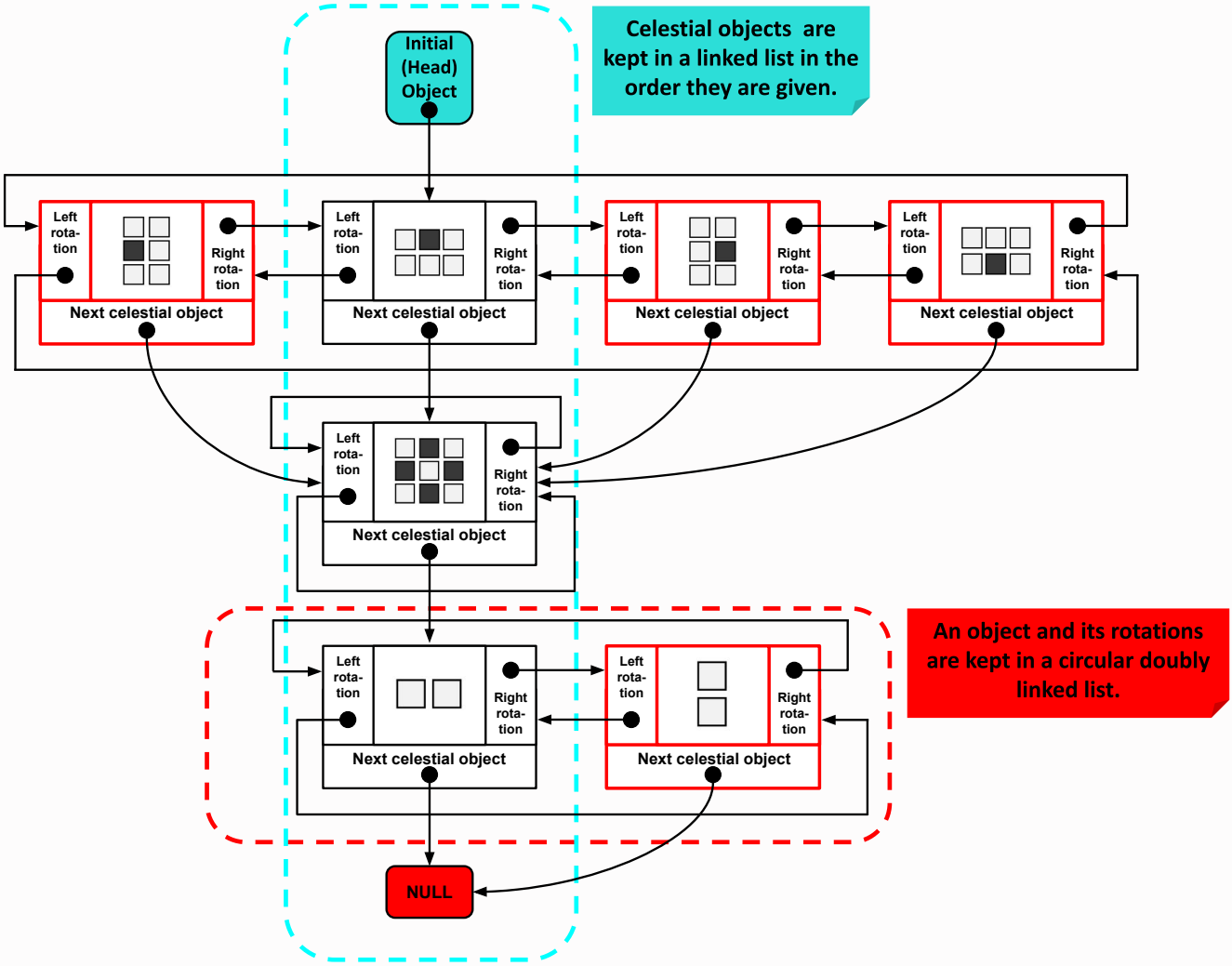


1.3 Initializing the Celestial Object List

In the game, **celestial objects will be represented as nodes (instances of `CelestialObject` class) in a multi-level linked list** reachable from the pointer `celestial_objects_list_head` in `AsteroidDash` class. Upon reading the sequential celestial objects from the corresponding input file, your first step is to identify each object's `shape`, `object_type`, `starting_row`, and `time_of_appearance`, instantiate it, and then integrate it into the game's linked list of celestial objects. This integration is achieved by accurately assigning the `next_celestial_object` pointer of the list's preceding celestial object. **Moreover, you're required to compute all possible rotations of each object, storing them within a circular doubly linked list**.



Each celestial object's `right_rotation` pointer should allow sequential access to its clockwise rotations, looping back in a circular manner. Similarly, the `left_rotation` pointer should enable consecutive access to its counter-clockwise rotations, also in a circular pattern. **It is important to note that both the original celestial object and each of its rotations must consistently point to the default state of the subsequent celestial object via the `next_celestial_object` pointer**.



Note that each celestial object can have none or multiple rotation states (needed for the cases when an asteroid rotates due to a collision with a projectile). Rotations are organized in a circular doubly-linked structure using the `right_rotation` and `left_rotation` pointers.

2 Key Functionalities to Be Implemented

In this section, we cover gameplay rules and essential functionalities for implementation. Close adherence to dynamic memory allocation requirements is crucial for full credit.

2.1 Reading and Processing Commands

`AsteroidDash` features seven commands that will be listed in the corresponding input file, each on a new line, without spaces. Commands must be interpreted dynamically and immediately as they are encountered (each command executed in one game tick), not stored in memory, mimicking real-time gameplay conditions. The commands are as follows:

- `PRINT_GRID`: Print the state of the space grid after the necessary updates in the current game tick.
- `MOVE_UP`: Move the player spacecraft one space up, if possible.

- `MOVE_DOWN`: Move the player spacecraft one space down, if possible.
- `MOVE_RIGHT`: Move the player spacecraft one space to the right, if possible.
- `MOVE_LEFT`: Move the player spacecraft one space to the left, if possible.
- `SHOOT`: Shoot a projectile.
- `NOP`: No-operation in the next tick: do nothing.

Unknown commands are handled with printing an error message, but processing continues with the next command. See an example on the right.

```
Unknown command: GIMME_POINTS
```

2.2 Game Rules

While this game draws inspiration from the classic *Shoot 'em up* games, it diverges in its ruleset. As such, it is crucial to meticulously review the game rules and execute the assignment tasks, paying close attention to each key requirement and detail outlined.

2.2.1 Player and Celestial Objects Movements and Collision Detection

The player's spacecraft and celestial objects both interact with the `space_grid` as they move and respond to player actions or game events. The player's spacecraft enters the grid at a specified starting position, as indicated by the player data file. Each time step, celestial objects advance one cell to the left until they either exit the grid or collide with the player. Simultaneously, the player can control the spacecraft's movements and initiate actions such as firing projectiles at incoming objects. The figure below illustrates the movement of celestial objects as the game time (ticks) progresses. Here, we assume that the first celestial object enters the space grid at game time $t = 0$ and that the player is initialized from the cell $(3, 1)$, and is not moving at all during these ten game ticks.

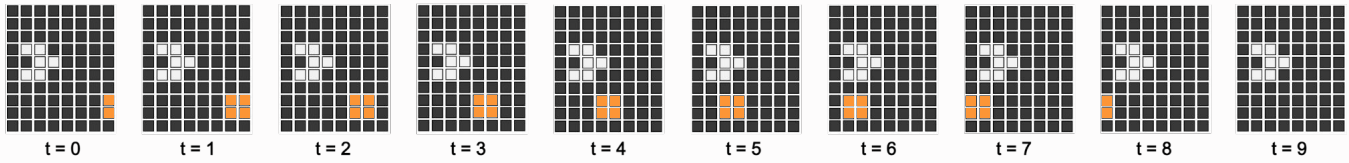
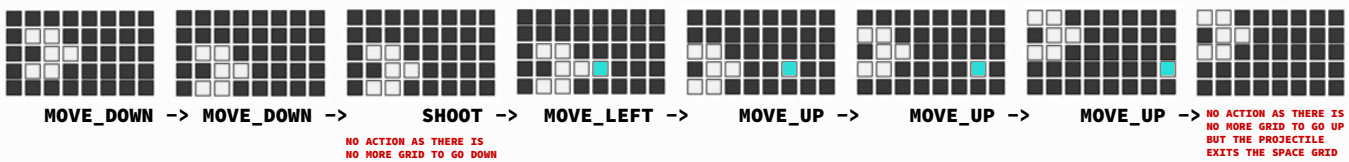


Figure 1: Celectial objects' movement on the grid as the game time progresses.

Player Movement and Actions: The player can move the spacecraft up, down, left, or right within the grid bounds, but only **one cell per tick**. Any attempt to move beyond the grid's edge will be nullified, leaving the player's position unchanged. When the `SHOOT` command is issued, a projectile is launched from the center row of the spacecraft, specifically from the column immediately to the right of the spacecraft. This projectile moves continuously rightward until it either collides with an object or exits the grid. You may assume that the spacecraft's height will always be an odd number, simplifying the calculation of its center row.



Collision Detection: The game must continuously check for collisions involving celestial objects and the player's spacecraft or projectiles. For instance, when celestial objects advance, a collision with either the player's spacecraft or a projectile hit must be properly addressed.

Collision of Asteroids with Player Spacecraft: If an asteroid collides with the player's spacecraft, the player's remaining lives should decrease, and the asteroid should be removed from the space grid. If this collision consumes the player's last life, the player should also disappear from the space grid. Figure 2 illustrates the first case. Check Section 2.2.4 for instructions on handling game-over scenarios.

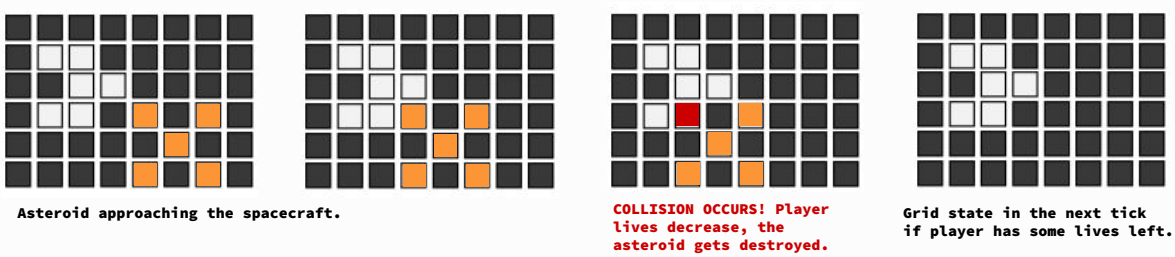


Figure 2: Illustration of the asteroid-player collisions.

Collision of Celestial Objects with Projectiles: Projectiles fired by the player move in a straight line until they either encounter an asteroid or reach the edge of the grid. Upon hitting an asteroid, the projectile removes a single cell from the asteroid (the first cell it contacts), dynamically altering the asteroid's shape. This modification should prompt an update across all rotations of the celestial object, generating them from the scratch.

The effect of a projectile hit on an asteroid depends on the impact location:

- If the projectile strikes the **upper part** of the asteroid, the asteroid should rotate clockwise (right).
- If the projectile strikes the **lower part** of the asteroid, it should rotate counterclockwise (left).
- If the projectile strikes the **middle section** (for asteroids with odd height), no rotation should occur.

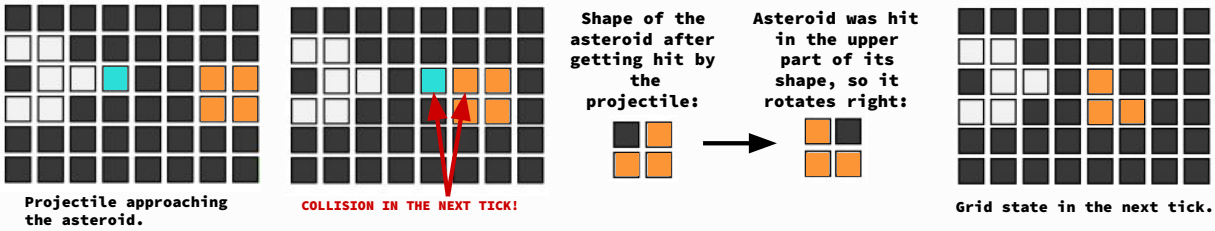


Figure 3: Illustration of the asteroid collisions.

When rotating a celestial object, it is essential to keep the object's top leftmost cell in the same position on the `space_grid`. For example, if a celestial object has dimensions $n \times m$ and its top leftmost cell is positioned at $grid[i][j]$, then after rotation (resulting in a size of $m \times n$), the top leftmost cell of the newly rotated celestial object should still be located at $grid[i][j]$ before advancing in the next tick. This is illustrated in the figure below.

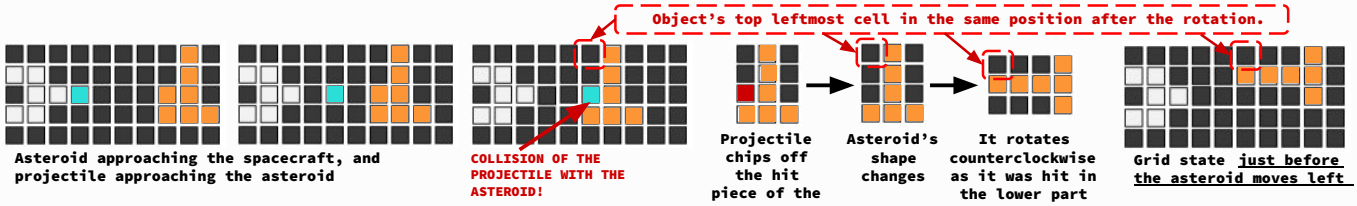


Figure 4: Rotation examples for non-square celestial object shapes.

To accomplish this, structural updates are necessary within the celestial objects list. When an asteroid is hit by a projectile, one of its cells is blasted off, altering its shape. This change must first be applied to the shape of the affected `CelestialObject` instance. **Subsequently, the rotations of the object need to be recalculated based on the updated shape (DO NOT FORGET TO FREE THE MEMORY OCCUPIED BY THE OLD ROTATIONS!)**. In some cases, this alteration may affect the total number of the object's rotations. Figures 5 and 6 illustrate this process and its impact on the celestial objects list within the game.

Figure 5: Updating object's shape and rotations after getting hit by a projectile - steps.

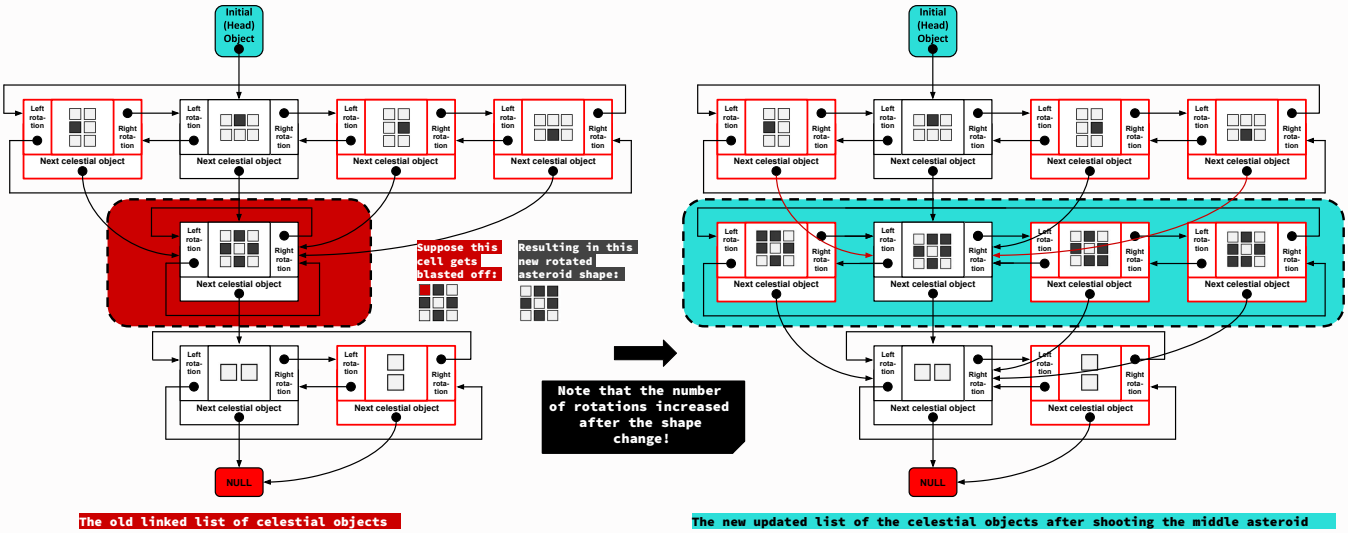


Figure 6: Updating object's shape and rotations after getting hit by a projectile - the linked list change with new rotations for the asteroid.

Now imagine that a second projectile hits the asteroid, as illustrated in Figure 7.

Figure 7: Updating object's shape and rotations after getting hit by a projectile for the second time.

The update in the linked list must also reflect this change, as illustrated in Figure 8.

Now suppose that a third projectile hits the asteroid as illustrated in Figure 9. **Note that even if the impact of multiple projectiles causes parts of the asteroid to become disconnected, the asteroid will continue to behave as a single entity, maintaining its overall dimensions.**

Since the asteroid gets hit in the center of its shape, it does not rotate. The update in the linked list must also reflect this change (see Fig. 10).

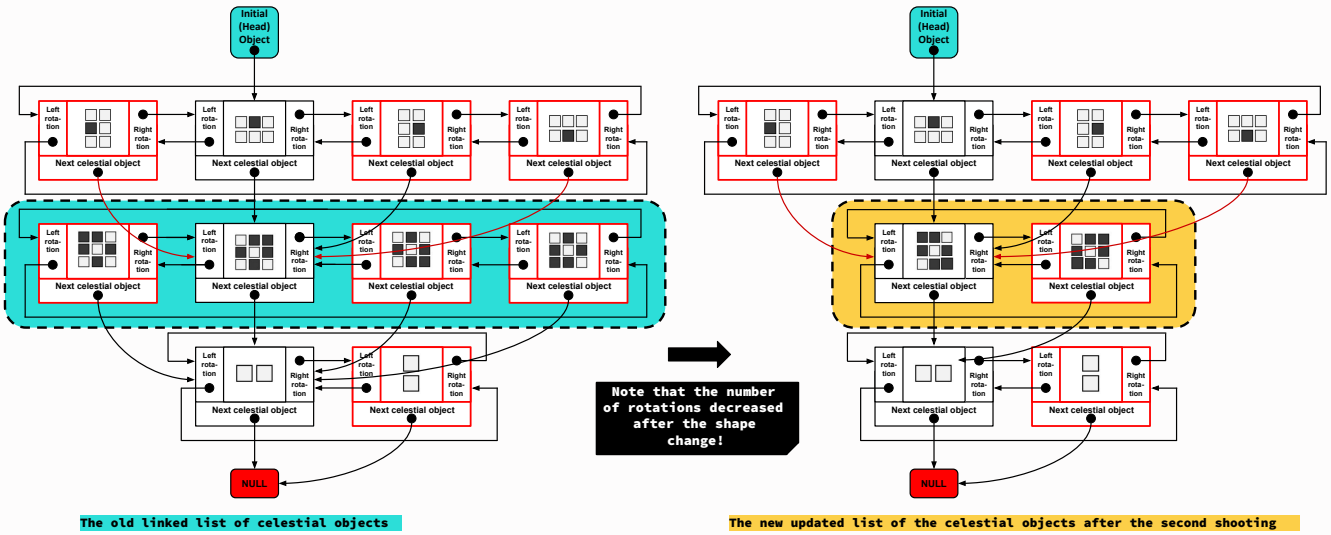


Figure 8: Updating object's shape and rotations after getting hit by a projectile again - the linked list change with new rotations for the asteroid.

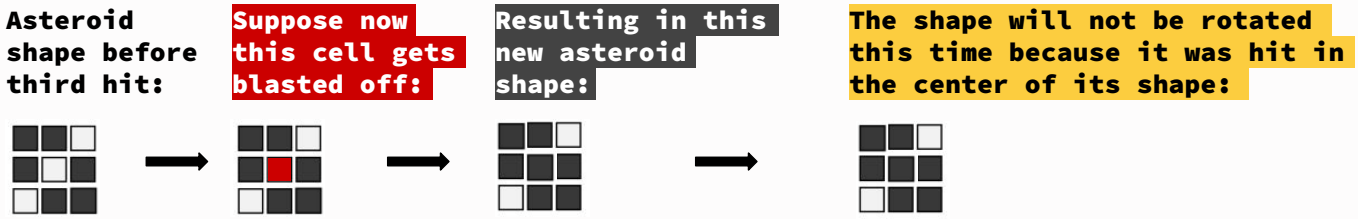


Figure 9: Updating object's shape and rotations after getting hit by a projectile for the third time.

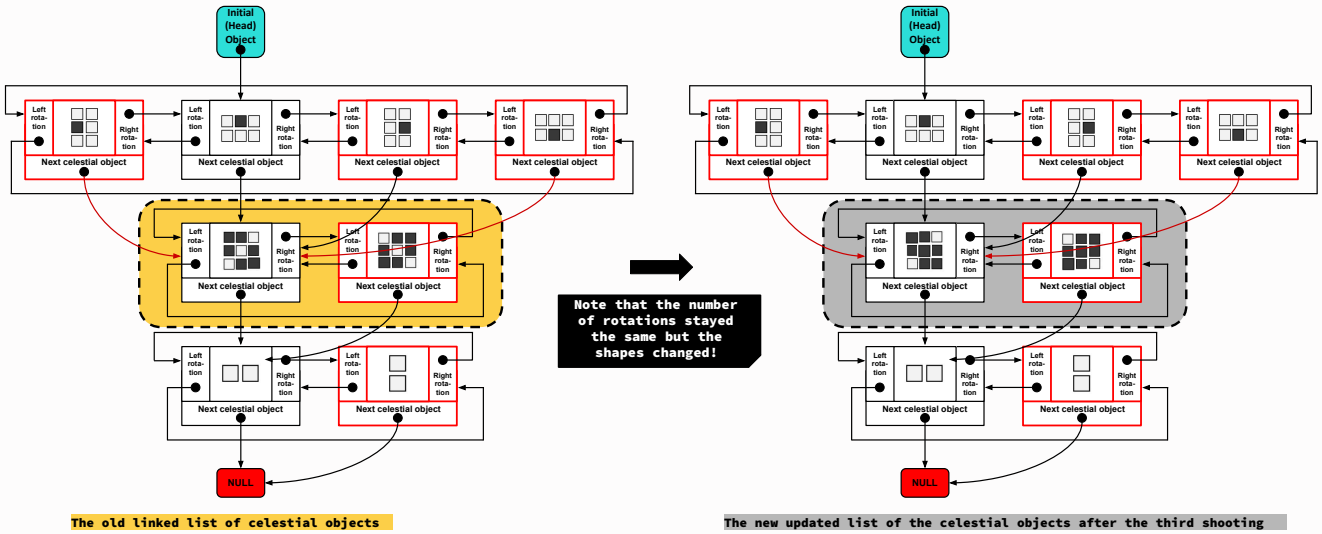


Figure 10: Updating object's shape and rotations after getting hit by a projectile for the third time - the linked list change with new rotations for the asteroid.

If a projectile encounters a power-up (extra life or ammo), the power-up remains unaffected, and the projectile passes through, continuing its path.

Power-Up Interactions: Power-ups provide specific benefits to the player upon collection (collision with the spacecraft), as defined in the celestial objects input file:

- `e:life` - Life Up - Increases the player's total lives by one (no limit on the number of lives).
- `e:ammo` - Ammo Refill - Replenishes the player's ammunition to `max_ammo`.

When a power-up is collected by the player, it behaves similarly to an asteroid upon collision, disappearing from the space grid. Figure. 11 illustrates these interactions with power-ups.

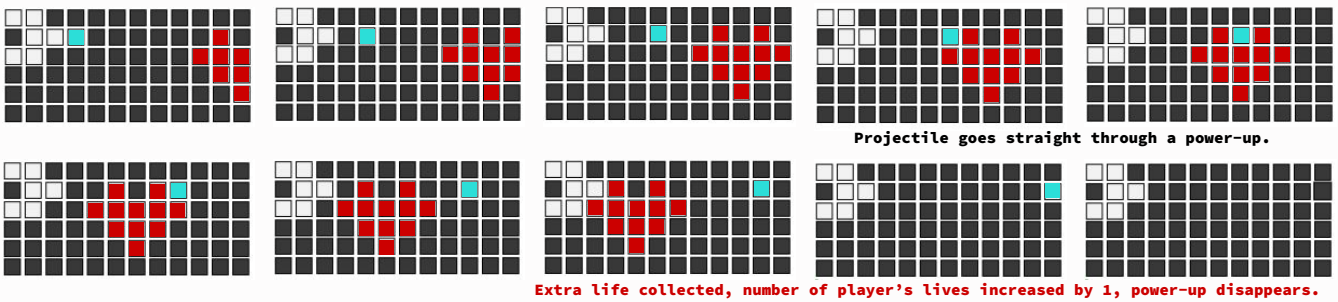
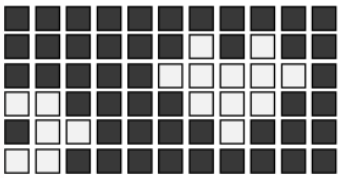


Figure 11: Interacting with power-ups.

Printing the Grid: The `PRINT_GRID` command should first display the current game time in ticks, followed by the player's remaining lives, ammunition, and current score. Next, it should display the all-time high score. If this is the first game played or if the player's current score surpasses the existing record, the new score should be saved and shown as the all-time high. After this information, the grid should be printed row-by-row, accurately representing all active objects, the player's spacecraft, projectiles, and any other occupied cells. An example output of the `PRINT_GRID` command is shown on the right.

```
Tick: 29
Lives: 3
Ammo: 7
Score: 29
High Score: 154
```



2.2.2 Scoring System

The scoring system in `AsteroidDash` is designed to reward players based on their interactions with celestial objects and actions taken throughout the game. The scoring breakdown is as follows:

Action	Score Impact
Projectile Hits Asteroid	Each time a projectile successfully hits and removes a cell from an asteroid, the player earns **10 points**. If the hit removes the last remaining cell of an asteroid, these 10 points are also awarded before the bonus points (see next row).
Destroying Entire Asteroid	When all cells of an asteroid are removed, the player receives a bonus of **100 points** for each **occupied** cell of the asteroid's original size, rewarding complete destruction of asteroids.
Surviving Each Game Tick	The player earns **1 point** for each tick they survive without colliding with an asteroid. This survival bonus incentivizes prolonged gameplay without taking damage.
New High Score	If the player's current score exceeds the all-time high score, it becomes the new record, displayed on the leaderboard.

The player's **final score** is recorded at the end of the game and entered into the `Leaderboard` for long-term tracking, if it is one of the top ten all-time high scores.

2.2.3 Leaderboard

The game features a leaderboard that holds up to **10** high score entries, with each entry comprising a **score**, **timestamp** (acquired by `time(nullptr)` upon entry creation), and **player name**. Prior to

each game session, the system should try to load the leaderboard from an existing file given as the *fifth command line argument*, if available. However, this file may not be present during an initial run, indicating an absence of high scores. Your code should handle this. Once a game session concludes, regardless of the reason for which the game has ended, the leaderboard needs to be refreshed to reflect the latest scores and then saved back to the same leaderboard file. This leaderboard retains a maximum of **10** top scores across all sessions and must be consistently sorted in descending order based on scores. It's important to note that there may be instances where fewer than ten high scores are stored. In such scenarios, avoid attempts to parse ten entries. Instead, adopt a flexible approach to prevent memory errors.

The text file contents should be structured as follows:

```
<score> <timestamp> <player_name>
```

As an example:

```
40000 1697910655 AsteroidBuster
1200 1697910655 StackOverthrower
...
```

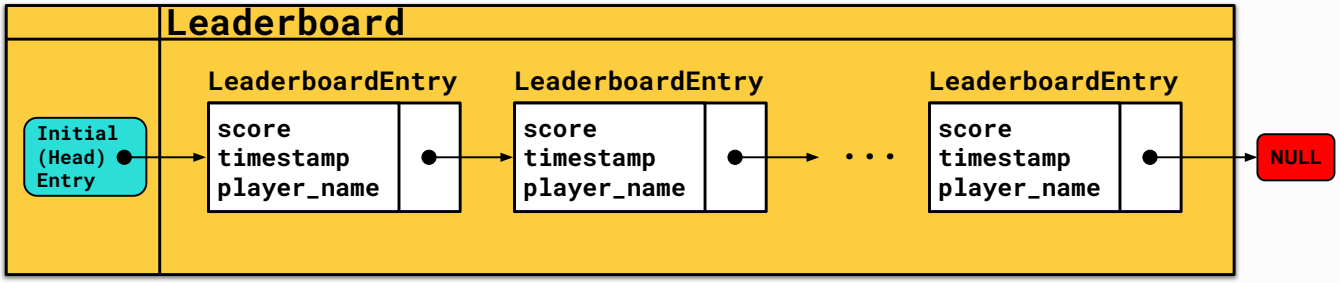
Moreover, the leaderboard should be printed to `STDOUT` in the following format:

```
Leaderboard
-----------
<#order>. <player_name> <score> <timestamp formatted as %H:%M:%S/%d.%m.%Y>
```

As an example:

```
Leaderboard
-----------
1. AsteroidBuster 40000 20:50:55/21.10.2024
2. StackOverthrower 1200 20:50:55/21.10.2024
...
```

The `Leaderboard` class has a pointer to the first (top) `LeaderboardEntry` named `head_leaderboard_entry`. It will be `NULL` if there are no highscores yet. Leaderboard must be stored as a linked list of `LeaderboardEntry` instances, such that each entry points to the next high score.



The leaderboard is designed to retain a maximum of ten all-time high scores, necessitating a dynamic implementation that can efficiently manage insertions and deletions while maintaining score order. As such, you are tasked with developing functionalities that correctly integrate new high scores into the leaderboard. This involves dynamically allocating memory for new entries and responsibly deallocating memory associated with those that are displaced from the top ten rankings. It's crucial to ensure that these operations preserve the leaderboard's integrity and its real-time reflection of player achievements.

2.2.4 Game Over

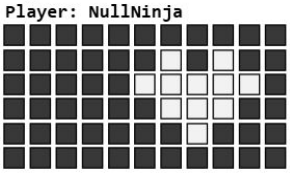
The game will continue progressing until one of the following termination conditions is met:

- The input file containing commands is fully read, leaving no further instructions to execute.
- The player's lives are exhausted, as each collision with an asteroid reduces the player's life count until it reaches zero.

The game concludes when the player either collides with an asteroid and loses their last life, or when there are no more commands to execute. At the game's end, the final grid layout, current score, and other relevant gameplay statistics are displayed. Additionally, if the player's score qualifies as a new high score, the leaderboard is updated accordingly.

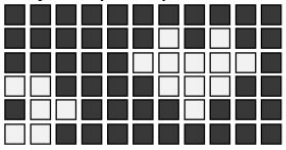
Output Format: Depending on the game-ending scenario, the program will generate distinct outputs to reflect the specific condition that caused the termination. Each output format should detail the current score, remaining lives, and the state of the grid, as illustrated in the example figure below.

```
GAME OVER!
Tick: 29
Lives: 0
Ammo: 7
Score: 26
High Score: 40000
Player: NullNinja
```



```
Leaderboard
-----------
1. AsteroidBuster 40000 20:50:55/21.10.2024
2. StackOverthrower 1200 20:51:23/20.10.2024
3. NullNinja 26 22:20:50/26.10.2024
```

```
GAME FINISHED! No more commands!
Tick: 20
Lives: 3
Ammo: 10
Score: 40
High Score: 40000
Player: SpaceExplorer
```



```
Leaderboard
-----------
1. AsteroidBuster 40000 20:50:55/21.10.2024
2. StackOverthrower 1200 20:51:23/20.10.2024
3. SpaceExplorer 40 22:21:42/26.10.2024
4. NullNinja 26 22:20:50/26.10.2024
```

2.3 Assignment Implementation Tasks and Requirements

In this section, we outline the classes and functions you are required to implement. Please ensure that your code adheres to the structure of the provided starter files, maintaining clarity, encapsulation, and consistency. Do not modify the names or signatures of any functions or member variables specified in the template. However, you may add additional functions or variables as needed.

2.3.1 `CelestialObject` Class

This class represents celestial objects, either asteroids or power-ups, within the game.

- **Constructor**:

```
CelestialObject(const vector<vector<bool>>& shape, ObjectType type, int
↪  start_row, int time_of_appearance)
```

- Initialize the object with its shape, type, starting row, and appearance time.

- **Copy Constructor**:

```
CelestialObject(const CelestialObject *other)
```

- Initialize the copy object from an existing object.

- **Function**:

  `void delete_rotations()`

  - Delete all old rotations of the celestial object after changes to its shape (free the dynamically allocated memory for them).

2.3.2 `AsteroidDash` Class

This class represents the main game logic and structure.

- **Constructor**:

```
AsteroidDash(const string &space_grid_file_name, const string
↪  &celestial_objects_file_name,
    const string &leaderboard_file_name, const string &player_file_name, const
    ↪  string &player_name)
```

  - Initializes the game by loading the space grid, player, celestial objects, and leaderboard using the specified input files.
- **Function**:

  `void print_space_grid() const`

  - Prints the space to `STDOUT`.
- **Function**:

  `void read_space_grid(const string &input_file)`

  - Reads and sets up the space grid from the input file, initializing the games playable area.
- **Function**:

  `void read_player(const string &player_file_name, const string &player_name)`

  - Reads the players information from a file and initializes the player within the grid.
- **Function**:

  `void read_celestial_objects(const string &input_file)`

  - Reads celestial objects from the specified file and sets up the linked list of celestial objects for the game.
- **Function**:

  `void update_space_grid()`

  - Updates the space grid each tick, managing the movement and collisions of the player, celestial objects, and projectiles.
- **Function**:

  `void shoot()`

  - Initiates a projectile from the players current position and adds it to the games active projectiles.
- **Destructor**:

  `~AsteroidDash()`

  - Cleans up dynamically allocated memory for celestial objects, player, and other game components.

2.3.3 `GameController` Class

Theis class is responsible for managing gameplay and interpreting commands from an input file.

- **Constructor**:

```
GameController(const string &space_grid_file_name, const string
↪  &celestial_objects_file_name,
    const string &leaderboard_file_name, const string &player_file_name, const
    ↪  string &player_name)
```

- Initializes the `AsteroidDash` game instance with the given input files, setting up the space grid, celestial objects, player, and leaderboard.

- **Function**:

```
void play(const string &commands_file)
```

- Executes gameplay by reading and interpreting commands from the specified file, updating the game state in each tick according to the commands provided.

- **Destructor**:

```
~GameController()
```

- Cleans up any dynamically allocated memory for the `AsteroidDash` game instance.

2.3.4 `Player` Class

The `Player` class represents the player's spacecraft and manages the player's actions, including movement and shooting.

- **Constructor**:

```
Player(const vector<vector<bool>> &shape, int row, int col, const string
↪  &player_name, int max_ammo = 10, int lives = 3)
```

- Initializes the player with the given spacecraft shape, starting position, player name, maximum ammo, and lives.

- **Function**:

```
void move_left()
```

- Moves the players spacecraft one cell to the left within grid boundaries.

- **Function**:

```
void move_right(int grid_width)
```

- Moves the players spacecraft one cell to the right within grid boundaries.

- **Function**:

```
void move_up()
```

- Moves the players spacecraft up one cell within grid boundaries.

- **Function**:

```
void move_down(int grid_height)
```

- Moves the players spacecraft down one cell within grid boundaries.

- **Destructor**:

```
~Player()
```

- Cleans up any resources allocated for the player.

2.3.5 `LeaderboardEntry` Class

This class represents a single entry on the leaderboard.

- **Constructor**:
  `LeaderboardEntry(unsigned long score, time_t lastPlayed, const string &playerName)`
  - Initialize a leaderboard entry with score, timestamp, and player name.

2.3.6 `Leaderboard` Class

This class manages leaderboard entries.

- **Function**:
  `void insert(LeaderboardEntry *new_entry)`
  - Insert a new entry, keeping the list in descending order by score. Limit the leaderboard to 10 entries.
- **Function**:
  `void read_from_file(const string& filename)`
  - Load the leaderboard from a file.
- **Function**:
  `void write_to_file(const string& filename)`
  - Write the current leaderboard to a file.
- **Function**:
  `void print_leaderboard()`
  - Print the leaderboard to the console.
- **Destructor**:
  `~Leaderboard()`
  - Clean up dynamically allocated memory for leaderboard entries.

Must-Use Starter Codes

You MUST use **this starter (template) code**. All headers and classes should be placed directly inside your **zip** archive.

Grading Policy

- No memory leaks and errors: 10%
  - No memory leaks: 5%
  - No memory errors: 5%
- Implementation of the game: 80%
  - Proper game grid and player initialization: 5%

  - Correct implementation of the multilevel linked list structure for game objects and their rotations, and related operations: 15%
  - Correct implementation of the player movements: 5%
  - Correct implementation of shooting and resulting object shape and rotations update: 15%
  - Correct implementation of objects movements and collisions: 7.5%
  - Correct implementation of power-up handling: 5%
  - Correct implementation of the scoring mechanism: 7.5%
  - Correct implementation of the leaderboard linked list and related operations: 15%
  - Proper game termination: 5%
- Output tests: 10%

Important Notes

- Do not miss the deadline: **Friday, 15.11.2024 (23:59:59)**.
- Save all your work until the assignment is graded.
- The assignment solution you submit must be your original, individual work. Duplicate or similar assignments are both going to be considered as cheating.
- You can ask your questions via Piazza (https://piazza.com/hacettepe.edu.tr/fall2024/bbm203), and you are supposed to be aware of everything discussed on Piazza.
- You must test your code via **Tur^{6}Bo Grader** https://test-grader.cs.hacettepe.edu.tr/ (**does not count as submission!**).
- You must submit your work via https://submit.cs.hacettepe.edu.tr/ with the file hierarchy given below:
  - **b<studentID>.zip**
    - AsteroidDash.h <FILE>
    - AsteroidDash.cpp <FILE>
    - CelestialObject.h <FILE>
    - CelestialObject.cpp <FILE>
    - GameController.h <FILE>
    - GameController.cpp <FILE>
    - LeaderboardEntry.h <FILE>
    - LeaderboardEntry.cpp <FILE>
    - Leaderboard.h <FILE>
    - Leaderboard.cpp <FILE>
    - Player.h <FILE>
    - Player.cpp <FILE>
- **You MUST use this starter code**. All classes should be placed directly in your **zip** archive.
- This file hierarchy must be zipped before submitted (not .rar, only .zip files are supported).

Run Configuration

Here is an example of how your code will be compiled (note that instead of `main.cpp` we will use our test files):

```
$ g++ -std=c++11 -g  main.cpp AsteroidDash.h AsteroidDash.cpp CelestialObject.h
    CelestialObject.cpp GameController.h GameController.cpp LeaderboardEntry.h
↪   LeaderboardEntry.cpp Leaderboard.h Leaderboard.cpp Player.h Player.cpp -o
↪   AsteroidDash
```

Or, you can use the provided `Makefile` or `CMakeLists.txt` within the sample input to compile your code:

```
$ make
```

or

```
$ mkdir AsteroidDash_build
$ cmake -S . -B AsteroidDash_build/
$ make -C AsteroidDash_build/
```

After compilation, you can run the program as follows:

```
$ ./AsteroidDash space_grid.dat celestial_objects.dat player.dat commands.dat
↪   leaderboard.txt AsteroidBuster
```

Academic Integrity Policy

All work on assignments **must be done individually**. You are encouraged to discuss the given assignments with your classmates, but these discussions should be carried out in an abstract way. That is, discussions related to a particular solution to a specific problem (either in actual code or in pseudocode) **will not be tolerated**. In short, turning in someone else's work (including work available on the internet), in whole or in part, as your own will be considered as **a violation of academic integrity**. Please note that the former condition also holds for the material found on the web as everything on the web has been written by someone else.

The submissions will be subjected to a similarity check. Any submissions that fail the similarity check will not be graded and will be reported to the ethics committee as a case of academic integrity violation, which may result in the suspension of the involved students.

Bonus Challenge With Tiny Awards

The first **three** students who

- successfully complete the assignment (get 100 points on the **Tur^{6}Bo Grader**), and
- develop an interactive interface for their game

will be rewarded with tiny awards.

Here is our implementation in action: Gameplay Recording