Kingdomino: Enter the Virtual World

Hai Duong, Tran
CSE, Frankfurt University of Applied Sciences
Frankfurt am Main, Germany
hai.tran2@stud.fra-uas.de

Pham Minh Tuan, Bui
CSE, Frankfurt University of Applied Sciences
Frankfurt am Main, Germany
pham.bui@stud.fra-uas.de

***Abstract*—This paper presents a digital adaptation of the board game Kingdomino, developed using Java and the LibGDX framework. Our implementation leverages an event-driven architecture, a flood-fill algorithm for scoring, and shader-based rendering to enhance the user experience. We explore key aspects of game logic, including tile placement validation, turn-based mechanics, and dynamic UI interactions. The paper details the design choices, algorithms, and modular structure of the system, ensuring efficiency and scalability. Additionally, we evaluate game performance through experimental results and discuss future improvements, such as AI integration and multiplayer capabilities.**

***Index Terms*—Kingdomino, boardgame, implementation, game development, OOP**

I. Introduction

Kingdomino [1] is a strategic tile placement game where players act as lords expanding their kingdoms. The objective is to construct the most prosperous territory by selecting and placing tiles that represent various terrain types, such as wheat fields, lakes, and mountains. Each tile comprises two sections that must be connected to the existing kingdom based on matching terrain types.

This paper presents a digital adaptation of Kingdomino, developed using Java and the LibGDX framework. Our implementation leverages an event-driven architecture, a flood-fill algorithm for scoring, and shader-based rendering to enhance the user experience. We explore key aspects of game logic, including tile placement validation, turn-based mechanics, and dynamic UI interactions. The paper details the design choices, algorithms, and modular structure of the system, ensuring efficiency and scalability. Additionally, we evaluate game performance through experimental results and discuss future improvements, such as AI integration and multiplayer capabilities.[1]

The paper is structured as follows: Section II describes the problem and game mechanics. Section III reviews related work and algorithms. Section IV discusses teamwork and collaboration tools. Section V outlines the proposed approaches, including the event-driven architecture and input handling. Section VI and Section VII detail the implementation, including game states, tile placement, scoring mechanisms, graphical enhancements, and visual effects. Section VIII presents experimental results and performance analysis. Finally, Section IX concludes the paper and suggests future work.

[1]The source code for the Kingdomino digital adaptation can be found on GitHub at https://github.com/fuisl/kingdomino.

II. Problem Description

The primary goal in Kingdomino is to construct the most prestigious kingdom by strategically placing tiles within a constrained 5×5 grid. Players must explore various terrain types—fields, lakes, mountains, forests, meadows, and swamps—and connect them to their existing kingdom while adhering to specific placement rules. Each tile consists of two sections, and at least one section must match the terrain type of an adjacent tile. Crowns on tiles act as multipliers to the score of connected terrain groups, encouraging players to prioritize valuable combinations.

Players compete for tiles through a selection mechanism that balances high-value tiles with the consequence of selecting later in subsequent rounds. This creates an optimization problem where players must maximize immediate gains while strategically positioning themselves for future moves. The game concludes when each player has filled their grid or can no longer place tiles, with scoring determining the winner based on terrain connectivity and crown placements.

Fig. 1. Kingdomino Board Game.

A. Formal Description

The game involves the following components and constraints:

1) Components:

- **Tiles**: Each tile consists of two sections, each representing one of the six terrain types (fields, lakes, mountains, forests, meadows, and swamps). Certain tiles include crowns, which serve as score multipliers.
- **Kingdom**: A 5×5 grid where tiles are placed. Each player starts with a central tile (wild type) and builds outward.

- **Selection Order**: Players select tiles in descending order of tile value (higher numbers first) and use their selection to determine the order in subsequent rounds.

2) Rules:

- **Placement**:
  - Tiles must connect to at least one adjacent tile with the same terrain type (horizontally or vertically).
  - The grid is limited to 5×5 dimensions; any tile that cannot be placed is discarded.
- **Scoring**:
  - Points are calculated as the product of the number of connected tiles of the same terrain and the number of crowns within the connected group.
  - Bonuses include:
    - +10 points for placing the central castle at the grid's center.
    - +5 points for completing a full grid.
- **Game Variants**:
  - A 7×7 grid variant for advanced play.
  - Multi-round gameplay (Dynasty mode) with cumulative scores.

3) Objective: Maximize the total score by constructing a kingdom that balances large connected terrain groups with crown placements, while competing against other players for optimal tile selection.

B. Examples of Gameplay

Gameplay is explain in Appendix A.

III. Related Work

Board games like Carcassonne [2] and Patchwork [3] offer valuable insights into the mechanics of tile placement and scoring, making them relevant to our implementation of Kingdomino. Carcassonne challenges players to build cities, roads, and fields by placing tiles based on terrain type, a mechanic that closely aligns with Kingdomino's terrain-matching rules. Similarly, Patchwork emphasizes the efficient use of a constrained grid, much like Kingdomino's 5×5 board, where players must optimize placement for maximum scoring. These games demonstrate how simple mechanics can result in complex decision-making, a feature we aim to replicate in our project.

In addition to inspiration from traditional board games, the implementation of Kingdomino leverages several key computational and architectural techniques. The scoring mechanism, for instance, utilizes a flood-fill algorithm [4] to evaluate the connected terrain groups and their associated scores. This algorithm, commonly employed in image processing and graph traversal [5], provides an efficient way to traverse and calculate properties of contiguous regions. Its application in Kingdomino ensures accurate and performant scoring calculations, even as the board becomes increasingly complex during gameplay.

The game is designed using an event-driven architecture [4], [6], where interactions, where interactions such as tile placement and scoring updates are managed asynchronously through an EventManager. This approach is prevalent in modern game frameworks and engines, such as LibGDX, Unity, and Unreal Engine, and ensures a clear separation between game logic and user input. By decoupling these components, the design achieves improved modularity and scalability, enabling future enhancements such as AI integration or multiplayer features. The intention was to implement Entity-Component-System (ECS) architecture [7] to further improve the game's performance and scalability. But we not strictly follow this architecture for flexibility and simplicity.

Furthermore, the aesthetic design of Kingdomino draws from pixel-art-inspired games, including Balatro [8], which use vibrant colors and minimalistic textures to create a visually engaging experience. This style has been adopted to maintain the simplicity of the board game while enhancing the digital adaptation with a modern and nostalgic visual appeal.

Finally, the use of the LibGDX framework [9] streamlines development, particularly in rendering, asset management, and input handling. By leveraging tools such as TextureAtlas [10] for managing game assets and InputMultiplexer for handling player interactions, the framework supports an efficient implementation of Kingdomino's mechanics and aesthetics. This integration aligns with industry-standard practices for creating scalable, interactive applications.

Incorporating these approaches and inspirations, the implementation of Kingdomino aims to balance simplicity, efficiency, and engagement, ensuring a faithful and enjoyable digital representation of the original board game.

IV. Teamwork

We divided the project into backend (game logic) and frontend (GUI), using GitHub ecosystem for tracking and communication. We also followed SCRUM planning techniques. (Fig 2).

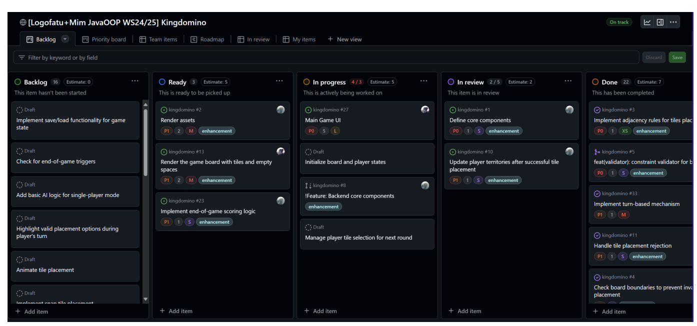


Fig. 2. Kanban Board implements SCRUM planning technique.

A. Roles and Responsibilities

Hai Duong Tran handled backend logic, including tile placement rules, scoring, and game flow. He also integrated the flood-fill algorithm and worked on shaders and effects. Pham Minh Tuan Bui focused on the GUI, rendering game elements with LibGDX, and work on creating our original assets for this project. He also worked on visual aesthetics. Both members contributed to the success of this project.

We collaborated on various aspects not just within our scope and responsibilities. We held regular meetings, and conducted code reviews to maintain quality and consistency. This approach enhanced productivity and understanding of the project.

B. Collaboration Tools and Workflow

Most of our communication happened on Github Issues and Pull Requests. The tools are used for tracking and code reviews. Each member created branches for their tasks, ensuring main branch stability. Code reviews were conducted via GitHub Pull Requests (Fig 4), maintaining code quality and facilitating knowledge sharing. For communication, we used Messenger and in-person meetings, documenting progress via commit messages and digital drafts.

C. Gitflow Workflow

We adopted the Gitflow workflow for version control, which is an industry-standard branching model for Git. This approach helps manage the development process by organizing branches and ensuring a clear path from development to production.

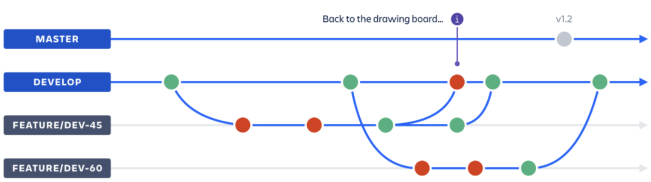


Fig. 3. Gitflow Workflow.

The Gitflow workflow consists of the following branches:

- **main**: The main branch containing production-ready code.
- **dev**: The integration branch for features and fixes.
- **feature/***: Branches for developing new features.
- **release/***: Branches for preparing releases.
- **hotfix/***: Branches for quick fixes to production code.

This workflow allows for parallel development, efficient collaboration, and a structured release process.

V. Proposed Approaches

A. Event-Driven Architecture

Kingdomino's digital adaptation uses an event-driven architecture to manage user interactions, game state transitions, and visual effects efficiently. This approach decouples game logic, input handling, and rendering, improving performance, modularity, and maintainability.

B. Core Components

1) Event Class: The `Event` class encapsulates actions that can be executed immediately or after a delay. It supports different types of triggers:

- **IMMEDIATE**: Executes the event as soon as it is created.

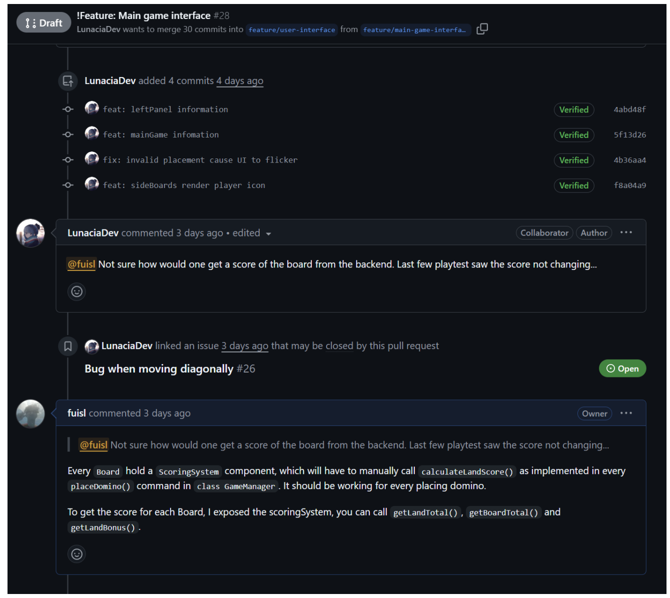


Fig. 4. A conversation in one Pull Request.

- **BEFORE**: Executes an action and then enforces a delay before completing.
- **AFTER**: Executes an action only after a specified delay.
- **CONDITION**: Executes when a certain condition is met.
- **EASE**: Implements smooth transitions for visual effects.

Each event has attributes such as blocking, blockable, complete, and delay, allowing for controlled execution within the game loop.

2) EventManager: The `EventManager` acts as the central execution hub for all game-related events. It maintains multiple concurrent event queues, categorized into:

- **Base**: Core gameplay logic (e.g., turn transitions, score updates).
- **Input**: Player interactions (e.g., movement, rotation, placement).
- **Background**: Visual effects and animations.
- **Sound**: Audio effects (e.g., movement, tile placement, invalid moves).
- **Other**: Miscellaneous actions.

The event manager ensures concurrency safety using `ConcurrentLinkedQueue`, preventing race conditions when processing multiple asynchronous actions.

3) GameTimer: The `GameTimer` class tracks real-time, total elapsed time, and background animation timing. It synchronizes event execution by accumulating delta time (dt) and updating event processing intervals accordingly. The timer provides a consistent timing reference for executing delayed events and easing functions.

4) Ease: The `Ease` class facilitates smooth animations for background effects, screen transitions, and UI enhancements. It supports different interpolation methods:

- **LERP (Linear Interpolation)**: Gradually changes a value over time.

- **ELASTIC**: Simulates spring-like motion for dynamic effects.
- **QUAD**: Creates a smooth acceleration effect.

This is particularly useful for animations such as screen shaking when an invalid move is made, or color transitions when switching player turns.

C. Input Handling

Kingdomino implements manual input event processing instead of relying on LibGDX's built-in event loop. The input handling is divided into two main components: `BoardInputController` and `BoardInputProcessor`, which handle input from controllers and keyboards, respectively. These components translate user inputs into structured events that are processed by the system.

1) BoardInputController: The `BoardInputController` class handles input from game controllers. It captures axis movements and button presses, translating them into actions such as moving, rotating, placing, or discarding a domino. The controller input is processed with a dead zone to ignore small, unintended movements, ensuring precise control.

```java
// filepath: /e:/projects/kingdomino/core/src/main/java/dev/kingdomino/game/BoardInputController.java
public class BoardInputController extends AbstractController {
    // ...existing code...
    @Override
    public boolean axisMoved(Controller controller, int axisCode, float value) {
        // ...existing code...
        Action action = translateAxisToAction(axisX, axisY);
        if (action != Action.NONE) {
            return boardInputHandler.keyDown(action); // Trigger movement
        } else {
            return boardInputHandler.keyUp(action); // Stop movement
        }
    }
    // ...existing code...
}
```

2) BoardInputProcessor: The `BoardInputProcessor` class handles input from the keyboard. It maps key presses to actions such as moving, rotating, placing, or discarding a domino. This class ensures that keyboard inputs are correctly translated into game actions.

```java
// filepath: /e:/projects/kingdomino/core/src/main/java/dev/kingdomino/game/BoardInputProcessor.java
public class BoardInputProcessor extends AbstractInputProcessor {
    // ...existing code...
    @Override
    public boolean keyDown(int keycode) {
        GameManager.setInputDevice(InputDevice.KEYBOARD);
        return boardInputHandler.keyDown(translateKeycodeToAction(keycode));
    }
    // ...existing code...
}
```

3) Input Handlers: The `BoardInputHandler` and `DraftInputHandler` classes are responsible for translating the inputs received from the `BoardInputController` and `BoardInputProcessor` into signals that the system can process. These handlers manage the state of the game and ensure that inputs are processed correctly based on the current game state.

```java
// filepath: /e:/projects/kingdomino/core/src/main/java/dev/kingdomino/game/BoardInputHandler.java
public class BoardInputHandler {
    // ...existing code...
    public boolean keyDown(Action action) {
        if (gameManager.getCurrentState() != GameManager.GameState.TURN_PLACING) {
            return false;
        }
        update(); // update states
        Event e = null;
        switch (action) {
            case MOVE_UP:
                // ...existing code...
            case MOVE_DOWN:
                // ...existing code...
            case MOVE_LEFT:
                // ...existing code...
            case MOVE_RIGHT:
                // ...existing code...
            case ROTATE_CLOCKWISE:
                // ...existing code...
            case ROTATE_COUNTERCLOCKWISE:
                // ...existing code...
            case PLACE_DOMINO:
                // ...existing code...
            case DISCARD_DOMINO:
                // ...existing code...
            default:
                break;
        }
        if (e != null) {
            eventManager.addEvent(e.copy(), "input", false);
            updated = true;
        }
        return true; // returning true indicates the event was handled
    }
    // ...existing code...
}
```

```java
// filepath: /e:/projects/kingdomino/core/src/main/java/dev/kingdomino/game/DraftInputHandler.java
public class DraftInputHandler extends AbstractInputProcessor {
    // ...existing code...
    public boolean keyDown(Action action) {
        if (gameManager.getCurrentState() != GameManager.GameState.TURN_CHOOSING) {
            updated = true;
            return false;
        }
        Event e = null;
        switch (action) {
            case MOVE_UP:
                // ...existing code...
            case MOVE_DOWN:
                // ...existing code...
            case SELECT_DOMINO:
                // ...existing code...
```

```
            default:
                break;
        }
        if (e != null) {
            eventManager.addEvent(e, "base", false);
            updated = true;
        }
        return true;
    }
    // ...existing code...
}
```

4) Key Features of Input Handling:

- **Deferred Execution**: Input actions are queued and processed asynchronously.
- **State-Based Processing**: Input is processed only in relevant game states.
- **Feedback Integration**: Invalid actions trigger audio and visual feedback.

5) Example: Handling Movement Events: When a player moves a domino using the keyboard or controller:

1) Input is captured (WASD keys or joystick).
2) `BoardInputHandler` checks validity (e.g., ensuring the move is within bounds).
3) If valid, an event is created and queued:

```
Event moveEvent = new Event(
    TriggerType.BEFORE, true, true, 0.15f, // Delay for smooth input
    () -> currentDomino.moveDomino(Direction.UP),
    null, null, null);
eventManager.addEvent(moveEvent, "input", false);
```

4) The event executes smoothly without blocking the main loop.
5) If the move is invalid, audio and visual feedback are triggered:

```
audioManager.playSound(AudioManager.SoundType.CANCEL);
BackgroundManager.screenShake();
```

D. User Interface

1) User Interface Layout Design: We chose to design the user interface based on tables, where each component or table is placed within a cell as demonstrated in 5.

We mimic how the game state of Kingdomino is distributed in real life by having the current player's game board being placed prominently in the center of the screen, whilst other informations that the player can find are placed to the side. We also decided to show how many points each player is currently scoring with their board as well, because it could enable more strategic choice if players know exactly how their board is doing compared to other, instead of focusing on picking the tile that would yield the most point for their board.

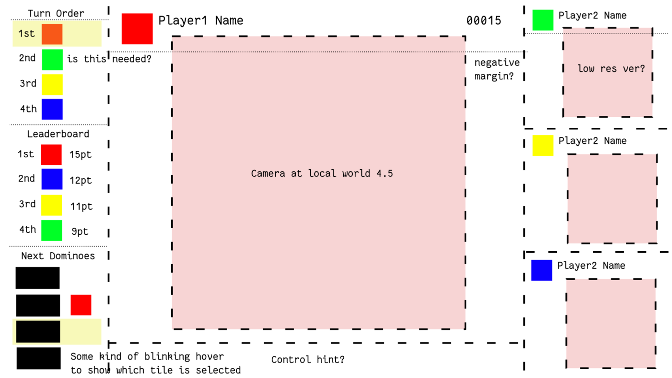


Fig. 5. First UI design prototype.

VI. Implementation Details

A. Game States

In the implementation of Kingdomino, the game progresses through a series of well-defined states. Each state represents a distinct phase of the game, ensuring a structured flow from the beginning to the end. The primary game states include:

- **Init**: The initial state where the game components are set up and initialized.
- **Setup**: This state involves preparing the game board, shuffling tiles, and setting up players for the game.
- **Turn Start**: Marks the beginning of a player's turn, where the current player and available tiles are determined.
- **Turn Placing**: The state where the player places their selected tile on the board according to the game rules.
- **Turn Choosing**: The state where the player selects a tile for the next round.
- **Turn End**: Concludes the current player's turn and transitions to the next player.
- **Game Over**: The final state when the game ends, and no more moves can be made.
- **Results**: Displays the final scores and determines the winner based on the game rules.

Each state transitions to the next based on player actions and game logic. The following diagram illustrates the state transitions:

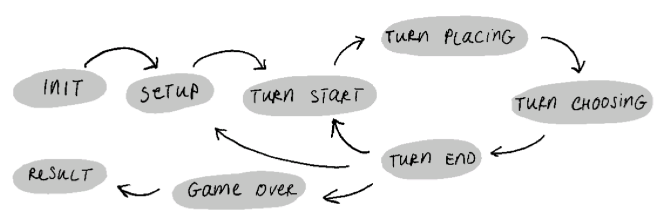


Fig. 6. Game State Transitions.

B. Game Units and High-Level Architecture

Tile is the essential, smallest accessible unit for this game. Tile holds 2 most important attributes: TerrainType and Number of Crowns.

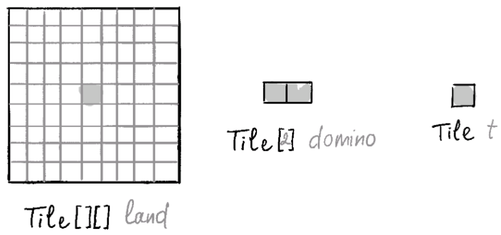


Fig. 7. Units of Tile.

Board is 2-dimensional array of type Tile. Though Board has more attributes and methods which supports interactions between GameManager and Board, Tile[][] is where Tiles being store and perform logic on.

On the other hand, Domino is an interface between human player and Board. It holds informations about location, val idator, etc., which related to action that can be performe on Board (like moving, rotating or placing Domino). Th actual Domino will not be placed in Board, only the two Tile that composes Domino would be added. Domino will then b discarded.

C. Tile Placement and Board Validation

1) Domino and DominoController: The `Domino` clas represents a domino in the game, consisting of two tiles an a controller. It provides methods to rotate, move, and se the position of the domino. The `DominoController` clas handles the rotation and placement of dominos, maintainin the state of the tiles and their positions.

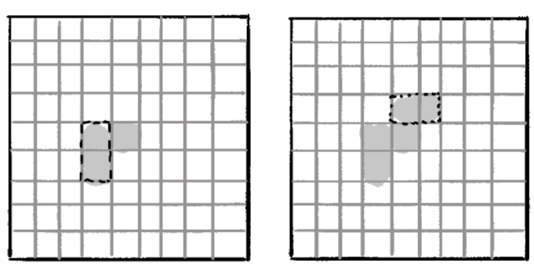


Fig. 8. Placing Domino.

```
// filepath: /e:/projects/kingdomino/core/src/main/
    java/dev/kingdomino/game/Domino.java
public class Domino {
    // ...existing code...
    public void rotateDomino(boolean clockwise,
    boolean shouldOffset) {
        dominoController.rotateDomino(clockwise,
    shouldOffset);
    }
    // ...existing code...
}
```

```
// filepath: /e:/projects/kingdomino/core/src/main/
    java/dev/kingdomino/game/DominoController.java
public class DominoController {
    // ...existing code...
    public void rotateDomino(boolean clockwise,
    boolean shouldOffset) {
        lastRotationIndex = rotationIndex;
        rotationIndex = (rotationIndex + (clockwise
    ? 1 : 3)) % 4;

        // rotate the 2nd Tile with 1st Tile as
    center
        tileRotator.rotate(posTileA, posTileB,
    rotationIndex, shouldOffset);

        lastAction = 1;
    }
    // ...existing code...
}
```

2) TileRotator: The `TileRotator` class is responsible for rotating tiles around a center position. It uses a predefined set of directions to determine the new position of the tile after rotation.

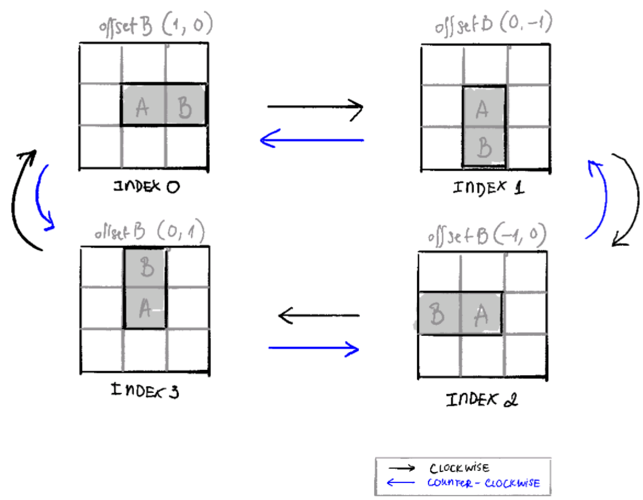


Fig. 9. Rotate Domino by moving TileB relative to TileA.

The rotation logic for the domino can be described as follows:

If rotating clockwise: $\boxed{\text{index} = (\text{index} + 1) \mod 4}$

If rotating counter-clockwise: $\boxed{\text{index} = (\text{index} + 3) \mod 4}$

```
// filepath: /e:/projects/kingdomino/core/src/main/
    java/dev/kingdomino/game/TileRotator.java
public class TileRotator {
    // ...existing code...
    public void rotate(Position center, Position
    tilePos, int rotationIndex, boolean shouldOffset
    ) {
        Position newPos = center.add(directions[
    rotationIndex]);
        tilePos.set(newPos);
    }
    // ...existing code...
}
```

3) Tile Placement Validation: The `TileValidator` class is responsible for validating the placement of tiles on the game board. It ensures that tiles are placed within bounds and are free from occupation. The following code snippet shows the implementation of the `TileValidator` class:

```
// filepath: /e:/projects/kingdomino/core/src/main/
    java/dev/kingdomino/game/TileValidator.java
public class TileValidator {
    // ...existing code...
    public boolean isTilePlaceable(Tile tile, int x,
     int y) {
        return isTileFree(x, y) && isTileWithinBound
    (x, y);
    }
    // ...existing code...
    public boolean isTileWithinBound(int x, int y) {
        if (x < minX || x > maxX) {
            if (abs(x - minX) + 1 > size || abs(x -
    maxX) + 1 > size) {
                return false;
            }
        }
        if (y < minY || y > maxY) {
            if (abs(y - minY) + 1 > size || abs(y -
    maxY) + 1 > size) {
                return false;
            }
        }
        return true;
    }
    // ...existing code...
    public boolean isTileFree(int x, int y) {
        if (isTileWithinLand(x, y)) {
            return land[y][x] == null;
        }
        return false;
    }
    // ...existing code...
}
```

D. Scoring Mechanism

The flood-fill algorithm is a computer graphics algorithm used to determine the area connected to a given node in a multi-dimensional array. It is commonly used in tools like paint bucket in graphics editors to fill bounded areas with color.

In the context of Kingdomino, the flood-fill algorithm is utilized to calculate the score by identifying and evaluating connected groups of the same terrain type. The algorithm traverses the game board, starting from a given tile, and recursively explores all adjacent tiles of the same terrain type. This allows the game to efficiently compute the size of each connected terrain group and apply the scoring rules based on the number of crowns within these groups.

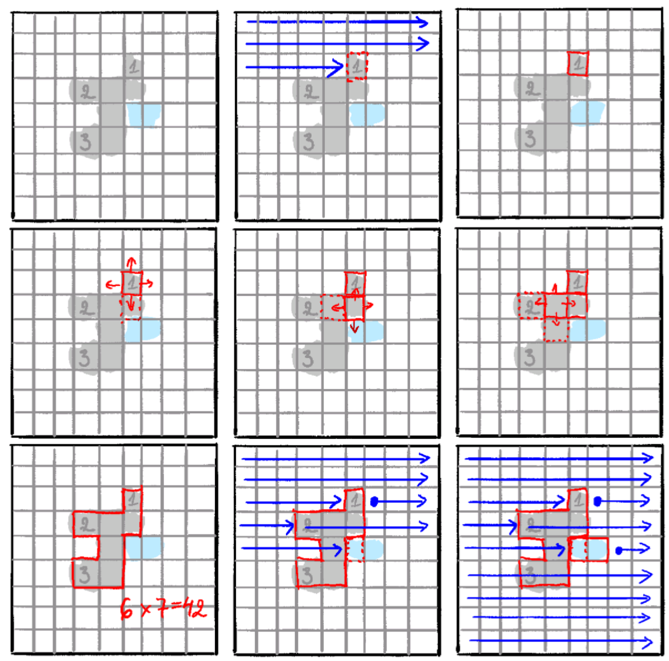
Below is a pseudocode representation of the recursive 4-way flood-fill algorithm:

This algorithm is in its original form. For Kingdomino with multiplicative scoring system, we need to modify the algorithm to calculate both the spanned region (area) and also count the number of crowns within the connected group. See Listing 1 for the implementation details.

To better illustrate the concept of flood-fill, consider the following example with a 9×9 grid:

Input: Tile
Output: Flood-filled region
if *Tile is not inside Region* **then**
    return;
end
Set the node;
Perform Flood-fill one step to the south of Tile;
Perform Flood-fill one step to the north of Tile;
Perform Flood-fill one step to the west of Tile;
Perform Flood-fill one step to the east of Tile;
return;

Algorithm 1: Flood-fill Algorithm



In this example, the algorithm starts at the selected tile (highlighted in red). It then recursively explores all adjacent tiles of the same terrain type, marking them as visited and counting the number of crowns within the connected group. The process continues until all connected tiles of the same terrain type have been evaluated, resulting in the final score for that terrain group.

Below is the actual implementation of the flood-fill algorithm in Kingdomino:

```
    private int floodFill(int x, int y, TerrainType
    terrain, boolean[][] visited) {
        if (x < 0 || x >= land.length || y < 0 || y
    >= land.length) {
            return 0;
        }

        if (land[y][x] == null || visited[y][x] ||
    land[y][x].getTerrain() != terrain) {
            return 0;
        }


        totalCrown += land[y][x].getCrown();
        visited[y][x] = true;
```

```
        return 1 +
                floodFill(x + 1, y, terrain, visited) +
                floodFill(x - 1, y, terrain, visited) +
                floodFill(x, y + 1, terrain, visited) +
                floodFill(x, y - 1, terrain, visited);
    }
```

Listing 1. Flood-Fill Algorithm

E. UML/Class Diagram

F. Used Libraries and Environment

VII. Graphical Enhancement and Visual Effects

A. Visual and Effects

Kingdomino's digital adaptation incorporates various visual and audio effects to enhance the user experience, making the game feel more responsive and engaging. Inspired by the talk "Juice it or lose it" by Martin Jonasson & Petri Purho, we aimed to create a game that feels alive and responds to player actions with minimal input.

1) Background Effects: The `BackgroundManager` class manages dynamic background effects using shader-based rendering. Key features include:

- **Color Transitions**: Smooth transitions between different background colors to indicate changes in game states or player turns.
- **Spinning Effects**: Controlled spinning of the background to add a sense of motion and excitement during gameplay.
- **Screen Shake**: A subtle screen shake effect to provide feedback for significant events, such as invalid moves or special actions.

2) Audio Effects: The `AudioManager` class handles the game's audio effects, providing immediate feedback for player actions. Key features include:

- **Background Music**: Continuous background music that sets the tone for the game.
- **Sound Effects**: Various sound effects for actions such as tile placement, selection, rotation, and scoring in order to provide immediate feedback for each action the player take.

3) Shader-Based Rendering: Kingdomino leverages shader-based rendering using OpenGL Shader language to enhance visual effects and provide a more immersive experience. Two primary shaders are used: the CRT shader and the background shader. The shader codes are an adaptation of existing shaders of Balatro [8] which adapted from Löve2D written in Lua to LibGDX Framework using with Java.

a) CRT Shader: The CRT shader simulates the appearance of an old CRT monitor, adding effects such as distortion, scanlines, noise, and bloom. It consists of a vertex shader and a fragment shader.

- **Vertex Shader (crt.vert)**: Applies transformations to simulate a parallax effect based on screen scale.
- **Fragment Shader (crt.frag)**: Implements various visual effects, including barrel distortion, edge feathering, glitch offsets, chromatic aberration, scanlines, and noise overlay.

b) Background Shader: The background shader creates dynamic and visually appealing backgrounds with effects like spinning and color transitions.

- **Vertex Shader (background.vert)**: Passes texture coordinates to the fragment shader.
- **Fragment Shader (background.frag)**: Applies pixelation, swirl, and paint effects based on time and spin parameters, along with color blending for a vibrant background.

These visual and audio enhancements contribute to a more engaging and immersive gaming experience, making Kingdomino not just a digital adaptation but a lively and responsive game that captivates players.

B. Game Textures

Aside from the image of the King which was taken by Adam Carmichael [11] and the font Pixelify Sans [12], all of the game's texture are hand-drawn. The tile are created from a 48x48 pixel square of the color representative of its terrain type, before applying Perlin Noise [13] so as to reduce the monotonousness of the texture. We delibrately choose low texture size as large, detailed texture would become inelligible by the CRT Shader.

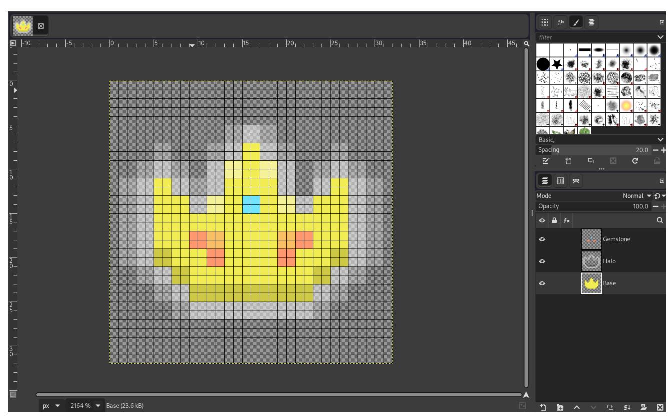


Fig. 10. Original design file of the Crown.

VIII. Experimental Results

A. Performance Analysis

framerate, input lag event-driven vs. sync update

B. Correctness and Scalability

C. UI/Gameplay Smoothness

IX. Conclusions and Future Work

Lorem ipsum dolor sit amet, consectetur adipiscing elit.

A. What We Learned

B. Future Development

APPENDIX A
GAMEPLAY MECHANIC

REFERENCES

[1] Wikipedia contributors, "Kingdomino — Wikipedia, the free encyclopedia," 2024, [Online; accessed 19-January-2025]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Kingdomino&oldid=1243538299

[2] ——, "Carcassonne (board game) — Wikipedia, the free encyclopedia," 2024, [Online; accessed 19-January-2025]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Carcassonne_(board_game)&oldid=1254238483

[3] ——, "Patchwork (board game) — Wikipedia, the free encyclopedia," 2024, [Online; accessed 19-January-2025]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Patchwork_(board_game)&oldid=1212675183

[4] ——, "Flood fill — Wikipedia, the free encyclopedia," 2024, [Online; accessed 19-January-2025]. [Online]. Available: https://en.wikipedia.org/wiki/Flood_fill#Stack-based_recursive_implementation_(four-way)

[5] ——, "Graph traversal — Wikipedia, the free encyclopedia," 2024, [Online; accessed 19-January-2025]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Graph_traversal&oldid=1250833600

[6] ——, "Event-driven architecture — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=Event-driven_architecture&oldid=1262812523, 2024, [Online; accessed 19-January-2025].

[7] ——, "Entity component system — Wikipedia, the free encyclopedia," 2024, [Online; accessed 19-January-2025]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Entity_component_system&oldid=1265223349

[8] ——, "Balatro (video game) — Wikipedia, the free encyclopedia," 2025, [Online; accessed 19-January-2025]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Balatro_(video_game)&oldid=1270307614

[9] "Libgdx: A java game development framework," https://libgdx.com/.

[10] Wikipedia contributors, "Texture atlas — Wikipedia, the free encyclopedia," 2024, [Online; accessed 19-January-2025]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Texture_atlas&oldid=1256529720

[11] Adam Carmichael, "KingDominoProject," Accessed: 01-February-2025. [Online]. Available: https://github.com/Adam-Carmichael/KingDominoProject

[12] Stefie Justprince, "Pixelify Sans," Accessed: 01-February-2025. [Online]. Available: https://github.com/eifetx/Pixelify-Sans

[13] Wikipedia contributors, "Perlin noise — Wikipedia, the free encyclopedia," Accessed: 01-February-2025. [Online]. Available: https://en.wikipedia.org/wiki/Perlin_noise