МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«ВЯТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(ВятГУ)

ОТЧЁТ ПО УЧЕБНОЙ ПРАКТИКЕ

Черкасов Александр Андреевич

(Ф.И.О. обучающегося)

09.03.01 Информатика и вычислительная техника

Инженерия программного и аппаратного обеспечения

(направление подготовки (специальность), направленность(профиль))

Место прохождения практики: *ФГБОУ ВО «ВятГУ», кафедра ЭВМ*

(наим. организации, структурного подразделения организации)

Итоговая оценка: _____

Руководитель
практики от университета *12.07.2025* _____ _____

(дата) *(подпись)* *(Ф.И.О.)*

Киров, 2025 г.

Содержание

Введение

Учебная практика проходила на базе ФГБОУ ВО «Вятский государственный университет» в период с 30.06.2025 по г. 13.07.2025. Цель практики: закрепление знаний, умений и навыков полученных на первом курсе.

Задачи практики:

- Решение общих задач научно-исследовательского характера, предполагающих поиск ответа ближайшего к оптимальному в условиях ограничения временных ресурсов
- Индивидуальное задание по разработке графического приложения

Общаяя часть

В данном разделе рассматриваются вопросы, связанные с прохождением общей для всех обучающихся части практики.

Первое задание: Обнаружение чёрных квадратов на изображении

В заданном графическом файле содержится изображение с чёрными квадратами, обладающими следующими свойствами:

- Фиксированный размер одного чёрного квадрата: 20×20 пикселей
- Псевдослучайное расположение с ограничением: площадь пересечения с существующими квадратами $\leq 30\%$ при добавлении
- Наличие искажений и шумов

Алгоритм:

1. **Предобработка:** Конвертация в градации серого и бинаризация с пороговым значением для выделения чёрных объектов
2. **Поиск контуров:** Примененение алгоритма обнаружения границ.
3. **Фильтрация квадратов:** Аппроксиммация контуров полигонами, отбор объектов с 4 вершиами, проверка соотношения сторон (0.9-1.1) и фильтрация по размеру квадрата ($20 \times 20\pm$ погрешность)
4. **Визуализация:** Отрисовка рамок вокруг обнаруженных квадратов с нумерацией

Второе задание: Поиск делителей большого числа

Цель: Найти все положительные делители большого числа (≤ 10000).

Алгоритм:

1. **Факторизация:**
   - Решето Эратосфена для простых чисел ($\leq 10^6$)
   - Алгоритм Полларда-Ро для больших делителей
   - Тест Миллера-Рабина для проверки простоты
2. **Генерация делителей:**
   - Для чисел с ($\leq 10^4$) делителей: полный перебор комбинаций простых множителей
   - Для чисел с ($> 10^4$) делителей: приоритетная очередь для генерации наименьших 9999 делителей
3. **Оптимизации:**
   - Длинная арифметика с базой 10^9 (BigInt)
   - Алгоритм Карацубы для быстрого умножения
   - Модульное возведение в степень

Третье задание: Раскраска графа

Цель: Минимизировать количество цветов для правильной раскраски графа.

Алгоритм DSATUR:

1. **Инициализация:**
   - $degree[i]$ — степень вешины i
   - $neigh\_colors[i]$ — множество цветов соседей
   - $used[i]$ - отметка о раскраске
2. **Выбор вершины:** Максимальная насыщенность (размер $neigh\_colors$) или степень. Если несколько вершин, выбираем с наибольшей степенью.
3. **Раскраска:** Минимальный отсутствующий цвет в $neigh\_colors$
4. **Обновление:** Добавление цвета в $neigh\_colors$ соседей

Четвёртое задание: Покрытие бинарной матрицы

Цель: Минимизировать количество прямоугольников для покрытия однородных областей.

Жадный алгоритм:

1. **Обход матрицы:** Сверху вниз, слева направо с пропуском покрытых ячеек
2. **Построение прямоугольника:**
   - Определение базового значения (0 или 1)
   - Расширение вправо до границы значения
   - Расширение вниз с проверкой столбцов
3. **Пометка покрытия:** Регистрация координат прямоугольника и пометка этих ячеек как покрытых

Вывод по общей части

В ходе выполнения общей части практики были успешно решены четыре алгоритмические задачи различной сложности. Основные достижения и наблюдения:

1. **Эффективность алгоритмов**:
   - Для задач обработки изображений (1) и матричных операций (4) оптимальны пошаговые методы с линейной сложностью
   - Комбинированные подходы (детерминированные + вероятностные) эффективны для факторизации больших чисел (2)
   - Эвристики (DSATUR) дают хорошие результаты для NP-трудных задач (3)
2. **Технические сложности**:
   - Обработка зашумленных изображений
   - Реализация длинной арифметики для чисел $> 10^{100}$
   - Оптимизация памяти при генерации делителей
3. **Практическая значимость**:
   - Освоение методов компьютерного зрения
   - Разработка оптимизированных структур данных
   - Реализация алгоритмов для больших чисел

Решения демонстрируют высокую эффективность: все задачи решены в рамках ограничений с минимальной вычислительной сложностью, что подтверждает корректность выбранных подходов.

Индивидуальное задание

В рамках индивидуального задания была разработана простая головоломка под названием **Ternary Tiles**, вдохновлённая механикой игры «2048». Однако в отличие от классического аналога, в данной реализации используются степени тройки, а числа могут быть как положительными, так и отрицательными.

Цель игры — достичь клетки со значением определённой степени по модулю (например, 81 или $81n$), комбинируя плитки с одинаковыми модулями. Размер игрового поля зависит от выбранной сложности: 3×3, 4×4 или 5×5.

Описание игрового процесса

Каждая плитка имеет значение вида 3^n, где n — целое число (в том числе отрицательное). Плитки могут быть положительными (3^n) или отрицательными (-3^n), и они равноправны: при создании новой плитки шанс получить положительное или отрицательное значение одинаков.

Правила слияния:

- Сливаются только плитки с одинаковыми модулями.
- Если у сливаемых плиток одинаковый знак, степень увеличивается на 1.
- Если знаки противоположные, степень уменьшается на 1.
- Если результатом слияния становится степень ноль ($3^0 = 1$), плитка исчезает.

Пример:

- 3 и $3 \Rightarrow 9$
- 3 и $-3 \Rightarrow 1$, затем исчезает
- 9 и $-9 \Rightarrow 3$

Игровая логика строится так, что игроку приходится внимательно следить не только за величинами плиток, но и за их знаками. Игнорирование отрицательных плиток может привести к потере прогресса и понижению степени уже собранного значения. (Рисунки 1.1 и 1.2 иллюстрируют игровой процесс на разных уровнях сложности.)

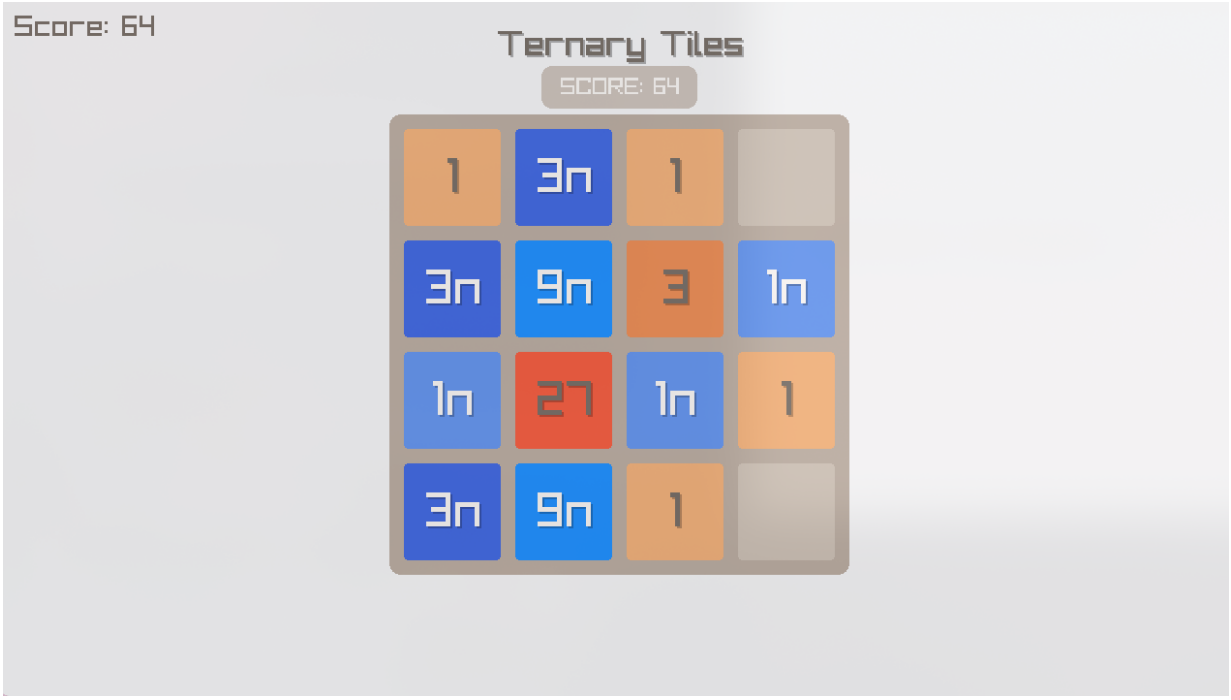

Рисунок 1.1 - Игровой процесс на базовой сложности с полем 4x4.

Рисунок 1.2 - Игровой процесс на максимальной сложности с полем 5x5.

Рисунок 1.3 - Главное меню игры.

Используемые технологии

Для реализации проекта применялись следующие инструменты:

- **C++17** — основной язык программирования. Используется стандартная библиотека STL (векторы, уникальные указатели, алгоритмы) и современный стиль разработки: RAII, умные указатели, scoped enums и другие конструкции.
- **Raylib** — кроссплатформенная C-библиотека для разработки графических приложений. Использовалась для:
  - отображения игрового поля и плиток;
  - обработки ввода с клавиатуры;
  - рендеринга текста и графических объектов;
- **OpenGL (через Raylib)** — графический API, лежащий в основе Raylib, используется для рендеринга.
- **Meson + Ninja** — система сборки. Позволяет быстро и удобно конфигурировать проект и производить инкрементальную сборку. Пример конфигурационного файла Meson представлен ниже:

```
project('TernaryTiles', 'cpp',
  default_options: ['cpp_std=c++17'])

raylib_dep = dependency('raylib', required: true)

executable('ternary_tiles',
  'src/main.cpp',
  dependencies: raylib_dep)
```

Результат

Разработанное приложение представляет собой минималистичную, но интересную головоломку, позволяющую в простой форме изучить механизмы взаимодействия данных, рендеринга и пользовательского ввода. Несмотря на малый объём, проект демонстрирует основы построения графических приложений, включая игровую логику, управление ресурсами, и сборку.

Вывод

Учебная практика позволила применить на практике ключевые знания, полученные в течение первого курса, в контексте решения прикладных и исследовательских задач. Были успешно выполнены как общие, так и индивидуальные задания, охватывающие различные аспекты программной инженерии.

В результате прохождения практики:

- Закреплены навыки анализа и реализации алгоритмов обработки данных, компьютерного зрения и комбинаторики;
- Получен практический опыт в проектировании и разработке настольного приложения на C++ с использованием библиотеки Raylib;
- Освоена система сборки Meson, обеспечивающая гибкость и масштабируемость проекта;
- Улучшены навыки структурирования и документирования кода;
- Получено понимание принципов взаимодействия пользовательского интерфейса с логикой приложения.

В рамках индивидуального проекта удалось реализовать полноценную головоломку с уникальной игровой механикой, что продемонстрировало способность к проектированию и реализации нестандартных решений. Практика прошла продуктивно и дала мощный стимул для дальнейшего профессионального роста.

Приложение А1. Исходный код индивидуального задания

```
#include "Game/Renderer.hpp"
#include <iostream>

int main() {
  constexpr int screenWidth = 1280;
  constexpr int screenHeight = 720;

  TernaryTiles::Renderer renderer(screenWidth, screenHeight, "Ternary Tiles");
  if (!renderer.isInitialized()) {
    std::cerr << "Failed to initialize renderer\n";
    return 1;
  }

  renderer.run();
  return 0;
}
```

Приложение А2. Исходный код индивидуального задания

```
#pragma once

#include "Game/GameBoard.hpp"
#include <map>
#include <raylib.h>

namespace TernaryTiles {

enum class RendererState { Menu, Playing, Help };

class Renderer {
public:
  Renderer(int width, int height, const char *title);
  ~Renderer();

  // Disable copying
  Renderer(const Renderer &) = delete;
  Renderer &operator=(const Renderer &) = delete;

  // Main rendering loop
  void run();

  // Check if the renderer was initialized successfully
  [[nodiscard]] bool isInitialized() const { return m_initialized; }

private:
  static constexpr int TILE_SIZE = 100;
  static constexpr int PADDING = 15;

  GameBoard m_board;

  Font m_font{};
  int m_fontSize{48};
  bool m_initialized{false};
```

```
// Colors
const Color BACKGROUND_COLOR{238, 228, 218, 255}; // Light beige
const Color BOARD_COLOR{187, 173, 160, 255};      // Gray
const Color TEXT_COLOR{119, 110, 101, 255};       // Dark gray
const Color TILE_TEXT_COLOR{249, 246, 242, 255};  // White for tile text

// Get color for a tile based on its value
[[nodiscard]] Color getTileColor(int value) const;

// Draw the game board
void drawBoard();

// Draw a single tile
void drawTile(int row, int col, int value);
void drawTile(int row, int col, int value, float animX, float animY,
              float tileScale);

// Draw text centered in a rectangle
void drawCenteredText(const char *text, int x, int y, int fontSize,
                      Color color) const;

// Draw text using the custom font if available
void drawText(const char *text, int posX, int posY, int fontSize,
              Color color) const;

// Draw text centered using the custom font if available
void drawCenteredText(const char *text, int y, int fontSize,
                      Color color) const;

// Handle input
void processInput();

// Animation state and helpers
struct TileAnim {
  enum class Type { Move, Spawn } type;
```

```
    float fromX, fromY, toX, toY;
    float progress; // 0.0 to 1.0
    float duration;
    int value;
    bool active;
    TileAnim(Type t, float fx, float fy, float tx, float ty, float dur, int val)
        : type(t), fromX(fx), fromY(fy), toX(tx), toY(ty), progress(0.0f),
          duration(dur), value(val), active(true) {}
    TileAnim()
        : type(Type::Move), fromX(0), fromY(0), toX(0), toY(0), progress(0),
          duration(0), value(0), active(false) {}
  };
  std::map<std::tuple<int, int, int>, TileAnim> m_tileAnims;
  float m_animSpeed = 8.0f;
  void updateAnimations(float dt);
  void startTileMoveAnim(int row, int col, int value, float from_x,
                         float from_y, float to_x, float to_y);
  void startTileSpawnAnim(int row, int col, int value, float x, float y);
  void drawAnimatingTiles();

  RendererState m_state = RendererState::Menu;
  int m_selectedBoardSize = 4; // Default 4x4
  bool m_hasWon = false;
  int m_pendingMoveDir = -1;
  bool m_waitingForAnim = false;

  void drawMenu();
  void drawHelp();
  void handleMenuInput();
  void handleHelpInput();
  void startGameWithBoardSize(int size);

  float getScale() const;
};

} // namespace TernaryTiles
```

Приложение А3. Исходный код индивидуального задания

```
#include "Game/Renderer.hpp"
#include <algorithm>
#include <cmath>
#include <functional>
#include <map>
#include <sstream>
#include <tuple>
#include <unordered_set>
#include <vector>

namespace TernaryTiles {

// Renderer handles window, drawing, and input
Renderer::Renderer(int width, int height, const char *title)
    : m_font(GetFontDefault()), m_fontSize(48), m_initialized(false),
      m_hasWon(false) {
  if (width < 320)
    width = 320;
  if (height < 240)
    height = 240;
  InitWindow(width, height, title);
  SetWindowMinSize(320, 240);
  SetWindowState(FLAG_WINDOW_RESIZABLE);
  SetTargetFPS(60);
  m_animSpeed = 14.0f;
  m_initialized = IsWindowReady();
}

float Renderer::getScale() const {
  // Compute scale so board fits in window
  const int size = m_board.grid().size();
  int boardPixels = TILE_SIZE * size + PADDING * (size + 1);
  float scaleX = (GetScreenWidth() - 2 * PADDING) / (float)boardPixels;
  float scaleY = (GetScreenHeight() - 2 * PADDING - 120) / (float)boardPixels;
```

```
  return std::min(1.0f, std::min(scaleX, scaleY));
}

Renderer::~Renderer() { CloseWindow(); }

// Main game loop
void Renderer::run() {
  float lastTime = GetTime();
  while (!WindowShouldClose()) {
    float now = GetTime();
    float dt = now - lastTime;
    lastTime = now;
    if (m_state == RendererState::Menu) {
      handleMenuInput();
      BeginDrawing();
      ClearBackground(RAYWHITE);
      drawMenu();
      EndDrawing();
      continue;
    } else if (m_state == RendererState::Help) {
      handleHelpInput();
      BeginDrawing();
      ClearBackground(RAYWHITE);
      drawHelp();
      EndDrawing();
      continue;
    }
    processInput();
    updateAnimations(dt);
    // Track win state
    if (m_board.isWin() && !m_hasWon) {
      m_hasWon = true;
    }
    BeginDrawing();
    ClearBackground(RAYWHITE);
    drawBoard();
```

```
    drawAnimatingTiles();
    // Draw score
    std::stringstream scoreText;
    scoreText << "Score: " << m_board.score();
    DrawText(scoreText.str().c_str(), PADDING, PADDING, 30, TEXT_COLOR);
    // Draw game over message if the game is over
    if (m_board.isGameOver()) {
      DrawRectangle(0, 0, GetScreenWidth(), GetScreenHeight(),
                    Color{0, 0, 0, 180});
      drawCenteredText("Game Over!", 0, -30, 40, WHITE);
      drawCenteredText("Press R to restart", 0, 20, 30, WHITE);
    }
    // Draw win banner if the player has won (but do not block input)
    if (m_hasWon) {
      int bannerHeight = 60;
      DrawRectangle(0, 0, GetScreenWidth(), bannerHeight,
                    Color{0, 128, 0, 220});
      drawCenteredText("You Win! Keep playing for a high score!", 0, 10, 32,
                       WHITE);
    }
    EndDrawing();
  }
}

// Draws the main menu
void Renderer::drawMenu() {
  const int w = GetScreenWidth();
  const int h = GetScreenHeight();
  drawCenteredText("BALANCED TERNARY 2048", h / 6, 48, TEXT_COLOR);
  drawCenteredText("Select Board Size:", h / 3, 32, TEXT_COLOR);
  int sizes[] = {3, 4, 5};
  int btnW = 100, btnH = 60, gap = 40;
  int totalW = 3 * btnW + 2 * gap;
  int startX = (w - totalW) / 2;
  int y = h / 3 + 50;
  for (int i = 0; i < 3; ++i) {
```

```
    int x = startX + i * (btnW + gap);
    Color c = (m_selectedBoardSize == sizes[i]) ? ORANGE : LIGHTGRAY;
    DrawRectangle(x, y, btnW, btnH, c);
    drawCenteredText(
        (std::to_string(sizes[i]) + "x" + std::to_string(sizes[i])).c_str(),
        x + btnW / 2, y + btnH / 2 - 16, 28, DARKGRAY);
  }
  // Start button
  int startBtnY = y + btnH + 40;
  DrawRectangle(w / 2 - 80, startBtnY, 160, 50, GREEN);
  drawCenteredText("Start Game", w / 2, startBtnY + 12, 28, WHITE);
  // Help button
  int helpBtnY = startBtnY + 70;
  DrawRectangle(w / 2 - 80, helpBtnY, 160, 40, BLUE);
  drawCenteredText("Help / Rules", w / 2, helpBtnY + 8, 24, WHITE);
}

// Draws the help/rules screen
void Renderer::drawHelp() {
  const int w = GetScreenWidth();
  const int h = GetScreenHeight();
  DrawRectangle(60, 60, w - 120, h - 120, ColorAlpha(LIGHTGRAY, 0.95f));
  drawCenteredText("How to Play", w / 2, 90, 36, TEXT_COLOR);
  int y = 140;
  int fontSize = 22;
  std::vector<std::string> lines = {
      "Combine tiles of the same absolute value:",
      "  - Same sign: merge to next power of 3 (e.g. 1+1=3, -1+-1=-3)",
      "  - Opposite sign: step down (e.g. 3+-3=1, 1+-1=0)",
      "  - Only +1 and -1 tiles spawn.",
      "Use arrow keys to move. Press R to restart.",
      "Game ends when no moves are possible."};
  for (const auto &line : lines) {
    drawCenteredText(line.c_str(), y, fontSize, TEXT_COLOR);
    y += fontSize + 8;
  }
```

```
  int btnW = 120, btnH = 40;
  int btnX = w / 2 - btnW / 2, btnY = h - 100;
  DrawRectangle(btnX, btnY, btnW, btnH, DARKGRAY);
  drawCenteredText("Back", btnY + 8, 24, WHITE);
}

// Handles menu input (mouse)
void Renderer::handleMenuInput() {
  int w = GetScreenWidth();
  int h = GetScreenHeight();
  int sizes[] = {3, 4, 5};
  int btnW = 100, btnH = 60, gap = 40;
  int totalW = 3 * btnW + 2 * gap;
  int startX = (w - totalW) / 2;
  int y = h / 3 + 50;
  Vector2 mouse = GetMousePosition();
  if (IsMouseButtonPressed(MOUSE_LEFT_BUTTON)) {
    for (int i = 0; i < 3; ++i) {
      int x = startX + i * (btnW + gap);
      Rectangle rect = {(float)x, (float)y, (float)btnW, (float)btnH};
      if (CheckCollisionPointRec(mouse, rect)) {
        m_selectedBoardSize = sizes[i];
      }
    }
    // Start button
    int startBtnY = y + btnH + 40;
    Rectangle startRect = {(float)(w / 2 - 80), (float)startBtnY, 160, 50};
    if (CheckCollisionPointRec(mouse, startRect)) {
      startGameWithBoardSize(m_selectedBoardSize);
    }
    // Help button
    int helpBtnY = startBtnY + 70;
    Rectangle helpRect = {(float)(w / 2 - 80), (float)helpBtnY, 160, 40};
    if (CheckCollisionPointRec(mouse, helpRect)) {
      m_state = RendererState::Help;
    }
```

```
  }
}

// Handles help screen input (mouse)
void Renderer::handleHelpInput() {
  int w = GetScreenWidth();
  int h = GetScreenHeight();
  Vector2 mouse = GetMousePosition();
  int btnW = 120, btnH = 40;
  int btnX = w / 2 - btnW / 2, btnY = h - 100;
  Rectangle backRect = {(float)btnX, (float)btnY, (float)btnW, (float)btnH};
  if (IsMouseButtonPressed(MOUSE_LEFT_BUTTON) &&
      CheckCollisionPointRec(mouse, backRect)) {
    m_state = RendererState::Menu;
  }
}

// Start a new game with selected board size
void Renderer::startGameWithBoardSize(int size) {
  int winValue = 0;
  switch (size) {
  case 3:
    winValue = 27;
    break;
  case 4:
    winValue = 81;
    break;
  case 5:
    winValue = 243;
    break;
  default:
    winValue = 81;
    break;
  }
  m_board = GameBoard(size, winValue);
  m_hasWon = false;
```

```
  m_state = RendererState::Playing;
}

// --- Animation helpers ---
// Animation state for a tile
struct TileAnim {
  enum class Type { Move, Spawn } type;
  float fromX, fromY, toX, toY;
  float progress; // 0.0 to 1.0
  float duration;
  int value;
  bool active;
  TileAnim(Type t, float fx, float fy, float tx, float ty, float dur, int val)
      : type(t), fromX(fx), fromY(fy), toX(tx), toY(ty), progress(0.0f),
        duration(dur), value(val), active(true) {}
};

// Update all active animations
void Renderer::updateAnimations(float dt) {
  for (auto it = m_tileAnims.begin(); it != m_tileAnims.end();) {
    TileAnim &anim = it->second;
    if (!anim.active) {
      ++it;
      continue;
    }
    anim.progress += dt / anim.duration;
    if (anim.progress >= 1.0f) {
      anim.progress = 1.0f;
      anim.active = false;
      it = m_tileAnims.erase(it);
    } else {
      ++it;
    }
  }
}
```

```
// Draw all animating tiles
void Renderer::drawAnimatingTiles() {
  for (const auto &[key, anim] : m_tileAnims) {
    if (!anim.active)
      continue;
    float x = anim.fromX + (anim.toX - anim.fromX) * anim.progress;
    float y = anim.fromY + (anim.toY - anim.fromY) * anim.progress;
    float scale = 1.0f;
    if (anim.type == TileAnim::Type::Spawn) {
      scale = 0.2f + 0.8f * anim.progress;
    }
    drawTile(std::get<0>(key), std::get<1>(key), anim.value, x, y, scale);
  }
}

// Start a move animation for a tile
void Renderer::startTileMoveAnim(int row, int col, int value, float from_x,
                                 float from_y, float to_x, float to_y) {
  m_tileAnims[{row, col, value}] =
      TileAnim(TileAnim::Type::Move, from_x, from_y, to_x, to_y, 0.15f, value);
}

// Start a spawn animation for a tile
void Renderer::startTileSpawnAnim(int row, int col, int value, float x,
                                  float y) {
  m_tileAnims[{row, col, value}] =
      TileAnim(TileAnim::Type::Spawn, x, y, x, y, 0.18f, value);
}

// Get color for a tile value
Color Renderer::getTileColor(int value) const {
  if (value == 0)
    return Color{205, 193, 180, 255};
  int absValue = std::abs(value);
  if (value > 0) {
    switch (absValue) {
```

```
    case 1:
      return Color{242, 177, 121, 255};
    case 3:
      return Color{236, 141, 83, 255};
    case 9:
      return Color{247, 124, 95, 255};
    case 27:
      return Color{245, 93, 62, 255};
    case 81:
      return Color{233, 89, 80, 255};
    default:
      return Color{200, 0, 0, 255};
    }
  } else {
    switch (absValue) {
    case 1:
      return Color{100, 149, 237, 255};
    case 3:
      return Color{65, 105, 225, 255};
    case 9:
      return Color{30, 144, 255, 255};
    case 27:
      return Color{0, 0, 205, 255};
    case 81:
      return Color{0, 0, 139, 255};
    default:
      return Color{0, 0, 100, 255};
    }
  }
}

// Draw the board, static tiles, and overlays
void Renderer::drawBoard() {
  const int size = m_board.grid().size();
  float scale = getScale();
  int boardPixels = TILE_SIZE * size + PADDING * (size + 1);
```

```
int scaledBoard = static_cast<int>(boardPixels * scale);
int startX = (GetScreenWidth() - scaledBoard) / 2;
int startY = 120;
// Center title horizontally
const char *title = "Ternary Tiles";
int titleFontSize = 36;
int titleY = 30;
int titleWidth = MeasureText(title, titleFontSize);
int titleX = (GetScreenWidth() - titleWidth) / 2;
drawText(title, titleX + 2, titleY + 2, titleFontSize,
         ColorAlpha(BLACK, 0.3f));
drawText(title, titleX, titleY, titleFontSize, TEXT_COLOR);
// Draw score with a nice background
std::string scoreText = "SCORE: " + std::to_string(m_board.score());
int scoreFontSize = 24;
int scoreY = 80;
int scorePadding = 20;
int scoreTextWidth = MeasureText(scoreText.c_str(), scoreFontSize);
int scoreBgX = (GetScreenWidth() - (scoreTextWidth + scorePadding * 2)) / 2;
int scoreBgY = scoreY - scorePadding / 2;
DrawRectangleRounded({static_cast<float>(scoreBgX),
                      static_cast<float>(scoreBgY),
                      static_cast<float>(scoreTextWidth + scorePadding * 2),
                      static_cast<float>(scoreFontSize + scorePadding)},
                     0.5f, 10, ColorAlpha(BOARD_COLOR, 0.7f));
drawCenteredText(scoreText.c_str(), scoreY, scoreFontSize, TILE_TEXT_COLOR);

// Draw board background
DrawRectangleRounded({static_cast<float>(startX), static_cast<float>(startY),
                      static_cast<float>(scaledBoard),
                      static_cast<float>(scaledBoard)},
                     0.05f, 10, BOARD_COLOR);
// Draw empty cells first (background)
for (int row = 0; row < size; ++row) {
  for (int col = 0; col < size; ++col) {
    int x = startX + static_cast<int>(PADDING * scale) +
```

```
            col * static_cast<int>((TILE_SIZE + PADDING) * scale);
    int y = startY + static_cast<int>(PADDING * scale) +
            row * static_cast<int>((TILE_SIZE + PADDING) * scale);
    DrawRectangleRounded({static_cast<float>(x), static_cast<float>(y),
                          static_cast<float>(TILE_SIZE * scale),
                          static_cast<float>(TILE_SIZE * scale)},
                         0.1f, 10, getTileColor(0));
  }
}
// Draw static tiles (skip animating ones)
struct TupleHash {
  std::size_t operator()(const std::tuple<int, int, int> &t) const {
    std::size_t h1 = std::hash<int>{}(std::get<0>(t));
    std::size_t h2 = std::hash<int>{}(std::get<1>(t));
    std::size_t h3 = std::hash<int>{}(std::get<2>(t));
    return h1 ^ (h2 << 1) ^ (h3 << 2);
  }
};
std::unordered_set<std::tuple<int, int, int>, TupleHash> animatingKeys;
for (const auto &pair : m_tileAnims) {
  if (pair.second.active)
    animatingKeys.insert(pair.first);
}
for (int row = 0; row < size; ++row) {
  for (int col = 0; col < size; ++col) {
    int value = m_board.grid()[row][col].value();
    if (value == 0)
      continue;
    auto key = std::make_tuple(row, col, value);
    if (animatingKeys.find(key) == animatingKeys.end()) {
      drawTile(row, col, value);
    }
  }
}
// Draw game over message if the game is over
if (m_board.isGameOver()) {
```

```
    DrawRectangle(0, 0, GetScreenWidth(), GetScreenHeight(),
                  Color{0, 0, 0, 128});
    const char *gameOverText = "GAME OVER";
    int gameOverFontSize = 48;
    int gameOverY = GetScreenHeight() / 2 - 60;
    drawCenteredText(gameOverText, gameOverY, gameOverFontSize, WHITE);
    const char *restartText = "Press R to restart";
    int restartFontSize = 24;
    int restartY = gameOverY + 80;
    drawCenteredText(restartText, restartY, restartFontSize, LIGHTGRAY);
  }
}

// Draw a tile at a custom position/scale (for animation)
void Renderer::drawTile([[maybe_unused]] int row, [[maybe_unused]] int col,
                        int value, float animX, float animY, float tileScale) {
  float scale = getScale();
  float tileSize = TILE_SIZE * scale * tileScale;
  float offset = (TILE_SIZE * scale - tileSize) / 2;
  DrawRectangleRounded({animX + offset, animY + offset, tileSize, tileSize},
                       0.1f, 10, getTileColor(value));
  if (value == 0)
    return;
  std::string displayText = TernaryTiles::Tile(value).toString();
  int fontSize = static_cast<int>(TILE_SIZE / 2 * scale * tileScale);
  int tileCenterX = static_cast<int>(animX + offset + tileSize / 2);
  int tileCenterY = static_cast<int>(animY + offset + tileSize / 2);
  int textWidth = MeasureText(displayText.c_str(), fontSize);
  int textHeight = fontSize;
  int textX = tileCenterX - textWidth / 2;
  int textY = tileCenterY - textHeight / 2;
  DrawText(displayText.c_str(), textX + 2, textY + 2, fontSize,
           ColorAlpha(BLACK, 0.25f));
  DrawText(displayText.c_str(), textX, textY, fontSize,
           (value < 0) ? TILE_TEXT_COLOR : TEXT_COLOR);
}
```

```
// Draw a static tile at its board position
void Renderer::drawTile(int row, int col, int value) {
  const int size = m_board.grid().size();
  float scale = getScale();
  int boardPixels = TILE_SIZE * size + PADDING * (size + 1);
  int scaledBoard = static_cast<int>(boardPixels * scale);
  int startX = (GetScreenWidth() - scaledBoard) / 2;
  int startY = 120;
  float animX = startX + PADDING * scale + col * (TILE_SIZE + PADDING) * scale;
  float animY = startY + PADDING * scale + row * (TILE_SIZE + PADDING) * scale;
  drawTile(row, col, value, animX, animY, 1.0f);
}

// Draw text at a position
void Renderer::drawText(const char *text, int posX, int posY, int fontSize,
                        Color color) const {
  DrawText(text, posX, posY, fontSize, color);
}

// Draw text centered horizontally
void Renderer::drawCenteredText(const char *text, int y, int fontSize,
                                Color color) const {
  int textWidth = MeasureText(text, fontSize);
  int x = (GetScreenWidth() - textWidth) / 2;
  drawText(text, x, y, fontSize, color);
}

// Draw text centered at (x, y)
void Renderer::drawCenteredText(const char *text, int x, int y, int fontSize,
                                Color color) const {
  int textWidth = MeasureText(text, fontSize);
  drawText(text, x - textWidth / 2, y, fontSize, color);
}

// Handle keyboard/game input
```

```
void Renderer::processInput() {
  if (m_board.isGameOver()) {
    if (IsKeyPressed(KEY_R)) {
      m_board.initialize();
    }
    return;
  }
  int moveDir = -1;
  if (IsKeyPressed(KEY_UP))
    moveDir = 0;
  else if (IsKeyPressed(KEY_RIGHT))
    moveDir = 1;
  else if (IsKeyPressed(KEY_DOWN))
    moveDir = 2;
  else if (IsKeyPressed(KEY_LEFT))
    moveDir = 3;
  if (moveDir != -1) {
    // Get move result for animation
    auto result = m_board.moveWithResult(moveDir);
    if (result.moved) {
      // Animate moved tiles
      float scale = getScale();
      int size = m_board.grid().size();
      int boardPixels = TILE_SIZE * size + PADDING * (size + 1);
      int scaledBoard = static_cast<int>(boardPixels * scale);
      int startX = (GetScreenWidth() - scaledBoard) / 2;
      int startY = 120;
      for (const auto &move : result.moves) {
        float fromX = startX + PADDING * scale +
                      move.fromCol * (TILE_SIZE + PADDING) * scale;
        float fromY = startY + PADDING * scale +
                      move.fromRow * (TILE_SIZE + PADDING) * scale;
        float toX = startX + PADDING * scale +
                    move.toCol * (TILE_SIZE + PADDING) * scale;
        float toY = startY + PADDING * scale +
                    move.toRow * (TILE_SIZE + PADDING) * scale;
```

```
        startTileMoveAnim(move.toRow, move.toCol, move.value, fromX, fromY, toX,
                          toY);
      }
      for (const auto &spawn : result.spawns) {
        float x = startX + PADDING * scale +
                  spawn.col * (TILE_SIZE + PADDING) * scale;
        float y = startY + PADDING * scale +
                  spawn.row * (TILE_SIZE + PADDING) * scale;
        startTileSpawnAnim(spawn.row, spawn.col, spawn.value, x, y);
      }
    }
  } else if (IsKeyPressed(KEY_R)) {
    m_board.initialize();
  }
}

} // namespace TernaryTiles
```

Приложение А4. Исходный код индивидуального задания

```
#pragma once

#include "Game/Tile.hpp"
#include <array>
#include <random>
#include <utility> // for std::pair
#include <vector>

namespace TernaryTiles {

class GameBoard {
public:
  static constexpr std::size_t DEFAULT_SIZE = 4;
  using Grid = std::vector<std::vector<Tile>>;
  using Row = std::vector<Tile>;

  GameBoard();
  explicit GameBoard(std::size_t size, int winValue = 81);

  // Initialize the board with two random tiles
  void initialize();

  // Move tiles in a direction (0: up, 1: right, 2: doum, 3: left)
  // Returns true if the board state changed
  bool move(int direction);

  // Check if the game is over (no more moves possible)
  [[nodiscard]] bool isGameOver() const;

  // Get the current score (sum of absolute values of all tiles)
  [[nodiscard]] int score() const {
    int total = 0;
    for (const auto &row : m_grid) {
      for (const auto &tile : row) {
```

```
      if (!tile.isEmpty()) {
        total += tile.absValue();
      }
    }
  }
  return total;
}

// Get a read-only view of the grid
[[nodiscard]] const Grid &grid() const { return m_grid; }

// For testing and debugging
void setTile(int row, int col, int value) {
  if (row >= 0 && row < static_cast<int>(DEFAULT_SIZE) && col >= 0 &&
      col < static_cast<int>(DEFAULT_SIZE)) {
    m_grid[row][col] = Tile(value);
  }
}

void setWinValue(int winValue) { m_winValue = winValue; }
void setSize(std::size_t size);
[[nodiscard]] std::size_t size() const { return m_size; }
[[nodiscard]] bool isWin() const;

struct MoveInfo {
  int fromRow, fromCol;
  int toRow, toCol;
  int value;
};
struct SpawnInfo {
  int row, col;
  int value;
};
struct MoveResult {
  bool moved = false;
  std::vector<MoveInfo> moves;
```

```
    std::vector<SpawnInfo> spawns;
  };

  // New: Move with result info for animation
  MoveResult moveWithResult(int direction);

private:
  std::size_t m_size;
  int m_winValue;
  Grid m_grid;
  std::mt19937 m_rng{std::random_device{}()};
  std::uniform_int_distribution<std::size_t> m_pos_dist{0, DEFAULT_SIZE - 1};
  std::uniform_int_distribution<int> m_value_dist{0, 1}; // 0 or 1 -> -1 or 1

  // Helper methods
  bool moveLeft();
  bool moveRight();
  bool moveUp();
  bool moveDown();
  bool moveRowLeft(Row &row);
  bool moveRowRight(Row &row);
  void addRandomTile();
  bool canMove() const;
  [[nodiscard]] bool isFull() const;

  // Get a list of empty cell positions
  std::vector<std::pair<size_t, size_t>> getEmptyCells() const;

  // Helper for moveWithResult
  void addRandomTileWithResult(MoveResult &result);
};

} // namespace TernaryTiles
```

Приложение А5. Исходный код индивидуального задания

```
#include "Game/GameBoard.hpp"
#include <algorithm>
#include <iostream>
#include <random>

namespace TernaryTiles {

GameBoard::GameBoard() { initialize(); }

GameBoard::GameBoard(std::size_t size, int winValue)
    : m_size(size), m_winValue(winValue),
      m_grid(size, std::vector<Tile>(size)) {
  initialize();
}

void GameBoard::setSize(std::size_t size) {
  m_size = size;
  m_grid.assign(size, std::vector<Tile>(size));
  initialize();
}

void GameBoard::initialize() {
  for (auto &row : m_grid) {
    for (auto &tile : row) {
      tile = Tile{};
    }
  }
  addRandomTile();
  addRandomTile();
}

std::vector<std::pair<size_t, size_t>> GameBoard::getEmptyCells() const {
  std::vector<std::pair<size_t, size_t>> emptyCells;
  for (size_t i = 0; i < m_size; ++i) {
```

```
    for (size_t j = 0; j < m_size; ++j) {
      if (m_grid[i][j].isEmpty()) {
        emptyCells.emplace_back(i, j);
      }
    }
  }
  return emptyCells;
}

bool GameBoard::move(int direction) {
  bool moved = false;
  switch (direction) {
  case 0:
    moved = moveUp();
    break;
  case 1:
    moved = moveRight();
    break;
  case 2:
    moved = moveDown();
    break;
  case 3:
    moved = moveLeft();
    break;
  }
  if (moved) {
    addRandomTile();
  }
  return moved;
}

// Move with animation result for UI
GameBoard::MoveResult GameBoard::moveWithResult(int direction) {
  MoveResult result;
  auto oldGrid = m_grid;
  bool moved = false;
```

```
switch (direction) {
case 0:
  moved = moveUp();
  break;
case 1:
  moved = moveRight();
  break;
case 2:
  moved = moveDown();
  break;
case 3:
  moved = moveLeft();
  break;
}
result.moved = moved;
if (moved) {
  // Track moved and merged tiles for animation
  std::vector<std::vector<bool>> matchedOld(m_size,
                                            std::vector<bool>(m_size, false));
  for (size_t toRow = 0; toRow < m_size; ++toRow) {
    for (size_t toCol = 0; toCol < m_size; ++toCol) {
      const Tile &newTile = m_grid[toRow][toCol];
      if (!newTile.isEmpty() && newTile != oldGrid[toRow][toCol]) {
        bool found = false;
        for (size_t fromRow = 0; fromRow < m_size && !found; ++fromRow) {
          for (size_t fromCol = 0; fromCol < m_size && !found; ++fromCol) {
            if (!matchedOld[fromRow][fromCol] &&
                !oldGrid[fromRow][fromCol].isEmpty() &&
                oldGrid[fromRow][fromCol].value() == newTile.value()) {
              result.moves.push_back({(int)fromRow, (int)fromCol, (int)toRow,
                                      (int)toCol, newTile.value()});
              matchedOld[fromRow][fromCol] = true;
              found = true;
            }
          }
        }
```

```
          if (!found) {
            result.moves.push_back({(int)toRow, (int)toCol, (int)toRow,
                                    (int)toCol, newTile.value()});
          }
        }
      }
    }
    addRandomTileWithResult(result);
  }
  return result;
}

bool GameBoard::moveLeft() {
  bool moved = false;
  for (auto &row : m_grid) {
    if (moveRowLeft(row)) {
      moved = true;
    }
  }
  return moved;
}

bool GameBoard::moveRight() {
  bool moved = false;
  for (auto &row : m_grid) {
    if (moveRowRight(row)) {
      moved = true;
    }
  }
  return moved;
}

bool GameBoard::moveUp() {
  bool moved = false;
  for (size_t col = 0; col < m_size; ++col) {
    Row column(m_size);
```

```
    for (size_t row = 0; row < m_size; ++row) {
      column[row] = m_grid[row][col];
    }
    if (moveRowLeft(column)) {
      moved = true;
      for (size_t row = 0; row < m_size; ++row) {
        m_grid[row][col] = column[row];
      }
    }
  }
  return moved;
}

bool GameBoard::moveDown() {
  bool moved = false;
  for (size_t col = 0; col < m_size; ++col) {
    Row column(m_size);
    for (size_t row = 0; row < m_size; ++row) {
      column[row] = m_grid[m_size - 1 - row][col];
    }
    if (moveRowLeft(column)) {
      moved = true;
      for (size_t row = 0; row < m_size; ++row) {
        m_grid[m_size - 1 - row][col] = column[row];
      }
    }
  }
  return moved;
}

// Slide and merge left
bool GameBoard::moveRowLeft(Row &row) {
  size_t write_pos = 0;
  bool moved = false;
  for (size_t i = 0; i < m_size; ++i) {
    if (!row[i].isEmpty()) {
```

```
      if (i != write_pos) {
        row[write_pos] = row[i];
        row[i].clear();
        moved = true;
      }
      ++write_pos;
    }
  }
  for (size_t i = 0; i < m_size - 1;) {
    if (!row[i].isEmpty() && row[i].canMergeWith(row[i + 1])) {
      Tile merged = row[i].mergeWith(row[i + 1]);
      row[i] = merged;
      for (size_t j = i + 1; j < m_size - 1; ++j) {
        row[j] = row[j + 1];
      }
      row[m_size - 1].clear();
      moved = true;
      continue;
    }
    ++i;
  }
  return moved;
}

// Slide and merge right
bool GameBoard::moveRowRight(Row &row) {
  int write_pos = m_size - 1;
  bool moved = false;
  for (int i = m_size - 1; i >= 0; --i) {
    if (!row[i].isEmpty()) {
      if (i != write_pos) {
        row[write_pos] = row[i];
        row[i].clear();
        moved = true;
      }
      --write_pos;
```

```
    }
  }
  for (int i = m_size - 1; i > 0;) {
    if (!row[i].isEmpty() && row[i].canMergeWith(row[i - 1])) {
      Tile merged = row[i].mergeWith(row[i - 1]);
      row[i] = merged;
      for (int j = i - 1; j > 0; --j) {
        row[j] = row[j - 1];
      }
      row[0].clear();
      moved = true;
      continue;
    }
    --i;
  }
  return moved;
}

// Add a random +1 or -1 tile to an empty cell
void GameBoard::addRandomTile() {
  auto emptyCells = getEmptyCells();
  if (emptyCells.empty())
    return;
  std::uniform_int_distribution<size_t> cellDist(0, emptyCells.size() - 1);
  const auto &[row, col] = emptyCells[cellDist(m_rng)];
  int value = (m_value_dist(m_rng) == 0) ? -1 : 1;
  m_grid[row][col] = Tile{value};
}

// Add a random tile and record it for animation
void GameBoard::addRandomTileWithResult(MoveResult &result) {
  auto emptyCells = getEmptyCells();
  if (emptyCells.empty())
    return;
  std::uniform_int_distribution<size_t> cellDist(0, emptyCells.size() - 1);
  const auto &[row, col] = emptyCells[cellDist(m_rng)];
```

```
  int value = (m_value_dist(m_rng) == 0) ? -1 : 1;
  m_grid[row][col] = Tile{value};
  result.spawns.push_back({(int)row, (int)col, value});
}

// Check if any moves are possible
bool GameBoard::canMove() const {
  for (const auto &row : m_grid) {
    for (const auto &tile : row) {
      if (tile.isEmpty()) {
        return true;
      }
    }
  }
  for (size_t i = 0; i < m_size; ++i) {
    for (size_t j = 0; j < m_size - 1; ++j) {
      if (m_grid[i][j].canMergeWith(m_grid[i][j + 1])) {
        return true;
      }
    }
  }
  for (size_t i = 0; i < m_size - 1; ++i) {
    for (size_t j = 0; j < m_size; ++j) {
      if (m_grid[i][j].canMergeWith(m_grid[i + 1][j])) {
        return true;
      }
    }
  }
  return false;
}

bool GameBoard::isFull() const {
  for (const auto &row : m_grid) {
    for (const auto &tile : row) {
      if (tile.isEmpty()) {
        return false;
```

```
      }
    }
  }
  return true;
}

// No more moves possible
bool GameBoard::isGameOver() const { return !canMove(); }

// Win if any tile reaches or exceeds win value
bool GameBoard::isWin() const {
  for (const auto &row : m_grid) {
    for (const auto &tile : row) {
      if (tile.absValue() >= m_winValue) {
        return true;
      }
    }
  }
  return false;
}

} // namespace TernaryTiles
```

Приложение А6. Исходный код индивидуального задания

```
#pragma once

#include <cstdint>
#include <string>
#include <vector>

namespace TernaryTiles {

using TernaryDigit = int8_t;


std::vector<TernaryDigit> decimalToBalancedTernary(int n);

/**
 * @brief Converts a balanced ternary number to decimal
 * @param digits Vector of ternary digits (least significant digit first)
 * @return The decimal equivalent
 */
int balancedTernaryToDecimal(const std::vector<TernaryDigit> &digits);

/**
 * @brief Adds two balanced ternary digits with carry
 * @param a First digit
 * @param b Second digit
 * @param carry Input/Output carry (can be -1, 0, or 1)
 * @return The sum digit
 */
TernaryDigit addTernaryDigits(TernaryDigit a, TernaryDigit b,
                              TernaryDigit &carry);

/**
 * @brief Adds two balanced ternary numbers
 * @param a First number (least significant digit first)
 * @param b Second number (least significant digit first)
```

```
 * @return The sum in balanced ternary (least significant digit first)
 */
std::vector<TernaryDigit>
addBalancedTernary(const std::vector<TernaryDigit> &a,
                   const std::vector<TernaryDigit> &b);

/**
 * @brief Converts a balanced ternary number to string representation
 * @param digits The number to convert (least significant digit first)
 * @return String representation (most significant digit first)
 */
std::string balancedTernaryToString(const std::vector<TernaryDigit> &digits);

/**
 * @brief Converts a balanced ternary number to its decimal string
 * representation
 * @param digits The number to convert (least significant digit first)
 * @return Decimal string representation
 */
std::string
balancedTernaryToDecimalString(const std::vector<TernaryDigit> &digits);

} // namespace TernaryTiles
```

Приложение А7. Исходный код индивидуального задания

```
#include "Game/BalancedTernary.hpp"
#include <algorithm>
#include <cassert>
#include <cmath>
#include <sstream>

namespace TernaryTiles {

std::vector<TernaryDigit> decimalToBalancedTernary(int n) {
  std::vector<TernaryDigit> result;

  if (n == 0) {
    result.push_back(0);
    return result;
  }

  while (n != 0) {
    int remainder = n % 3;
    n = n / 3;

    if (remainder == 2) {
      remainder = -1;
      n++;
    } else if (remainder == -2) {
      remainder = 1;
      n--;
    } else if (remainder == -1) {
      remainder = -1;
    }

    result.push_back(static_cast<TernaryDigit>(remainder));
  }

  return result;
```

```
}

int balancedTernaryToDecimal(const std::vector<TernaryDigit> &digits) {
  int result = 0;
  int power = 1;

  for (TernaryDigit d : digits) {
    // Use assert instead of exception since exceptions are disabled
    assert(d >= -1 && d <= 1 && "Invalid balanced ternary digit");
    result += d * power;
    power *= 3;
  }

  return result;
}

TernaryDigit addTernaryDigits(TernaryDigit a, TernaryDigit b,
                              TernaryDigit &carry) {
  // Normalize inputs to -1, 0, or 1
  a = (a > 0) - (a < 0);
  b = (b > 0) - (b < 0);

  int sum = a + b + carry;

  // Determine the result digit and new carry
  TernaryDigit result;
  if (sum > 1) {
    result = sum - 3;
    carry = 1;
  } else if (sum < -1) {
    result = sum + 3;
    carry = -1;
  } else {
    result = sum;
    carry = 0;
  }
```

```
  return result;
}

std::vector<TernaryDigit>
addBalancedTernary(const std::vector<TernaryDigit> &a,
                   const std::vector<TernaryDigit> &b) {
  std::vector<TernaryDigit> result;
  TernaryDigit carry = 0;
  size_t max_len = std::max(a.size(), b.size());

  for (size_t i = 0; i < max_len || carry != 0; ++i) {
    TernaryDigit digit_a = (i < a.size()) ? a[i] : 0;
    TernaryDigit digit_b = (i < b.size()) ? b[i] : 0;

    TernaryDigit sum = addTernaryDigits(digit_a, digit_b, carry);
    result.push_back(sum);
  }

  // Remove leading zeros
  while (result.size() > 1 && result.back() == 0) {
    result.pop_back();
  }

  return result;
}

std::string balancedTernaryToString(const std::vector<TernaryDigit> &digits) {
  if (digits.empty()) {
    return "0";
  }

  std::string result;
  // Iterate from most significant to least significant digit
  for (auto it = digits.rbegin(); it != digits.rend(); ++it) {
    if (*it == -1) {
```

```
      result += 'n';
    } else {
      result += ('0' + *it);
    }
  }

  return result;
}

std::string
balancedTernaryToDecimalString(const std::vector<TernaryDigit> &digits) {
  int decimal = balancedTernaryToDecimal(digits);
  return std::to_string(decimal);
}

} // namespace TernaryTiles
```

Приложение А8. Исходный код индивидуального задания

```
#pragma once

#include <cmath>
#include <cstdint>
#include <string>
#include <vector>

namespace TernaryTiles {

class Tile {
public:
  // Default constructor creates a tile with value 0 (empty)
  constexpr Tile() = default;

  // Create a tile with a specific value (must be a power of 3 or its negative,
  // or 0) If an invalid value is provided, creates an empty tile (0)
  constexpr explicit Tile(int value) : m_value{0} {
    if (value != 0) {
      int abs_val = std::abs(value);
      if (isPowerOfThree(abs_val)) {
        m_value = value;
      }
      // If not a power of 3, leave as 0 (empty)
    }
  }

  // Get the tile's value
  [[nodiscard]] constexpr int value() const { return m_value; }

  // Check if the tile is empty (value == 0)
  [[nodiscard]] constexpr bool isEmpty() const { return m_value == 0; }

  // Check if this tile can merge with another tile
  // Tiles can only merge if they have the same absolute value
```

```
[[nodiscard]] constexpr bool canMergeWith(const Tile &other) const {
  if (isEmpty() || other.isEmpty())
    return false;

  // Only allow merges between tiles with the same absolute value
  return std::abs(m_value) == std::abs(other.m_value);
}

// Merge this tile with another tile (assumes canMergeWith is true)
// Returns the new tile after merging according to the rules
[[nodiscard]] Tile mergeWith(const Tile &other) const {
  if (!canMergeWith(other)) {
    return *this; // Shouldn't happen if used correctly
  }

  int abs_this = std::abs(m_value);
  int abs_other = std::abs(other.m_value);

  // Handle same value merges (1+1=3, -1+-1=-3, 3+3=9, etc.)
  if (m_value == other.m_value) {
    return Tile(m_value * 3);
  }

  // Handle opposite value merges (1+-1=0, 3+-3=1, 9+-9=3, etc.)
  if (m_value == -other.m_value) {
    if (abs_this == 1) {
      return Tile(0); // 1 + -1 = 0
    } else {
      return Tile(m_value > 0
                      ? (m_value / 3)
                      : (m_value / 3)); // 3 + -3 = 1, -3 + 3 = -1, etc.
    }
  }

  // Handle step-down merges (3 + 1 = 1, -3 + -1 = -1, etc.)
  if (abs_this * 3 == abs_other) {
```

```
      return Tile(m_value > 0 ? (abs_this) : (-abs_this));
    } else if (abs_this == abs_other * 3) {
      return Tile(other.m_value > 0 ? (abs_other) : (-abs_other));
    }

    // Should never reach here if canMergeWith is correct
    return *this;
  }

  // Reset the tile to empty
  constexpr void clear() { m_value = 0; }

  // Get the absolute value of the tile
  [[nodiscard]] constexpr int absValue() const { return std::abs(m_value); }

  // Check if two tiles are equal
  [[nodiscard]] constexpr bool operator==(const Tile &other) const {
    return m_value == other.m_value;
  }

  // Check if two tiles are not equal
  [[nodiscard]] constexpr bool operator!=(const Tile &other) const {
    return !(*this == other);
  }

  // Get a string representation of the tile
  [[nodiscard]] std::string toString() const {
    if (isEmpty())
      return "0";
    if (m_value < 0)
      return std::to_string(-m_value) + "n";
    return std::to_string(m_value);
  }

private:
  int m_value{0}; // Can be 0, ±1, ±3, ±9, ±27, etc.
```

```
  // Helper function to check if a number is a power of 3
  static constexpr bool isPowerOfThree(int n) {
    if (n <= 0)
      return false;
    while (n % 3 == 0) {
      n /= 3;
    }
    return n == 1;
  }
};

} // namespace TernaryTiles
```

Приложение А9. Исходный код индивидуального задания

```
project(
  'ternary_tiles',
  'cpp',
  version: '0.1',
  meson_version: '>=1.3.0',
  default_options: [
    'warning_level=3',
    'cpp_std=c++17',
    'default_library=static',
    'buildtype=release',
    'cpp_eh=none',
    'cpp_rtti=false',
  ]
)

# Include directories
inc = include_directories('include')

# Source files
srcs = files(
  'src/main.cpp',
  'src/Game/GameBoard.cpp',
  'src/Game/Renderer.cpp',
  'src/Game/BalancedTernary.cpp',
)

# Dependencies
raylib_dep = dependency('raylib')

deps = [raylib_dep]

# Executable
ternary_tiles_exe = executable(
  'ternary_tiles',
```

```
  srcs,
  include_directories: inc,
  dependencies: deps,
  install: true,
  cpp_args: ['-DRAYGUI_IMPLEMENTATION'],
)

# Basic test (runs the executable)
test('run-game', ternary_tiles_exe)
```