DashMorph: A Stack-Based 2D Transformation Game

Nimo Benne
University of the Fraser Valley, Abbotsford, BC
nimo.benne@student.ufv.ca

ABSTRACT

DashMorph is a dynamic 2D side-scroller platformer game that challenges players to traverse levels by jumping and morphing into different geometric shapes to overcome obstacles. The game utilizes stack-based push-pop transformations for object hierarchy and movement. This report details the design, implementation, and user interaction mechanisms of DashMorph, including custom UI elements, physics-based interactions, and visual effects. The game was developed using Processing and follows the principles of hierarchical modeling and transformation matrices.

AUTHOR KEYWORDS

2D Transformation, Stack-Based Graphics, Game Development, Processing, Hierarchical Modeling, UI Interaction.

INTRODUCTION

DashMorph is a game similar to Geometry Dash. The player navigates through the level by morphing into different shapes to adapt to the obstacles. The game is created in Processing using Java and fulfills the 8 main goals for this project. This report will discuss the implementation of the game and how we got to the final product that is DashMorph.

PROJECT REQUIREMENTS IMPLEMENTATION

1. USE OF JAVA GENERICS

- The game uses an ArrayList<> to dynamically manage obstacles
- A static array is used for storing predefined shape morphs: String[] shapes = {"Square", "Rectangle", "Triangle"};

2. CLASS HIERARCHY AND PARENT-CHILD RELATIONSHIP

- The game implements an abstract class (Entity) as a grandparent class
- The Player, Obstacle, and FinishLine inherit from Entity
- The Player can morph into three different shapes, and the environment consists of dynamic obstacles
- The game has multiple different shaped objects used throughout the whole game

3. PUSH AND POP STACK STRUCTURE

- Using pushMatrix() and popMatrix() for the player transformations

4. MULTIPLE CUSTOM FUNCTIONS FOR OBJECT MOVEMENT

The game implements multiple functions for unique movement:

- jump() for vertical movement
- collidesWith(Obstacle o) for collision detection
- Player.mode to switch between shapes but not technically a “movement”
- update() handles the players gravity which is considered a movement because it updates the position dynamically

5. MOUSE AND KEYBOARD INTERACTIONS

KEYBOARD

- SPACE to jump
- 1, 2, 3 to morph into different shapes
- P to pause the game
- R to restart after game over or win

MOUSE

- UI buttons (Start, Exit) respond to mouse hover
- mousePressed() detects clicks on Start and Exit

6. CUSTOM IMAGE FILTER

- A grayscale filter is applied when the player loses.
- A blur filter is applied when the player wins.

7. UI COMPONENTS

UI elements implemented:

1. Custom Rollover Effect: Background color changes when hovering over Start & Exit buttons.

2. Custom UI Buttons (Start & Exit) with changing colors.

3. Instruction Text: Displays control instructions dynamically.

4. Game Over & Win Messages: Displays dynamic game states.

5. Pause Menu: Pauses the game logic dynamically.

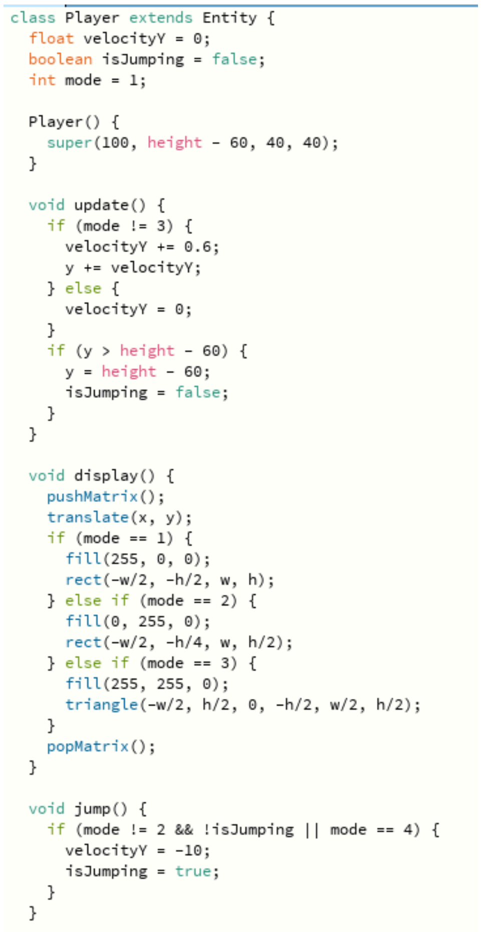
6. Shape Morph Feedback: Console gets an output to what shape the player changed to.

7. Hover Activated Color Change: Hovering buttons changes the background of the menu.

METHODOLOGY

The game design incorporates multiple core elements, including:

- **Stack-Based 2D Transformations:**

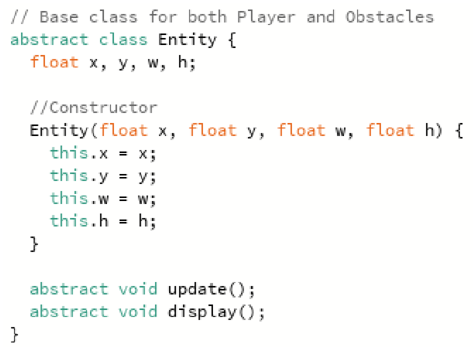


```
class Player extends Entity {
  float velocityY = 0;
  boolean isJumping = false;
  int mode = 1;

  Player() {
    super(100, height - 60, 40, 40);
  }

  void update() {
    if (mode != 3) {
      velocityY += 0.6;
      y += velocityY;
    } else {
      velocityY = 0;
    }
    if (y > height - 60) {
      y = height - 60;
      isJumping = false;
    }
  }

  void display() {
    pushMatrix();
    translate(x, y);
    if (mode == 1) {
      fill(255, 0, 0);
      rect(-w/2, -h/2, w, h);
    } else if (mode == 2) {
      fill(0, 255, 0);
      rect(-w/2, -h/4, w, h/2);
    } else if (mode == 3) {
      fill(255, 255, 0);
      triangle(-w/2, h/2, 0, -h/2, w/2, h/2);
    }
    popMatrix();
  }

  void jump() {
    if (mode != 2 && !isJumping || mode == 4) {
      velocityY = -10;
      isJumping = true;
    }
  }
```

Figure 1. Player Code Snippet

This code snippet uses push and pop matrices and "morphing" the shapes with the use of modes which allow for seamless transformation of the shapes during gameplay. There are also constraints depending on what shape the player is currently morphed as.

- **Class-Based Object Structure:**

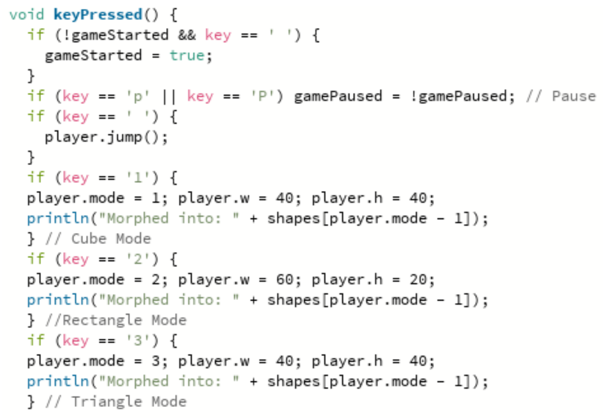


```
// Base class for both Player and Obstacles
abstract class Entity {
  float x, y, w, h;

  //Constructor
  Entity(float x, float y, float w, float h) {
    this.x = x;
    this.y = y;
    this.w = w;
    this.h = h;
  }

  abstract void update();
  abstract void display();
}
```

Figure 2. Code Snippet Abstract Base Class

Using Entity as the base and allowing Player and Obstacle to use it as a template for further use.

- **Mouse and Keyboard Interaction:**

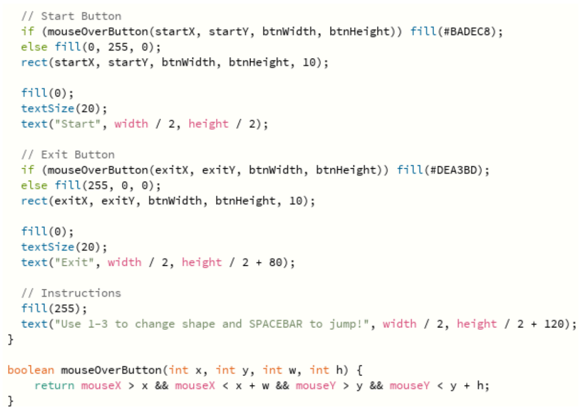


```
void keyPressed() {
  if (!gameStarted && key == ' ') {
    gameStarted = true;
  }
  if (key == 'p' || key == 'P') gamePaused = !gamePaused; // Pause
  if (key == ' ') {
    player.jump();
  }
  if (key == '1') {
  player.mode = 1; player.w = 40; player.h = 40;
  println("Morphed into: " + shapes[player.mode - 1]);
  } // Cube Mode
  if (key == '2') {
  player.mode = 2; player.w = 60; player.h = 20;
  println("Morphed into: " + shapes[player.mode - 1]);
  } //Rectangle Mode
  if (key == '3') {
  player.mode = 3; player.w = 40; player.h = 40;
  println("Morphed into: " + shapes[player.mode - 1]);
  } // Triangle Mode
```

Figure 3. keyPressed Code Snippet

Players can use keyboard inputs to jump and morph into various shapes

- **Custom UI Components:**

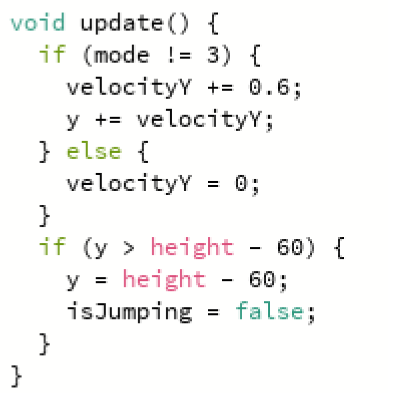


```
  // Start Button
  if (mouseOverButton(startX, startY, btnWidth, btnHeight)) fill(#BADEC8);
  else fill(0, 255, 0);
  rect(startX, startY, btnWidth, btnHeight, 10);

  fill(0);
  textSize(20);
  text("Start", width / 2, height / 2);

  // Exit Button
  if (mouseOverButton(exitX, exitY, btnWidth, btnHeight)) fill(#DEA3BD);
  else fill(255, 0, 0);
  rect(exitX, exitY, btnWidth, btnHeight, 10);

  fill(0);
  textSize(20);
  text("Exit", width / 2, height / 2 + 80);

  // Instructions
  fill(255);
  text("Use 1-3 to change shape and SPACEBAR to jump!", width / 2, height / 2 + 120);
}

boolean mouseOverButton(int x, int y, int w, int h) {
    return mouseX > x && mouseX < x + w && mouseY > y && mouseY < y + h;
}
```

Figure 4. Custom Menu UI Code Snippet

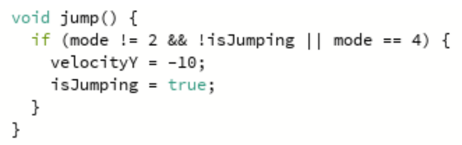
Custom UI for the game menu with added buttons and a rollover effect when hovering on the buttons in the menu.

- **Game Physics:**



```
void update() {
  if (mode != 3) {
    velocityY += 0.6;
    y += velocityY;
  } else {
    velocityY = 0;
  }
  if (y > height - 60) {
    y = height - 60;
    isJumping = false;
  }
}
```

Figure 5. Gravity Code Snippet

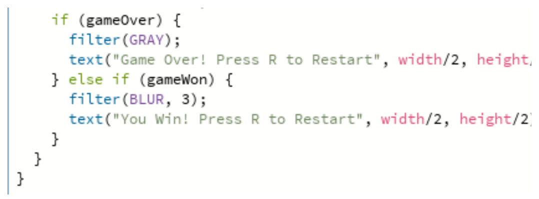


```
void jump() {
  if (mode != 2 && !isJumping || mode == 4) {
    velocityY = -10;
    isJumping = true;
  }
}
```

Figure 6. Jump Code Snippet

The game implements gravity, collision detection and shape based movement dynamics that are all used together to create a smooth and seamless gameplay loop. Figure 5 shows how we stop the player from falling below a certain height and that we are pulling the shape down with a gradual velocity.

- **Image Filters:**



```
    if (gameOver) {
      filter(GRAY);
      text("Game Over! Press R to Restart", width/2, height,
    } else if (gameWon) {
      filter(BLUR, 3);
      text("You Win! Press R to Restart", width/2, height/2
    }
  }
}
```

Figure 7. Filter Code Snippet

Figure 7 shows how we apply filters to the players screen based on if they lost the game or won the game. Either by gray scaling the screen on a loss or giving the screen a blur effect on a win.

GAME FEATURES

Character Morphing: The player's shape can change into different geometric shapes (Square, Rectangle, Triangle) that have different effects.

Obstacle Avoidance: Different barriers require specific shapes for traversal.

Finish Line Mechanism: The goal is to reach the end of each level.

Dynamic UI Components: Custom-made UI elements allow for game control and visual adjustments.

Stack-Based Animation: Push and pop transformations ensure smooth transitions between morphing states.

Game Over and Background Visual Effects: Implements grayscale filtering when the player loses and a parallax background effect.

ALGORITHM IMPLEMENTATION

The game follows a structured algorithmic approach:

1. Initialize game entities (player, obstacles, finish line).
2. Detects user input for movement and shape transformation.
3. Apply stack-based transformations.
4. Implement collision detection between shape and obstacle.
5. Update the game state and redraw the scene dynamically.
6. Apply grayscale effect on game over (collision).
7. Check if the player has reached the finish line.

USER INTERACTION

Keyboard Controls:

- Spacebar for jumping.
- '1-3' keys for morphing into different shapes.
- 'P' key for pausing the game.
- 'R' key for restarting after game over or win.

Mouse Interaction:

- UI buttons Start and Exit with hover effects.
- Background color changes dynamically when hovering over UI elements.

UML DIAGRAM

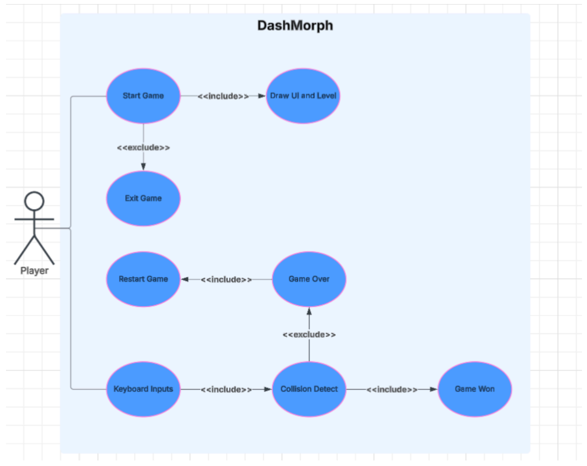


Figure 8. UML Use Case Diagram

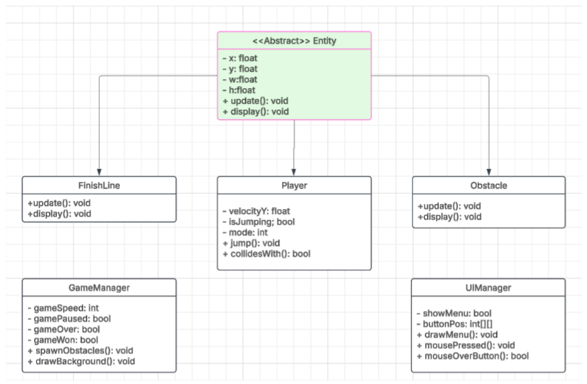


Figure 9. UML Class Diagram

DEMO IMAGES

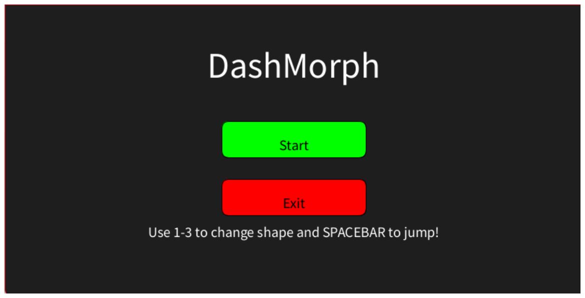


Figure 10. DashMorph Menu UI

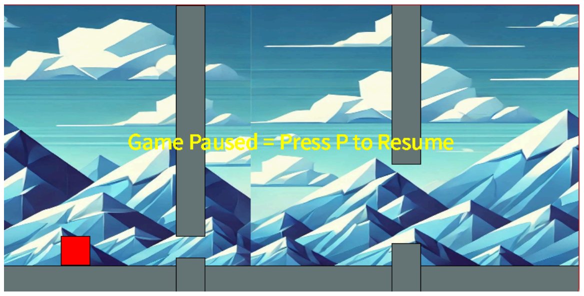


Figure 11. DashMorph Gameplay Demo

CONCLUSION

The game is complete and functional and implements the 8 core goals that are needed. Using stack-based approaches we ensure smooth transition between “morphing” and the hierarchical object modeling allows for scalability. Using custom UI, filters, background parallax and visual improvements to enhance the players experience. Future improvements on level design and difficulty scaling would help with replayability for the player. Improving the physics and creating new obstacles and shapes will help the flow of the game. Overall this is a quick fun game that implements all the basics of Stack-based Push-Pop 2D Transformation Matrices.

REFERENCES

- GeeksforGeeks. (n.d.). *Abstract classes in Java*. Retrieved from https://www.geeksforgeeks.org/abstract-classes-in-java/

- Happy Coding. (n.d.). *Processing collision detection tutorial*. Retrieved from https://happycoding.io/tutorials/processing/collision-detection

- Processing Foundation. (n.d.). *Processing reference*. Retrieved from https://processing.org

- TutsPlus Game Development. (n.d.). *Game UI by example: A crash course in the good and the bad*. Retrieved from https://gamedevelopment.tutsplus.com/tutorials/game-ui-by-example-a-crash-course-in-the-good-and-the-bad--gamedev-3943

- Version Museum. (n.d.). *History of Geometry Dash*. Retrieved from https://www.versionmuseum.com/history-of/geometry-dash

- YouTube - Coding Train. (n.d.). *Processing 2D transformation tutorial* [Video]. Retrieved from https://www.youtube.com/watch?v=9xEIrHJZMaw