# Design Overview for <<BeautifulGame>>

Name: Nguyen Dinh Dung

Student ID: 104772138

# Summary of Program

Describe what you want the program to do… one or two paragraphs.

### Overview of the Ruby Program

**Game Launcher Window**

**The player will see a screen showing 3 levels, the player can choose the level and they will be moved to that game window. After winning each game players can go back to this window and choose for other levels. The difficulties increasing by each level.**

1. **Initialization and Setup**
   * The launcher initializes a window with a size of 640x480 pixels.
   * It sets up a menu with a background image, title text, and interactive buttons for selecting different game modes.
2. **Menu Elements**
   * **Title Text**: Displays the title of the launcher or the name of the game.
   * **Buttons**: Interactive elements that allow the player to select which game to start. There are 3 games for the user to choose with the difficulties increasing by each level.
3. **Event Handling**
   * **Button Clicks**: Detects when a button is clicked and starts the corresponding game by loading the relevant script.
   * **Key Presses**: Optionally handles key presses to navigate through the menu or activate certain options.
4. **Game Selection Logic**
   * Based on user input (e.g., button clicks or key presses), the launcher closes itself and launches the selected game.
   * The launcher ensures that all required assets are loaded and the game is properly initialized before starting.
5. **Update Loop**
   * The launcher may update its display or handle user interactions continuously while waiting for the user to make a selection.

**Maze Game**

**The player will navigate through a maze to obtain rubies. When the player can obtain the ruby they will win the game.**

1. **Initialization and Setup**
   * **Window Setup**: Initializes an 800x600 pixel game window.
   * **Maze Layout**: Defines a maze layout using a 2D array where 1 represents walls and 0 represents paths.
   * **Cell Size**: Defines the size of each cell in the maze (40 pixels).
   * **Assets**: Loads and sets up images for the player sprite and collectible item (ruby), and initializes sound and music assets.
2. **Game Elements**
   * **Maze**: The maze is represented as a grid with walls and paths. The maze is drawn on the screen using rectangles.
   * **Player Sprite**: The main character controlled by the player. It can move in four directions (up, down, left, right) and has an animation for movement.
   * **Ruby**: A collectible item that increases the player’s score when collected. It is placed at random valid positions in the maze.
3. **Event Handling**
   * **Key Held**: Moves the player sprite in the direction specified by the arrow keys. Movement is constrained by maze walls. When the sprite collects the ruby, the score is updated, and the ruby is repositioned.
   * **Key Up**: Stops the sprite’s animation when no key is pressed.
   * **Key Down**: If the spacebar is pressed and the game is over, the game is reset.
4. **Collision Detection**
   * **Wall Collision**: Checks if the player sprite collides with maze walls. Movement is prevented if a collision is detected.
   * **Ruby Collision**: Checks if the player sprite collides with the ruby. If a collision occurs, the score increases, a sound effect plays, and the ruby is moved to a new position.
5. **Game Over Logic**
   * **Winning Condition**: If player obtain 2 ruby, a "You win" message is displayed, and the player is prompted to press the spacebar to exit the game.
   * **Reset Game**: Pressing the spacebar after winning will close the game.
6. **Update Loop**
   * **Maze Drawing**: The maze is drawn initially using the draw\_maze function.
   * **Player Movement**: Continuously updates the player’s position based on key inputs while checking for collisions with walls and the ruby.
   * **Score Management**: Updates the score display and checks for win conditions.
7. **Additional Features**
   * **Sound Effects**: Plays a sound effect when the player collects the ruby.
   * **Background Music**: Loops background music throughout the game.

**Collector Game**

**The player will have to control the character to run away from the monster chasing them and at the same time have to obtain 20 rubies to win. When player move to the border of the window they will be move to another map.**

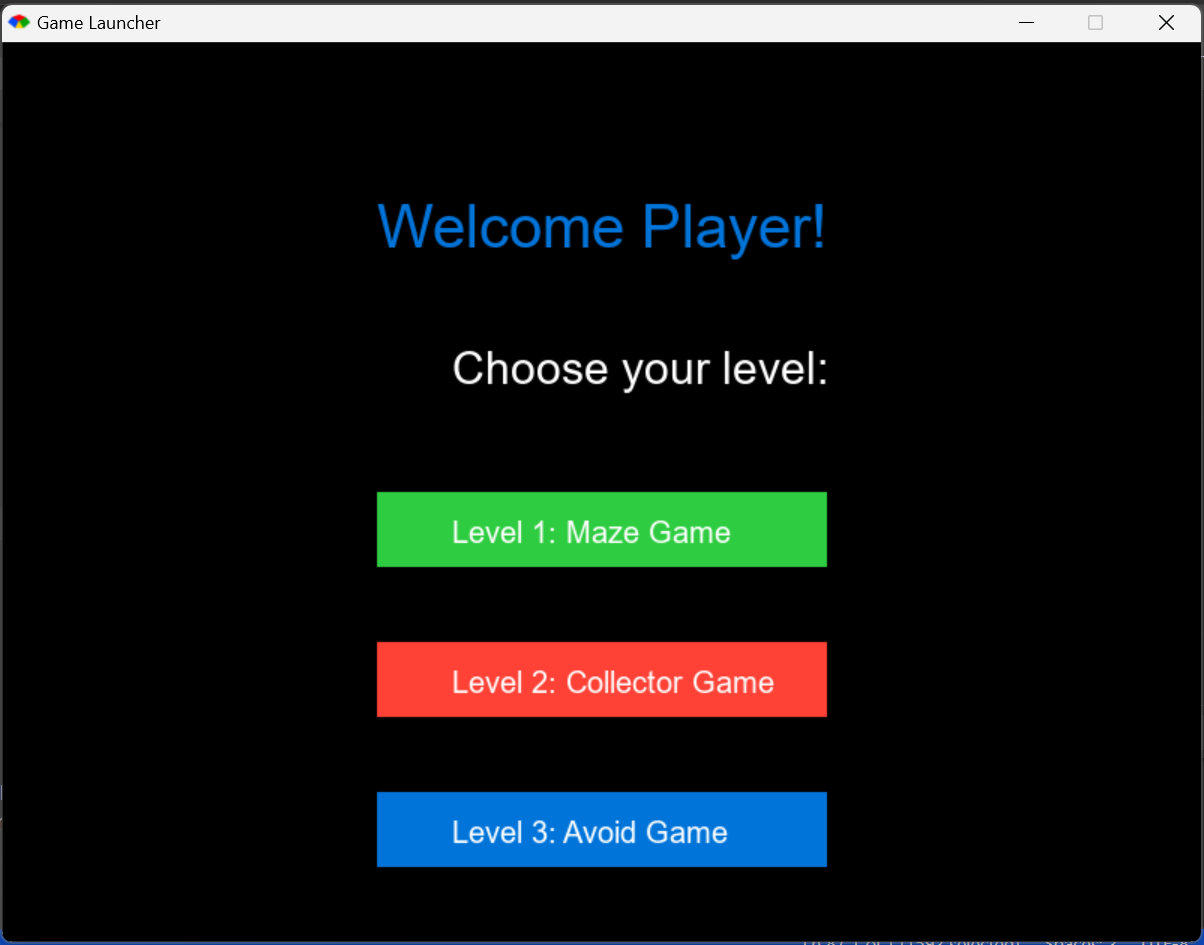
1. **Initialization and Setup**
   * **Window Setup**: Initializes a 640x480 pixel game window.
   * **Background Image**: Loads a background image to be displayed in the game.
   * **Game Elements**:
     + **Circle**: A yellow circle with a radius of 90 pixels.
     + **Triangle**: An orange triangle defined by three vertices.
     + **Sprite**: A player-controlled character sprite with animation.
     + **Ruby**: A collectible item represented by an image.
     + **Monster**: An enemy represented by an image.
   * **Sound and Music**: Loads sound effects for jumps and deaths, and plays background music in a loop.
   * **Text Elements**: Displays the game title and score on the screen.
2. **Game Elements**
   * **Circle and Triangle**: Static shapes drawn on the screen as part of the game’s visual elements.
   * **Sprite**: The main character controlled by the player, which can move in all directions and has an animation for movement.
   * **Ruby**: A collectible item that increases the player’s score when touched.
   * **Monster**: An enemy that chasing the player and decreases the score upon collision.
3. **Event Handling**
   * **Key Held**: Moves the player sprite in the direction specified by the arrow keys. If the sprite moves outside the window boundaries, the background image is switched.
   * **Key Up**: Stops the sprite’s animation when no key is pressed.
   * **Key Down**: If the spacebar is pressed, the game either resets (if the score is 0) or closes (if a win condition is met).
4. **Collision Detection**
   * **Ruby Collision**: Checks if the player sprite collides with the ruby. If a collision occurs, the score increases, the ruby moves to a new position, and a sound effect plays.
   * **Monster Collision**: Checks if the player sprite collides with the monster. If a collision occurs, the score decreases, and a death sound plays if the score drops to 0.
   * **Background Switching**: If the sprite flies outside the window borders, the background image is switched.
5. **Game Over Logic**
   * **Winning Condition**: If the player’s score reaches 20, a "You Win" message is displayed, and the player is prompted to press the spacebar to move to another level.
   * **Game Over Condition**: If the score drops to 0, a "Game Over" message is displayed, and the player is prompted to press the spacebar to restart the game.
6. **Update Loop**
   * **Monster Movement**: Continuously updates the monster’s position to move towards the player sprite.
   * **Collision Detection**: Checks for collisions between the player sprite and other game elements (ruby and monster) to update the score and handle game-over conditions.
7. **Additional Features**
   * **Background Switching**: When the sprite moves outside the window, the background image toggles between two images, and all game elements are re-rendered to appear on top of the new background so it works as a map changing.

**Avoid game**

**This game will require players to move up and down to avoid the knife moving towards the character. Avoid 30 and you will win, if you touch 1 you will lose. The hitbox of the knife is quite large so this is the hardest level.**

1. **Initialization and Setup**
   * **Window Setup**: Initializes a 600x400 pixel game window.
   * **Background Image**: Loads a background image to be displayed in the game.
   * **Game Elements**:
     + **Sprite**: The player-controlled character sprite with animation, starting at a fixed position.
     + **Obstacle**: Knives represented by an image, which appear and move across the screen.
   * **Sound and Music**: Loads sound effects for jumps and game over, and plays background music in a loop.
   * **Text Elements**: Displays the game title, score, and messages for game over or winning conditions.
2. **Game Elements**
   * **Sprite**: The main character that the player controls, with vertical movement allowed within the window’s boundaries.
   * **Obstacles**: Knives that move from right to left across the screen. Each obstacle has a hitbox to detect collisions with the player.
3. **Event Handling**
   * **Key Held**: Moves the sprite up or down in response to the arrow keys while ensuring it stays within the window boundaries.
   * **Key Up**: Stops the sprite’s animation when no key is pressed.
   * **Key Down**: If the spacebar is pressed, the game either resets (if game over) or closes (if the player has won).
4. **Collision Detection**
   * **Obstacle Collision**: Checks if the sprite’s hitbox intersects with any obstacles' hitboxes. If a collision occurs, the game over sound plays, and a "Game Over" message is displayed.
   * **Obstacle Movement**: Moves obstacles from right to left. If an obstacle moves off-screen, it is removed, and the score is increased. If the score reaches 30, a "You Win" message is displayed.
5. **Game Over Logic**
   * **Winning Condition**: If the score reaches 30, a "You Win" message is displayed, and the player is prompted to press the spacebar to close the game.
   * **Game Over Condition**: If the sprite collides with an obstacle, a "Game Over" message is displayed, and the player is prompted to press the spacebar to restart the game.
6. **Update Loop**
   * **Obstacle Management**: Continuously updates the position of obstacles, spawns new obstacles at set intervals, and checks for collisions with the sprite. If the score reaches 30, the win condition is checked.
   * **Obstacle Spawning**: New obstacles are spawned at regular intervals, and their positions are updated each frame.
7. **Additional Features**
   * **Reset Game**: Resets the game state, including the sprite's position, obstacles, score, and text messages.
   * **Obstacle Rotation**: Obstacles are displayed with a 90-degree rotation to add visual variety.

Include a sketch of sample output to illustrate your idea.



A screenshot of a video game

Description automatically generated

A screenshot of a video game

Description automatically generated

A screenshot of a video game

Description automatically generated

# Required Data Types

Describe each of the records and enumerations you will create using the following table (one per record).

Table 1: <<record name>> details

|  |  |  |
| --- | --- | --- |
| Field Name | Type | Notes |
| |  | | --- | | width |  |  | | --- | |  | | Integer | |  | | --- | | Width of the game window in pixels |  |  | | --- | |  | |
| height | Integer | Height of the game window in pixels |
| x | Integer | X-coordinate of the sprite's position |
| y | Integer | Y-coordinate of the sprite's position |
| animations |  | Animation states |
| file\_path | String | Path to all sound and image files |
| content | String | Message when player lose |
| color | String | Color of the text |
| object1 | Object | The first object involved in the collision |
| object2 | Object | The second object involved in the collision |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

Table 2: <<enumeration name>> details

|  |  |
| --- | --- |
| Value | Notes |
| up | Indicates upward movement |
| down | Indicates downward movement |
| left | Indicates leftward movement |
| right | Indicates rightward movement |
| playing | Indicates the game is currently playing |
| game\_over | Indicates the game is over |
| transition | Indicates a transition between two states. |
|  |  |
|  |  |
|  |  |
|  |  |

…

# Overview of Program Structure

List the main functions/procedures you are going to need to create this program. For each function/procedure provide its name and a brief description of what it will do.

Don’t spend too long on this at this stage. Focus on the main things you think you are likely to need and you can build on this as your program develops.

### Main Functions/Procedures

Collector Game

1. **setup\_game\_window**
   * **Description**: Initializes the game window with specified dimensions and loads all necessary assets such as background images, sounds, and music.
2. **create\_sprite**
   * **Description**: Creates and initializes the main player sprite with specified attributes like position, size, and animations.
3. **create\_image**
   * **Description**: Loads an image into the game with a specified file path and position.
4. **create\_sound**
   * **Description**: Loads a sound effect with the given file path.
5. **create\_text**
   * **Description**: Creates a text element at a specified position with given content, size, and color.
6. **move\_sprite**
   * **Description**: Moves the sprite based on user input (key presses) and updates its position accordingly.
7. **check\_collision**
   * **Description**: Checks for collisions between two game objects and returns a boolean value indicating whether a collision occurred.
8. **move\_monster**
   * **Description**: Moves the monster towards the player's sprite continuously.
9. **move\_ruby**
   * **Description**: Moves the ruby to a random position within the game window.
10. **update\_score**
    * **Description**: Updates the score display based on the player's current score.
11. **reset\_game**
    * **Description**: Resets the game state, including the player sprite's position, the ruby's position, and the monster's position, and removes the game-over text if it exists.
12. **play\_sound**
    * **Description**: Plays the specified sound effect.
13. **change\_map**
    * **Description**: Closes the current game window and opens a new game window with a different map (i.e., switches between demo.rb and demo1.rb).
14. **handle\_key\_held**
    * **Description**: Handles continuous key press events to move the sprite and checks for collisions with the ruby and monster.
15. **handle\_key\_up**
    * **Description**: Stops the sprite's animation when a key is released.
16. **handle\_key\_down**
    * **Description**: Handles single key press events, such as restarting the game when the spacebar is pressed after a game-over.
17. **game\_loop**
    * **Description**: The main game loop that continuously updates the game state, including moving the monster, checking for collisions, and handling game-over conditions.

Include a structure chart (once you have your proposal approved by your tutor)

A screenshot of a computer

Description automatically generated

Maze Game

**1.draw\_maze(maze, cell\_size)**:

* **Description**: This function is responsible for rendering the maze layout on the screen. It iterates through the maze array and draws a rectangle for each cell that is a wall (represented by 1).

**2.random\_valid\_position(maze, cell\_size)**:

* **Description**: This function finds and returns a random valid position in the maze where there is no wall (represented by 0). It ensures that the position is within the bounds of the maze and not occupied by a wall.

**3.set\_ruby\_position(cell\_size)**:

* **Description**: This function sets the position of the ruby image at a random valid position in the maze. It uses the random\_valid\_position function to determine the position.

**4.reset\_game**:

* **Description**: This function resets the game state, closing the current game window. It's used to reset the game when the player chooses to restart.

**5.wall\_collision?(x, y, maze, cell\_size)**:

* **Description**: This function checks if a given position (x, y) in the maze collides with any walls. It returns true if there is a collision and false otherwise.

**6.check\_collision(object1, object2)**:

* **Description**: This function checks if two objects collide with each other. It returns true if there is a collision and false otherwise.

**7.Event handlers**:

* **Description**: These are various event handler functions that respond to key presses and control the game logic. Specifically:
  + **on**

**do |event|**: Handles continuous key presses to move the player sprite in the maze, checks for wall collisions, and updates the score when the player collects a ruby.

* + **on**

**do**: Stops the player's sprite animation when a key is released.

* + **on**

**do |event|**: Handles the 'space' key press to reset the game if the game is over.

**8.Initialization and Setup**:

* **Description**: These are not single functions but parts of the script where the window is set up, game elements (like the player sprite and ruby image) are initialized, and the maze is drawn. This setup ensures that all elements are in place before the game starts.

Include a structure chart (once you have your proposal approved by your tutor)

A screenshot of a computer screen

Description automatically generated

Avoid Game

1. **reset\_game**:
   * **Description**: This function resets the game state, setting the player sprite back to its starting position, clearing all obstacles, resetting the score, and removing any game over or win text.
2. **spawn\_obstacle**:
   * **Description**: This function creates a new obstacle image and a smaller hitbox for collision detection. The obstacle is positioned at a random vertical location off the right edge of the screen and is added to the obstacles array.
3. **check\_collision(object1, object2)**:
   * **Description**: This function checks if two objects collide with each other. It returns true if there is a collision and false otherwise.
4. **Event Handlers**:
   * **on**

**do |event|**:

* + - **Description**: Handles continuous key presses to move the player sprite up or down, ensuring it stays within the screen bounds.
  + **on**

**do**:

* + - **Description**: Stops the player's sprite animation when a key is released.
  + **on**

**do |event|**:

* + - **Description**: Handles the 'space' key press to reset the game if it's over, or close the window if the player has won.

1. **update**:
   * **Description**: This is the main game loop that updates the game state. It moves obstacles across the screen, checks for collisions, updates the score, and handles game over or win conditions. It also spawns new obstacles at regular intervals.

**Additional Components in the Program**

1. **Initialization and Setup**:
   * **Description**: This sets up the game window, loads assets (images, sounds, music), initializes game elements (player sprite, background image, text elements), and starts playing background music.
2. **Drawing and Rendering**:
   * **Description**: While not a single function, the drawing and rendering part involves creating and positioning game elements such as the background image, player sprite, obstacles, and text elements.

Include a structure chart (once you have your proposal approved by your tutor)

A diagram of a computer program

Description automatically generated with medium confidence

Game Launcher

1. **setup\_window**
   * **Description**: Sets up the main window for the game launcher, including the title and dimensions.
   * **Parameters**: None
   * **Returns**: None
2. **create\_welcome\_text**
   * **Description**: Creates and displays the welcome text on the main screen.
   * **Parameters**: None
   * **Returns**: Text object representing the welcome text
3. **create\_level\_text**
   * **Description**: Creates and displays the text for level selection prompt.
   * **Parameters**: None
   * **Returns**: Text object representing the level selection text
4. **create\_level\_button**
   * **Description**: Creates and displays a button for a specific game level.
   * **Parameters**: x: int, y: int, width: int, height: int, color: String, text: String
   * **Returns**: Tuple containing the Rectangle object for the button and the Text object for the button's label
5. **launch\_game**
   * **Description**: Launches the specified game in a new thread.
   * **Parameters**: game\_file: String
   * **Returns**: None
6. **handle\_mouse\_click**
   * **Description**: Handles mouse click events to check if a level button was clicked and launch the corresponding game.
   * **Parameters**: event: Event
   * **Returns**: None

**Detailed Descriptions:**

**setup\_window**

* **Description**: Initializes the window settings for the game launcher.
* **Implementation**:

ruby

def setup\_window

set title: "Game Launcher"

set width: 800, height: 600

end

**create\_welcome\_text**

* **Description**: Displays a welcome message to the player.
* **Implementation**:

ruby

def create\_welcome\_text

Text.new(

'Welcome Player!',

x: 250, y: 100,

size: 40,

color: 'blue'

)

end

**create\_level\_text**

* **Description**: Displays the text prompting the player to choose a level.
* **Implementation**:

ruby

def create\_level\_text

Text.new(

'Choose your level:',

x: 300, y: 200,

size: 30,

color: 'white'

)

end

**create\_level\_button**

* **Description**: Creates a button for a specific game level.
* **Parameters**:
  + x: The x-coordinate of the button.
  + y: The y-coordinate of the button.
  + width: The width of the button.
  + height: The height of the button.
  + color: The color of the button.
  + text: The label text of the button.
* **Implementation**:

ruby

def create\_level\_button(x, y, width, height, color, text)

button = Rectangle.new(

x: x, y: y,

width: width, height: height,

color: color

)

button\_text = Text.new(

text,

x: x + 50, y: y + 15,

size: 20,

color: 'white'

)

[button, button\_text]

end

**launch\_game**

* **Description**: Launches the specified game file in a new thread.
* **Parameters**:
  + game\_file: The filename of the game to be launched.
* **Implementation**:

ruby

def launch\_game(game\_file)

Thread.new do

# Use exec to run the game, but ensure it runs in its own thread

system("ruby #{game\_file}")

end

end

**handle\_mouse\_click**

* **Description**: Handles mouse click events to check if a level button was clicked and launch the corresponding game.
* **Parameters**:
  + event: The mouse click event object.
* **Implementation**:

ruby

def handle\_mouse\_click(event)

if level1\_button.contains?(event.x, event.y)

launch\_game('maze\_game.rb')

elsif level2\_button.contains?(event.x, event.y)

launch\_game('collector\_game.rb')

elsif level3\_button.contains?(event.x, event.y)

launch\_game('avoid\_game.rb')

end

end

Include a structure chart (once you have your proposal approved by your tutor)

A screenshot of a computer

Description automatically generated