VolcanoIsles Programmer's Documentation

Contents

Project structure

The game consists of two parts – the general game engine and the game itself. The files of the game engine are located in the 'GameEngine' folder and all classes which belong to the engine are subsumed under the namespace g.

The Game Engine

General Information

The game engine is responsible for displaying the graphics, distributing events and managing the game state. The SDL library which provides a simplified access to OpenGL is used to display the visual elements.

Setting up a Game

The GameManager class runs the main loop of the game in which the events are handled, the game is updated and the graphics is drawn on the screen. The game is started by instantiating the GameManager and calling the method start(). A game state needs to be passed to the start method which describes the actions that should happen in the game loop.

The GameState class can serve as a base class for a custom game state class. It's most important methods are add_action(), which registers an object that shall be called on each game update and add_graphics() method, which adds an objects to the lists of objects that are drawn on the game canvas. The section bellow describes the objects that can be displayed.

Displaying Graphics

The Graphics class is a base class which specifies the interface shared by all displayable objects. Each displayable object either represents an object which is directly drawn on the screen or is a container for a group of such objects. The engine contains two container classes: GraphicsContainer and StaticGraphicsContainer. Both these containers have add_child() method which appends a new element to the list of drawn objects. The two containers differ only in a way in which they take the child. The former takes a raw pointer to the element whereas the latter requires a unique pointer.

The most important class which represents a directly displayed element is a class Image. A name of the image that should be displayed is passed in the constructor as well as the coordinates of the top left corner of the image on screen. The image to be displayed needs to be saved in the Assets folder inside the game directory and it must have a suffix .png. The Image class contains also additional methods to manipulate with the image properties, such as rotation, visibility and transparency (alpha).

The engine contains also a similar but more advanced class called Sprite, which is capable of displaying only a part of the source image. This can be useful for drawing images from sprite sheets or when drawing animations. The area that is displayed is specified by the return value of a function get_source_rect() which needs to be overridden together with the get_destination_rect() method that returns a target rectangle into which the graphics shall be drawn.

There are also classes for displaying simpler shapes such as a Rectangle class and a Line class.

Handling Events

The game state keeps a separate list of objects that are notified when some event occurs for each event type. Use methods such as add_key_handler() to register objects which shall receive the information. Separate category of events is mouse events. Those are associated not only with the callback but also with the object, on which the mouse event occurred. There are two types of mouse events: mouse click and mouse motion. The mouse motion event is called whenever the mouse enters or leaves the observed object. The methods add_mouse_click_handler() and add_mouse_motion_handler() take 2 parameters. First of them is a reference to an object derived from class CollisionComputable on which the event is tracked and the second parameter is the handler called when the event occurs. The Graphics objects such as Image and Sprite are derived from CollisionComputable.

Button

The Button class provides a more advanced and often used control element with predefined on mouse enter / on mouse leave behavior. It is used in the same manner as the Image class. The GameState needs to be passed to the button in the constructor and the mouse enter and mouse leave callbacks are registered automatically. However, the Button still needs to be put in the display list manually and mouse click response needs to be set by the user as well. There is a method add_click_handler() for convenience directly on the button, though. The button can be enabled / disabled by using the set_enabled method.

Other Features

The engine also contains a few often used functions such as templated functions for erasing of an object from the vector (remove and vector_remove for removing by index) which are located in the common namespace together with the function for conversion from radians to degrees rad_to_deg.

The DelayTimer class is useful for deferred actions. The start method starts takes the number of seconds to wait for and the time_elapsed() method keeps returning false until the given time period passes.

The VolcanoIsles Game

Overall Structure

The game model is held by the GamePlan class. The transitions between turns and simulation of the environment are managed by the VolcanoGameDirector class. The flow of the game is conducted from the VolcanoGameState class which is derived from the GameState class. The GamePlanGraphics class owns reference to GamePlan class and paints the scene on the screen. Islands are represented by the GameNode class. The players are controlled by classes that inherit from the class AbstractPlayerControl class.

Game Plan

The GamePlan class contains a list of islands (GameNode class) and a list of fireballs (FireBall class) that are travelling over the islands. The list of islands, the links between them and their positions on screen are loaded from file in the load() method. The files are text files with the suffix .map and with the following structure:

Each line in the file stores information about one island: its x position on screen, its y position, and a list of indices of its neighbors. An index of an island corresponds to a line number of a line on which it is defined. For an example, see maps folder in the game directory. The island on the first line (with index 1) becomes the base of the blue player; the island on the last line becomes the base of the red player.

The other methods in the GamePlan class are useful for convenient work with the game plan. It contains a method get_neighbours_of() which returns a list of neighbors of a given node, random_neighbour() and similar methods.

Game Node

The GameNode class saves the information about the island: the owner of the island, the type of the island (empty, base, volcano) its location on screen, a list of indices of neighbor nodes and an index of a node towards which the fireballs will travel from this island. These characteristics can be accessed and changed using the getter and setter methods.

Graphical Representation

For each class which models the game plan there is a class with the same name but the Graphics suffix, such as: GamePlanGraphics, GameNodeGraphics, FireBallGraphics. An instance of GamePlanGraphics class is initialized after the game plan is loaded in the VolcanoGameState::init() method. The GameNodeGraphics is a container which contains and displays all graphics that is related to island, that is the island graphics and the arrow which shows the path direction. The island graphics is drawn by the class IslandGraphics which is a Sprite. It loads a spritesheet with all possible types and colors of an island and displays them according to the current state of the underlying island. Graphics of an arrow is a simple Image. The list of islands remains unchanged during the whole game, each object of type GameNodeGraphics keeps a pointer to the island it represents. However, the list of FireBalls changes and therefore it is necessary to update the graphical representation each time the game scene changes. This is performed by the VolcanoGameState each time the turn ends. It notifies the GamePlanGraphics to update the list of drawn FireBallGraphics objects.

Simulation of the Environment

The VolcanoGameDirector contains methods such as change_direction(), occupy_island(), produce_fire(), which apply the given action as if it was performed by the player who is currently on turn. Because player can perform only one action per turn, once such a method is called, the director automatically ends the turn and simulates the game progress. This is done by calling the simulate_nature() method, which moves the fireballs according to the rules of the game: All fireballs move in the direction of the arrows, destroying the island and vanishing if two or more fireballs meet on the same island or on the same link in the opposite direction.

Controlling the Players

The abstract class AbstractPlayerControl provides an interface for controlling the player. It comprises methods turn_started(), turn_ended(), proceed(), which are called by the VolcanoGameState. It checks the state of VolcanoGameDirector on each game update and if the player on turn changed, it notifies the controllers. The turn_started() method is called when the turn of the player controlled by the AbstractPlayerControl starts, turn_ended(), method when the turn ends and proceed() method is called regularly in between. The game implementation contains two alternative controllers: PlayerMouseControl class and PlayerAIControl class. The first one is used in case the player is controlled by the user of the computer, the second one if the player is controlled by AI.

Mouse Control

The PlayerMouseControl is responsible for controlling the game by mouse. It adds mouse over and mouse click listeners on the islands. If the user selects a valid action, the game director is called and the action is performed, which ends the turn and the control is deactivated. A clicking on the rotate button stops the island mouse click listeners and activates RotationControl which enables the rotation action and reports the result of the rotation back to PlayerMouseControl.

Control by Artificial Intelligence

The AI controller uses a minimax algorithm to decide on the next action. The algorithm, which can be found in a separate class MiniMax, is called asynchronously, so that the computation does not block the user interface. To prevent the opposite problem that the

computation might take too short time, the controller waits for the specified amount of time before applying the action.

Minimax Algorithm

The state space of the game is huge; therefore it was necessary to introduce some optimizations to be able to search through at least 3 - 4 turns in advance (6 – 8 turns of both players). The algorithm searches the states up to the certain deepness, which is given as a constant. Then the game plan is evaluated and a score is assigned.

Classic Alpha-Beta pruning was implemented, to make it efficient; the actions are sorted in a descending order according to the probability that they will have a high score.

Because of the many possible node rotations, the branching factor is high. To reduce the branching factor, the algorithm expands only rotations of those islands, which are on a possible path of some fireball.

Game Constants

The header file GameConstants.h contains a namespace gameconst with the constants that are used throughout the program. Edit this file to change the size of the game window, size of the graphical elements, the minimal length of the turn of the AI or the maximum depth for the Minimax algorithm.