**Sokoban**

In this part of the Java 2D games tutorial, we will create a Java Sokoban game clone.

**Sokoban**

*Sokoban* is another classic computer game. It was created in 1980 by Hiroyuki Imabayashi. Sokoban means a warehouse keeper in Japanese. The player pushes boxes around a maze. The objective is to place all boxes in designated locations.

**Development**

We control the sokoban object with cursor keys. We can also press the R key to restart the level. When all bags are placed on the destination areas, the game is finished. We draw "Completed" string in the left upper corner of the window.

Board.java

package sokoban;

import java.awt.Color;

import java.awt.Graphics;

import java.awt.event.KeyAdapter;

import java.awt.event.KeyEvent;

import java.util.ArrayList;

import javax.swing.JPanel;

public class Board extends JPanel {

private final int OFFSET = 30;

private final int SPACE = 20;

private final int LEFT\_COLLISION = 1;

private final int RIGHT\_COLLISION = 2;

private final int TOP\_COLLISION = 3;

private final int BOTTOM\_COLLISION = 4;

private ArrayList walls = new ArrayList();

private ArrayList baggs = new ArrayList();

private ArrayList areas = new ArrayList();

private Player soko;

private int w = 0;

private int h = 0;

private boolean completed = false;

private String level =

" ######\n"

+ " ## #\n"

+ " ##$ #\n"

+ " #### $##\n"

+ " ## $ $ #\n"

+ "#### # ## # ######\n"

+ "## # ## ##### ..#\n"

+ "## $ $ ..#\n"

+ "###### ### #@## ..#\n"

+ " ## #########\n"

+ " ########\n";

public Board() {

addKeyListener(new TAdapter());

setFocusable(true);

initWorld();

}

public int getBoardWidth() {

return this.w;

}

public int getBoardHeight() {

return this.h;

}

public final void initWorld() {

int x = OFFSET;

int y = OFFSET;

Wall wall;

Baggage b;

Area a;

for (int i = 0; i < level.length(); i++) {

char item = level.charAt(i);

if (item == '\n') {

y += SPACE;

if (this.w < x) {

this.w = x;

}

x = OFFSET;

} else if (item == '#') {

wall = new Wall(x, y);

walls.add(wall);

x += SPACE;

} else if (item == '$') {

b = new Baggage(x, y);

baggs.add(b);

x += SPACE;

} else if (item == '.') {

a = new Area(x, y);

areas.add(a);

x += SPACE;

} else if (item == '@') {

soko = new Player(x, y);

x += SPACE;

} else if (item == ' ') {

x += SPACE;

}

h = y;

}

}

public void buildWorld(Graphics g) {

g.setColor(new Color(250, 240, 170));

g.fillRect(0, 0, this.getWidth(), this.getHeight());

ArrayList world = new ArrayList();

world.addAll(walls);

world.addAll(areas);

world.addAll(baggs);

world.add(soko);

for (int i = 0; i < world.size(); i++) {

Actor item = (Actor) world.get(i);

if ((item instanceof Player)

|| (item instanceof Baggage)) {

g.drawImage(item.getImage(), item.x() + 2, item.y() + 2, this);

} else {

g.drawImage(item.getImage(), item.x(), item.y(), this);

}

if (completed) {

g.setColor(new Color(0, 0, 0));

g.drawString("Completed", 25, 20);

}

}

}

@Override

public void paint(Graphics g) {

super.paint(g);

buildWorld(g);

}

class TAdapter extends KeyAdapter {

@Override

public void keyPressed(KeyEvent e) {

if (completed) {

return;

}

int key = e.getKeyCode();

if (key == KeyEvent.VK\_LEFT) {

if (checkWallCollision(soko,

LEFT\_COLLISION)) {

return;

}

if (checkBagCollision(LEFT\_COLLISION)) {

return;

}

soko.move(-SPACE, 0);

} else if (key == KeyEvent.VK\_RIGHT) {

if (checkWallCollision(soko,

RIGHT\_COLLISION)) {

return;

}

if (checkBagCollision(RIGHT\_COLLISION)) {

return;

}

soko.move(SPACE, 0);

} else if (key == KeyEvent.VK\_UP) {

if (checkWallCollision(soko,

TOP\_COLLISION)) {

return;

}

if (checkBagCollision(TOP\_COLLISION)) {

return;

}

soko.move(0, -SPACE);

} else if (key == KeyEvent.VK\_DOWN) {

if (checkWallCollision(soko,

BOTTOM\_COLLISION)) {

return;

}

if (checkBagCollision(BOTTOM\_COLLISION)) {

return;

}

soko.move(0, SPACE);

} else if (key == KeyEvent.VK\_R) {

restartLevel();

}

repaint();

}

}

private boolean checkWallCollision(Actor actor, int type) {

if (type == LEFT\_COLLISION) {

for (int i = 0; i < walls.size(); i++) {

Wall wall = (Wall) walls.get(i);

if (actor.isLeftCollision(wall)) {

return true;

}

}

return false;

} else if (type == RIGHT\_COLLISION) {

for (int i = 0; i < walls.size(); i++) {

Wall wall = (Wall) walls.get(i);

if (actor.isRightCollision(wall)) {

return true;

}

}

return false;

} else if (type == TOP\_COLLISION) {

for (int i = 0; i < walls.size(); i++) {

Wall wall = (Wall) walls.get(i);

if (actor.isTopCollision(wall)) {

return true;

}

}

return false;

} else if (type == BOTTOM\_COLLISION) {

for (int i = 0; i < walls.size(); i++) {

Wall wall = (Wall) walls.get(i);

if (actor.isBottomCollision(wall)) {

return true;

}

}

return false;

}

return false;

}

private boolean checkBagCollision(int type) {

if (type == LEFT\_COLLISION) {

for (int i = 0; i < baggs.size(); i++) {

Baggage bag = (Baggage) baggs.get(i);

if (soko.isLeftCollision(bag)) {

for (int j=0; j < baggs.size(); j++) {

Baggage item = (Baggage) baggs.get(j);

if (!bag.equals(item)) {

if (bag.isLeftCollision(item)) {

return true;

}

}

if (checkWallCollision(bag,

LEFT\_COLLISION)) {

return true;

}

}

bag.move(-SPACE, 0);

isCompleted();

}

}

return false;

} else if (type == RIGHT\_COLLISION) {

for (int i = 0; i < baggs.size(); i++) {

Baggage bag = (Baggage) baggs.get(i);

if (soko.isRightCollision(bag)) {

for (int j=0; j < baggs.size(); j++) {

Baggage item = (Baggage) baggs.get(j);

if (!bag.equals(item)) {

if (bag.isRightCollision(item)) {

return true;

}

}

if (checkWallCollision(bag,

RIGHT\_COLLISION)) {

return true;

}

}

bag.move(SPACE, 0);

isCompleted();

}

}

return false;

} else if (type == TOP\_COLLISION) {

for (int i = 0; i < baggs.size(); i++) {

Baggage bag = (Baggage) baggs.get(i);

if (soko.isTopCollision(bag)) {

for (int j = 0; j < baggs.size(); j++) {

Baggage item = (Baggage) baggs.get(j);

if (!bag.equals(item)) {

if (bag.isTopCollision(item)) {

return true;

}

}

if (checkWallCollision(bag,

TOP\_COLLISION)) {

return true;

}

}

bag.move(0, -SPACE);

isCompleted();

}

}

return false;

} else if (type == BOTTOM\_COLLISION) {

for (int i = 0; i < baggs.size(); i++) {

Baggage bag = (Baggage) baggs.get(i);

if (soko.isBottomCollision(bag)) {

for (int j = 0; j < baggs.size(); j++) {

Baggage item = (Baggage) baggs.get(j);

if (!bag.equals(item)) {

if (bag.isBottomCollision(item)) {

return true;

}

}

if (checkWallCollision(bag,

BOTTOM\_COLLISION)) {

return true;

}

}

bag.move(0, SPACE);

isCompleted();

}

}

}

return false;

}

public void isCompleted() {

int num = baggs.size();

int compl = 0;

for (int i = 0; i < num; i++) {

Baggage bag = (Baggage) baggs.get(i);

for (int j = 0; j < num; j++) {

Area area = (Area) areas.get(j);

if (bag.x() == area.x()

&& bag.y() == area.y()) {

compl += 1;

}

}

}

if (compl == num) {

completed = true;

repaint();

}

}

public void restartLevel() {

areas.clear();

baggs.clear();

walls.clear();

initWorld();

if (completed) {

completed = false;

}

}

}

The game is simplified. It only provides the very basic functionality. The code is than easier to understand. The game has one level.

private final int OFFSET = 30;

private final int SPACE = 20;

private final int LEFT\_COLLISION = 1;

private final int RIGHT\_COLLISION = 2;

private final int TOP\_COLLISION = 3;

private final int BOTTOM\_COLLISION = 4;

The wall image size is 20x20px. This reflects the SPACE constant. The OFFSET is the distance between the borders of the window and the game world. There are four types of collisions. Each one is represented by a numerical constant.

private ArrayList walls = new ArrayList();

private ArrayList baggs = new ArrayList();

private ArrayList areas = new ArrayList();

The walls, baggs and areas are special containers, which will hold all the walls, baggs and areas of the game.

private String level =

" ######\n"

+ " ## #\n"

+ " ##$ #\n"

+ " #### $##\n"

+ " ## $ $ #\n"

+ "#### # ## # ######\n"

+ "## # ## ##### ..#\n"

+ "## $ $ ..#\n"

+ "###### ### #@## ..#\n"

+ " ## #########\n"

+ " ########\n";

This is the level of the game. Except for the space, there are five characters. The hash (#) stands for a wall. The dollar ($) represents the box to move. The dot (.) character represents the place where we must move the box. The at character (@) is the sokoban. And finally the new line character (\n) starts a new row of the world.

public final void initWorld() {

int x = OFFSET;

int y = OFFSET;

...

The initWorld() method initiates the game world. It goes through the level string and fills the above mentioned lists.

} else if (item == '$') {

b = new Baggage(x, y);

baggs.add(b);

x += SPACE;

In case of the dollar character, we create a Baggage object. The object is appended to the baggs list. The x variable is increased accordingly.

public void buildWorld(Graphics g) {

...

The buildWorld() method draws the game world on the window.

ArrayList world = new ArrayList();

world.addAll(walls);

world.addAll(areas);

world.addAll(baggs);

world.add(soko);

We create a world list which includes all objects of the game.

for (int i = 0; i < world.size(); i++) {

Actor item = (Actor) world.get(i);

if ((item instanceof Player)

|| (item instanceof Baggage)) {

g.drawImage(item.getImage(), item.x() + 2, item.y() + 2, this);

} else {

g.drawImage(item.getImage(), item.x(), item.y(), this);

}

...

We iterate through the world container and draw the objects. The player and the baggage images are a bit smaller. We add 2px to their coordinates to center them.

if (completed) {

g.setColor(new Color(0, 0, 0));

g.drawString("Completed", 25, 20);

}

If the level is completed, we draw "Completed" in the upper left corner of the window.

if (key == KeyEvent.VK\_LEFT) {

if (checkWallCollision(soko,

LEFT\_COLLISION)) {

return;

}

if (checkBagCollision(LEFT\_COLLISION)) {

return;

}

soko.move(-SPACE, 0);

...

Inside the keyPressed() method, we check what keys were pressed. We control the sokoban object with the cursor keys. If we press the left cursor key, we check if the sokoban collides with a wall or with a baggage. If it does not, we move the sokoban to the left.

} else if (key == KeyEvent.VK\_R) {

restartLevel();

}

We restart the level if we press the R key.

if (type == LEFT\_COLLISION) {

for (int i = 0; i < walls.size(); i++) {

Wall wall = (Wall) walls.get(i);

if (actor.isLeftCollision(wall)) {

return true;

}

}

return false;

...

The checkWallCollision() method was created to ensure that the sokoban or a baggage do not pass the wall. There are four types of collisions. The above lines check for the left collision.

private boolean checkBagCollision(int type) {

}

The checkBagCollision() is a bit more involved. A baggage can collide with a wall, with a sokoban object or with another baggage. The baggage can be moved only if it collides with a sokoban and does not collide with another baggage or a wall. When the baggage is moved, it is time to check if the level is completed by calling the isCompleted() method.

for (int i = 0; i < num; i++) {

Baggage bag = (Baggage) baggs.get(i);

for (int j = 0; j < num; j++) {

Area area = (Area) areas.get(j);

if (bag.x() == area.x()

&& bag.y() == area.y()) {

compl += 1;

}

}

}

The isCompleted() method checks if the level is completed. We get the number of bags. We compare the x, y coordinates of all the bags and the destination areas.

if (compl == num) {

completed = true;

repaint();

}

The game is finished, when the completed variable equals the number of bags in the game.

public void restartLevel() {

areas.clear();

baggs.clear();

walls.clear();

initWorld();

if (completed) {

completed = false;

}

}

If we do some bad move, we can restart the level. We delete all objects from the important lists and initiate the world again. The completed variable is set to false.

Actor.java

package sokoban;

import java.awt.Image;

public class Actor {

private final int SPACE = 20;

private int x;

private int y;

private Image image;

public Actor(int x, int y) {

this.x = x;

this.y = y;

}

public Image getImage() {

return this.image;

}

public void setImage(Image img) {

image = img;

}

public int x() {

return this.x;

}

public int y() {

return this.y;

}

public void setX(int x) {

this.x = x;

}

public void setY(int y) {

this.y = y;

}

public boolean isLeftCollision(Actor actor) {

if (((this.x() - SPACE) == actor.x()) &&

(this.y() == actor.y())) {

return true;

} else {

return false;

}

}

public boolean isRightCollision(Actor actor) {

if (((this.x() + SPACE) == actor.x())

&& (this.y() == actor.y())) {

return true;

} else {

return false;

}

}

public boolean isTopCollision(Actor actor) {

if (((this.y() - SPACE) == actor.y()) &&

(this.x() == actor.x())) {

return true;

} else {

return false;

}

}

public boolean isBottomCollision(Actor actor) {

if (((this.y() + SPACE) == actor.y())

&& (this.x() == actor.x())) {

return true;

} else {

return false;

}

}

}

This is the Actor class. The class is a base class for other actors in the game. It encapsulates the basic functionality of an object in the Sokoban game.

public boolean isLeftCollision(Actor actor) {

if (((this.x() - SPACE) == actor.x()) &&

(this.y() == actor.y())) {

return true;

} else {

return false;

}

}

This method checks if the actor collides with another actor (wall, baggage, sokoban) to the left.

Wall.java

package sokoban;

import java.awt.Image;

import java.net.URL;

import javax.swing.ImageIcon;

public class Wall extends Actor {

private Image image;

public Wall(int x, int y) {

super(x, y);

URL loc = this.getClass().getResource("wall.png");

ImageIcon iia = new ImageIcon(loc);

image = iia.getImage();

this.setImage(image);

}

}

This is the Wall class. It inherits from the Actor class. Upon construction, it loads a wall image from the filesystem.

Player.java

package sokoban;

import java.awt.Image;

import java.net.URL;

import javax.swing.ImageIcon;

public class Player extends Actor {

public Player(int x, int y) {

super(x, y);

URL loc = this.getClass().getResource("sokoban.png");

ImageIcon iia = new ImageIcon(loc);

Image image = iia.getImage();

this.setImage(image);

}

public void move(int x, int y) {

int nx = this.x() + x;

int ny = this.y() + y;

this.setX(nx);

this.setY(ny);

}

}

This is the Player class. It is the class to create the sokoban object.

public void move(int x, int y) {

int nx = this.x() + x;

int ny = this.y() + y;

this.setX(nx);

this.setY(ny);

}

This class has a move() method, which moves the object inside the world.

Baggage.java

package sokoban;

import java.awt.Image;

import java.net.URL;

import javax.swing.ImageIcon;

public class Baggage extends Actor {

public Baggage(int x, int y) {

super(x, y);

URL loc = this.getClass().getResource("baggage.png");

ImageIcon iia = new ImageIcon(loc);

Image image = iia.getImage();

this.setImage(image);

}

public void move(int x, int y) {

int nx = this.x() + x;

int ny = this.y() + y;

this.setX(nx);

this.setY(ny);

}

}

This is the class for the Baggage object. This object is movable, so it has the move() method also.

Area.java

package sokoban;

import java.awt.Image;

import java.net.URL;

import javax.swing.ImageIcon;

public class Area extends Actor {

public Area(int x, int y) {

super(x, y);

URL loc = this.getClass().getResource("area.png");

ImageIcon iia = new ImageIcon(loc);

Image image = iia.getImage();

this.setImage(image);

}

}

This is the Area class. It is the object, on which we try to place the baggages.

Sokoban.java

package sokoban;

import javax.swing.JFrame;

public final class Sokoban extends JFrame {

private final int OFFSET = 30;

public Sokoban() {

InitUI();

}

public void InitUI() {

Board board = new Board();

add(board);

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

setSize(board.getBoardWidth() + OFFSET,

board.getBoardHeight() + 2\*OFFSET);

setLocationRelativeTo(null);

setTitle("Sokoban");

}

public static void main(String[] args) {

Sokoban sokoban = new Sokoban();

sokoban.setVisible(true);

}

}

This is the main class.

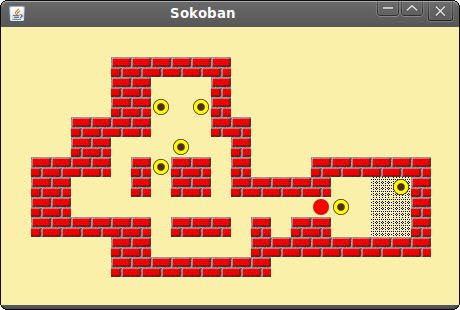


Figure: Sokoban