МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ  
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования   
**«Национальный исследовательский   
Нижегородский государственный университет им. Н.И. Лобачевского»**

**(ННГУ)**

**Институт информационных технологий, математики и механики**

Направление подготовки: «Программная инженерия»

**ОТЧЁТ**

по проектной работе

**ИГРА В ПЯТНАДЦАТЬ**

**Выполнил:** студент группы

382008-1

\_\_\_А.Д. Кандрин \_\_\_\_\_\_\_

Подпись

**Проверил:** младший научный сотрудник

\_\_\_\_Д.А. Карчков\_\_\_\_\_\_\_\_

Подпись

Нижний Новгород  
2023 г.

## Содержание

[1. Постановка задачи 3](#_Toc27873363)

[2. ХОД РАБОТЫ](#_Toc27873364) 4

[3. ЗАКЛЮЧЕНИЕ](#_Toc27873365) 12

[4. ЛИТЕРАТУРА](#_Toc27873374) 13

## Постановка задачи.

Разработать приложение, позволяющее играть в игру под названием “Пятнашки” (или “Игра в пятнадцать”).

## Ход работы.

1. Создадим компонент, отвечающий за отображение единственной игровой ячейки. Добавим этому компоненту сигнал clicked, позволяющий обработать нажатие по этой ячейке.

Item {

id: item

property int value

property color itemColor

signal clicked(var signalValue)

Rectangle

{

width: parent.width

height: parent.height

color: itemColor

Text

{

anchors.centerIn: parent

text: value

visible: value !== 0

}

MouseArea

{

anchors.fill: parent

onClicked:

{

if (item.value != 0)

item.clicked(item.value);

}

}

}

}

1. Создадим компонент, отвечающий за отображение игрового поля. Вынесем его основные свойства (размер поля (4 на 4), размер ячейки) в качестве свойств на верхний уровень. Для игры используем javascript-модель данных, потому что она удобна в генерации.  
   Модель данных у нас будет состоять из элементов вида {cellValue: …, color: …}. cellValue – это значение в текущей ячейке. Color – цвет ячейки.  
   Тогда функция генерации модели данных выглядит следующим образом:

function generateGameModel(size)

{

var model = [];

var entriesToAdd = size \* size; // сколько нужно добавить записей в модель данных

var cellValues = []; // массив индексов [0; entriesToAdd)

for(var i = 0; i < entriesToAdd; ++i)

{

cellValues.push(i);

}

while (entriesToAdd > 0)

{

var cellIndex = randInt(entriesToAdd);

var currentValue = cellValues[cellIndex];

model.push({cellValue: currentValue, color : currentValue !== 0 ? getColor() : "transparent" });

cellValues.splice(cellIndex, 1);

entriesToAdd--;

}

console.log("Created game model size = " + model.length);

return model;

}

1. Здесь берется массив из значений от 0 до 15, и генерируется какая-то его перестановка (случайным образом). Но такой подход не совсем верный, потому что только половина всех перестановок головоломки имеет решение. Соответственно, если мы хотим, чтобы игру всегда можно было закончить, надо иметь возможность создавать решаемую модель.  
   Для этого введем функцию проверки решаемости головоломки.

// Функция проверки разрешимости

function canSolve(model, size)

{

var cellValues = []

for (var i = 0; i < model.length; ++i)

cellValues.push(model[i].cellValue);

// Пусть плитка с числом i расположена до k плиток с числами, меньшими i; тогда обозначим

// Тогда обозначим n[i] = count.

// В частности, если после плитки с числом i нет плиток с числами, меньшими i, то

// n[i] = 0

var n = []; // массив n

// заполняем нулями

for (i = 0; i < cellValues.length; ++i)

n.push(0);

for (i = 0; i < cellValues.length; ++i)

{

var currentValue = cellValues[i];

// ищем (спереди) количество (ненулевых) чисел, меньших currentValue

var count = 0;

for (var j = i + 1; j < cellValues.length; ++j)

{

var nextValue = cellValues[j];

if (nextValue !== 0 && nextValue < currentValue)

count++;

}

n[currentValue] = count;

}

// Также введём число e — номер ряда пустой клетки (считая с 1).

var e = getTwoDimIndex(getEmptyIndex(model), size).x + 1;

var N = 0;

for (i = 0; i < cellValues.length; ++i)

{

N += n[i];

}

N += e;

return N % 2 == 0;

}

1. В соответствии с данной функцией напишем функцию по генерации разрешимой модели:

function createSolvableGame(size)

{

var model = generateGameModel(size);

while (!canSolve(model, size))

model = generateGameModel(size);

return model

}

1. Теперь добавим к нашей модели представление SilicaGridView. В качестве делагата используем ранее созданный компонент ячейки. Добавим обработчик на его сигнал clicked. Суть работы обработчика следующая. Из сигнала нам приходит значение нажатого элемента модели. По этому значению мы находим индекс элемента в модели данных. По индексу элемента в одномерной модели данных вычисляем индекс элемента, какой бы он был, если бы модель данных была бы двумерная. Также находим двумерный индекс пустой ячейки (ячейки со значением ноль). Если нажатый элемент и пустой элемент – соседи, то в модели данных меняем их местами. После этого переприсваиваем модель данных в представление, чтобы принудительно её обновить, а также проверяем, достигнуто ли условие победы.

Column

{

width: parent.width

SilicaGridView

{

id: view

model: gameModel

width: cellSize \* fieldSize

height: cellSize \* fieldSize

cellHeight: cellSize

cellWidth: cellSize

delegate: Cell {

value: modelData.cellValue

width: cellSize

height: cellSize

itemColor: modelData.color

onClicked:

{

restartButton.text = "Начать заново";

var valueIndex = getIndex(signalValue, gameModel);

var emptyIndex = getEmptyIndex(gameModel);

if (valueIndex !== -1 && emptyIndex !== -1)

{

var twoDimIndex = getTwoDimIndex(valueIndex, fieldSize);

var emptyTwoDimIndex = getTwoDimIndex(emptyIndex, fieldSize);

console.log("My index = " + twoDimIndex);

console.log("Empty index = " + emptyTwoDimIndex);

var canSwap = isNeighborIndices(twoDimIndex, emptyTwoDimIndex);

if (canSwap)

{

console.log("Is neighbors");

swapElements(gameModel, valueIndex, emptyIndex);

for (var i in gameModel)

{

console.log("val = " + gameModel[i].cellValue);

}

}

}

if (checkWin(gameModel))

{

winLabel.visible = true;

}

view.model = gameModel;

}

}

}

Label

{

id: winLabel

visible: false

width: parent.width

text: "Вы выиграли."

}

Button

{

id: restartButton

width: parent.width

text: "Начать заново"

onClicked: {

if (text === "Начать заново")

{

text = "Вы уверены?";

}

else if (text === "Вы уверены?")

{

text = "Абсолютно уверены?";

}

else if (text === "Абсолютно уверены?")

{

text = "Начать заново";

winLabel.visible = false;

gameModel = createSolvableGame(fieldSize);

view.model = gameModel;

}

}

}

}

1. С реализацией всех дополнительных функций этот компонент выглядит следующим образом:

Page {

id: page

property int fieldSize : 4

property int cellSize : 150

property var gameModel: []

function randInt(max) // return number in range: [0; max)

{

return Math.floor(Math.random() \* max);

}

function getColor()

{

var res = randInt(256 \* 256 \* 256 - 1).toString(16);

while (res.length < 6)

res = '0' + res;

console.log("Color = " + res);

return "#" + res;

}

function getIndex(value, model)

{

var valueIndex = -1;

for (var i in model)

{

if (model[i].cellValue === value)

{

valueIndex = i;

break;

}

}

console.log("Value index = " + valueIndex);

return valueIndex;

}

function getEmptyIndex(model)

{

return getIndex(0, model);

}

function getTwoDimIndex(oneDimIndex, size)

{

var res = { x: Math.floor(oneDimIndex / size), y : oneDimIndex % size };

console.log("x = " + res.x, "; y = " + res.y + " (" + "oneDimIndex = " + oneDimIndex + "; size = " + size + ")");

return res;

}

function generateGameModel(size)

{

var model = [];

var entriesToAdd = size \* size; // сколько нужно добавить записей в модель данных

var cellValues = []; // массив индексов [0; entriesToAdd)

for(var i = 0; i < entriesToAdd; ++i)

{

cellValues.push(i);

}

while (entriesToAdd > 0)

{

var cellIndex = randInt(entriesToAdd);

var currentValue = cellValues[cellIndex];

model.push({cellValue: currentValue, color : currentValue !== 0 ? getColor() : "transparent" });

cellValues.splice(cellIndex, 1);

entriesToAdd--;

}

console.log("Created game model size = " + model.length);

return model;

}

// Функция проверки разрешимости

function canSolve(model, size)

{

var cellValues = []

for (var i = 0; i < model.length; ++i)

cellValues.push(model[i].cellValue);

// Пусть плитка с числом i расположена до k плиток с числами, меньшими i; тогда обозначим

// Тогда обозначим n[i] = count.

// В частности, если после плитки с числом i нет плиток с числами, меньшими i, то

// n[i] = 0

var n = []; // массив n

// заполняем нулями

for (i = 0; i < cellValues.length; ++i)

n.push(0);

for (i = 0; i < cellValues.length; ++i)

{

var currentValue = cellValues[i];

// ищем (спереди) количество (ненулевых) чисел, меньших currentValue

var count = 0;

for (var j = i + 1; j < cellValues.length; ++j)

{

var nextValue = cellValues[j];

if (nextValue !== 0 && nextValue < currentValue)

count++;

}

n[currentValue] = count;

}

// Также введём число e — номер ряда пустой клетки (считая с 1).

var e = getTwoDimIndex(getEmptyIndex(model), size).x + 1;

var N = 0;

for (i = 0; i < cellValues.length; ++i)

{

N += n[i];

}

N += e;

return N % 2 == 0;

}

function isNeighborIndices(twoDimIndex1, twoDimIndex2)

{

var xDiff = Math.abs(twoDimIndex1.x - twoDimIndex2.x);

var yDiff = Math.abs(twoDimIndex1.y - twoDimIndex2.y);

return xDiff <= 1 && yDiff <= 1 && (twoDimIndex1.x === twoDimIndex2.x || twoDimIndex1.y === twoDimIndex2.y);

}

function swapElements(array, index1, index2)

{

var temp = array[index1];

array[index1] = array[index2];

array[index2] = temp;

}

function checkWin(model)

{

for (var i = 0; i < model.length - 1; ++i)

{

if (model[i].cellValue !== i + 1)

return false;

}

return true;

}

Column

{

width: parent.width

SilicaGridView

{

id: view

model: gameModel

width: cellSize \* fieldSize

height: cellSize \* fieldSize

cellHeight: cellSize

cellWidth: cellSize

delegate: Cell {

value: modelData.cellValue

width: cellSize

height: cellSize

itemColor: modelData.color

onClicked:

{

restartButton.text = "Начать заново";

var valueIndex = getIndex(signalValue, gameModel);

var emptyIndex = getEmptyIndex(gameModel);

if (valueIndex !== -1 && emptyIndex !== -1)

{

var twoDimIndex = getTwoDimIndex(valueIndex, fieldSize);

var emptyTwoDimIndex = getTwoDimIndex(emptyIndex, fieldSize);

console.log("My index = " + twoDimIndex);

console.log("Empty index = " + emptyTwoDimIndex);

var canSwap = isNeighborIndices(twoDimIndex, emptyTwoDimIndex);

if (canSwap)

{

console.log("Is neighbors");

swapElements(gameModel, valueIndex, emptyIndex);

for (var i in gameModel)

{

console.log("val = " + gameModel[i].cellValue);

}

}

}

if (checkWin(gameModel))

{

winLabel.visible = true;

}

view.model = gameModel;

}

}

}

Label

{

id: winLabel

visible: false

width: parent.width

text: "Вы выиграли."

}

Button

{

id: restartButton

width: parent.width

text: "Начать заново"

onClicked: {

if (text === "Начать заново")

{

text = "Вы уверены?";

}

else if (text === "Вы уверены?")

{

text = "Абсолютно уверены?";

}

else if (text === "Абсолютно уверены?")

{

text = "Начать заново";

winLabel.visible = false;

gameModel = createSolvableGame(fieldSize);

view.model = gameModel;

}

}

}

}

function createSolvableGame(size)

{

var model = generateGameModel(size);

while (!canSolve(model, size))

model = generateGameModel(size);

return model

}

Component.onCompleted:

{

winLabel.visible = false;

gameModel = createSolvableGame(fieldSize);

view.model = gameModel;

}

}

1. **Заключение.**

Мы ещё раз повторили изученные в курсе материалы и применили их на практике. В этой работе я с использованием материалов курса написал приложение для игры в “Пятнашки”.

**4. Литература.**

1. Документация Qt: сайт. — URL: https://doc.qt.io (дата обращения: 01.03.2022). — Текст: электронный.