МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ, СВЯЗИ И МАССОВЫХ КОММУНИКАЦИЙ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

КАФЕДРА ПРИКЛАДНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ

Курсовая работа

по дисциплине «Объектно-ориентированное программирование»

Тема: «Движение составного графического объекта, управляемого с клавиатуры с помощью стрелок»

Выполнил: студент группы ИС-142

Наумов А.А.

Проверил: ассистент

Бублей Д.А.

Новосибирск – 2022

Содержание

1. Постановка задачи.

10. *Движение составного графического объекта, управляемого с клавиатуры с помощью стрелок.

Графика будет реализована при помощи SFML. Будет специальный класс, который будет наследовать все классы фигур. И будет создан класс, который будет реагировать на нажатие клавиш и хранить в себе массив с фигурами. Также этот класс будет отвечать за перемещение составного графического объекта.

2. Технологии ООП

Инкапсуляция

Инкапсуляция нужна, чтобы все поля данных не доступны из внешних функций. В моей реализации это пригодилось для определения скорости персонажа, размера и цвета фигур.

```
void Shape::setSpeed(sf::Vector2f _speed)
{
    speed = _speed;
}
```
z

Наследование

Наследование — это механизм создания нового класса на основе уже существующего. При этом к существующему классу могут быть добавлены новые элементы, либо существующие функции могут быть изменены. Эта технология активно используется в курсовой работе.

```
class Ellipse : public Shape
{
private:
    sf::Vector2f radius;
    const int pointCount = 90;
    void updatePoints(void);
    void setPointCount();
    void setPoint();
```

Полиморфизм

Полиморфизм – свойство, которое позволяет создавать классы с одинаковым интерфейсом. Однако их методы решают схожие задачи по-разному. Это свойство нам понадобилось, чтобы переписать метод движения для персонажа, которым управляет персонаж. В метод нужно было изменить так, чтобы персонаж не мог уйти за рамки экрана.

```
void Rectangle::move(sf::Vector2i win, sf::Vector2f offset)
{
    sf::Vector2f pos = getPosition();
    if (pos.x < 50 && getSpeed().x < 0)
        return;
    else if (pos.x > 670 && getSpeed().x > 0)
        return;
    else if (pos.y < 30 && getSpeed().y < 0)
        return;
    else if (pos.y > 460 && getSpeed().y > 0)
        return;

    setPosition(getPosition() + offset);
}
```

```
void Shape::move(int winX, int winY, float offsetX, float offsetY)
{
    move(sf::Vector2i(winX, winY), sf::Vector2f(offsetX, offsetY)); // Основное движение
}
```

Конструктор.

Конструктор предназначен для инициализации объектов класса. И в курсовой работе эта технология ООП пригодилась. При создании новых фигур не нужно вызывать дополнительные функции, которые бы задали им размер и скорость движения. Это происходит автоматически после их создания.

```
Human::Human(float Speed)
{
    speed = Speed;
    setFillColor(sf::Color::Green);
    setSize(70, 140);
    child.push_back(new Rectangle());
    child.push_back(new Rectangle());
    child.push_back(new Rectangle());
    child.push_back(new Rectangle());
    child.push_back(new Ellipse());
}
```

Перегрузка конструкторов

Перегрузка конструкторов Использована в работе для того, чтобы в случае, когда при создании персонажа не указывается скорость, ему задавалась скорость, которая установлена по умолчанию.

```
Human::Human() : speed{4}
{
    // speed = 3.0;
    setFillColor(sf::Color::Green);
    setSize(70, 140);
    child.push_back(new Rectangle());
    child.push_back(new Rectangle());
    child.push_back(new Rectangle());
    child.push_back(new Rectangle());
    child.push_back(new Ellipse());
}

Human::Human(float Speed)
{
    speed = Speed;
    setFillColor(sf::Color::Green);
    setSize(70, 140);
    child.push_back(new Rectangle());
    child.push_back(new Rectangle());
    child.push_back(new Rectangle());
    child.push_back(new Rectangle());
    child.push_back(new Ellipse());
}
```

Списки инициализации

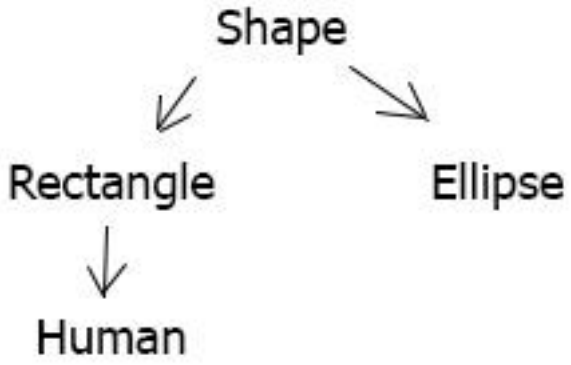
Список инициализации использован для установления скорости персонажа по умолчанию при необходимости.

```
Human::Human() : speed{4}
```

Структура классов

Основным классом является Shape. В этом классе содержатся методы для перемещения фигур и переменная со скоростью фигуры. Сама фигура в классе не создаётся.

Этот класс наследуется всеми фигурами, используемыми в работе. У каждой фигуры свой класс: Ellipse, Rectangle. Также у фигур есть методы для определения их размера.



Класс Human – класс составного графического объекта, которым управляет пользователь при помощи клавиатуры. Этот класс наследует класс эллипса и имеет массив фигур, из которых создаётся составной графический объект, и методы для реагирования на нажатие клавиш и перемещения графического объекта.

4. Программная реализация

Каждой фигуре можно задать скорость при помощи метода setSpeed.
Метод move перемещает фигуры при помощи переопределения положения фигуры. Метод прибавляется текущему положение смещение.

В конструкторах фигур определён размер фигур и их цвет по умолчанию. Изменить размер можно методом setSize. У эллипса определяется его радиус, для этого существует функция setRadius.

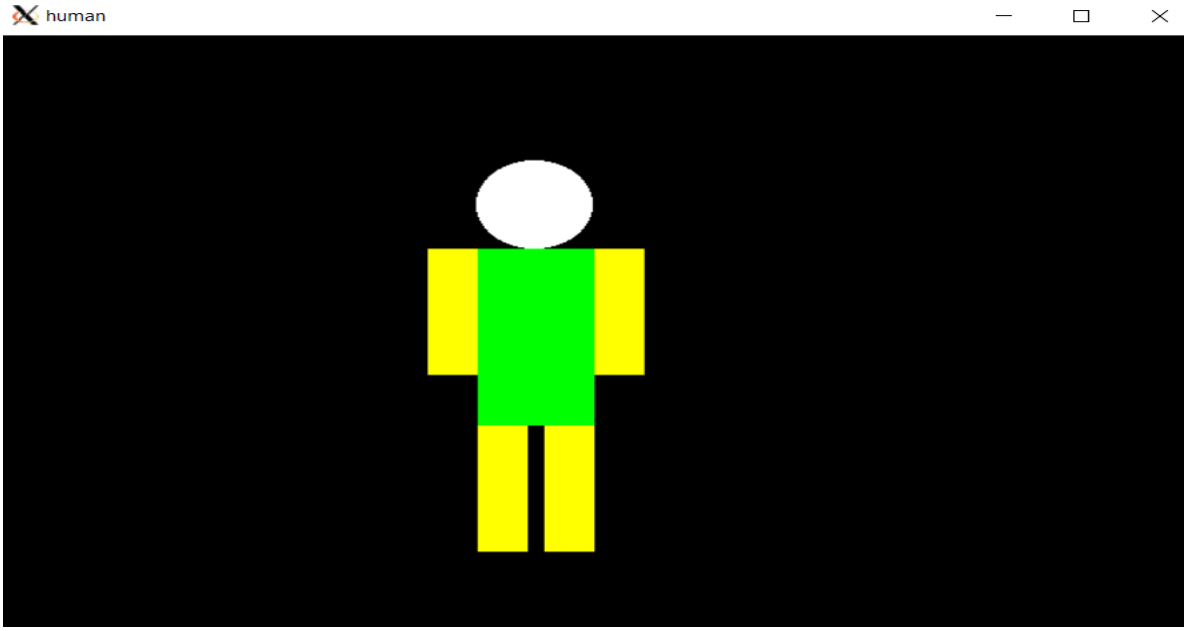
При создании составного графического объекта определяется его скорость и создаются его дочерние фигуры. Фигуры хранятся в динамическом массиве, и чтобы получить указатель на первый элемент используется функция getChild.

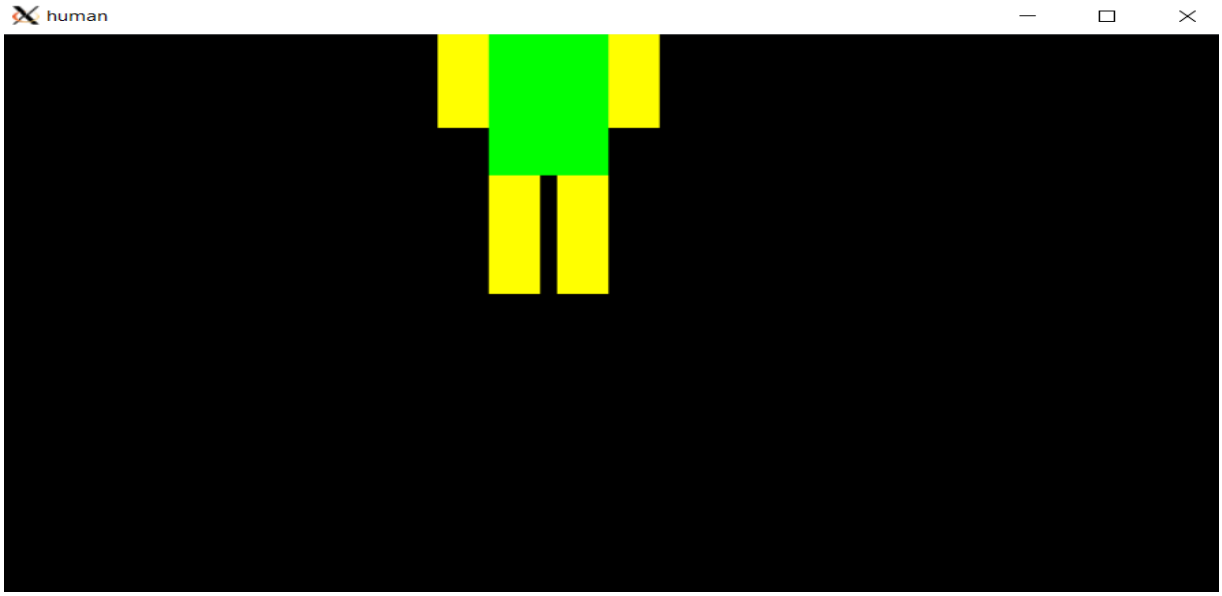
При запуске программы создаётся окно и затем объект класса Human и указатель на его дочерние фигуры. Далее объект устанавливается случайное место на экране и запускается цикл, который отслеживает нажатие клавиш и отрисовывает фигруры. Цикл заканчивает работу после закрытия окна.

Отслеживание нажатия клавиш работает при помощи SFML. При нажатии на стрелочки программа определяет в какую сторону нужно переместиться. Затем перемещается объект и его дочерние фигуры. Дочерние фигуры перемещаются при помощи функции moveElements, им указывается положение относительно родительской фигуры.

После перемещения фигур, программа начинает их отрисовывать. Процесс начинается с родительской фигуры, чтобы она не перекрывала дочерние.

5. Результат работы





6. Заключение

При создании курсовой работы, я стал лучше понимать, как работают классы и графика sfml, этим заданием я подкрепил знания о технологиях ООП.

Благодаря наследованию классов было легко реализовать перемещение разных фигур. Полиморфизм позволил реализовать для фигур с одинаковым интерфейсом реализовать функции, которые выполняют свою работу по-разному. А перегрузка позволила передавать в методы разные значения, что пригодилось при перемещении объекта класса Human. В работе реализован необходимый минимум: Инкапсуляция, наследования, полиморфизм, конструктор, перегрузка конструкторов.

7. Используемые источники

1. Лафоре Р. "Объектно-ориентированное программирование в С++"
2. Бьерн Страуструп. Язык программирования С++
3. Бертран Мейер «Почувствуй класс. Учимся программировать хорошо с объектами и контрактами»
4. Гради Буч «Объектно-ориентированный анализ и проектирование с примерами приложений»
5. Мэтт Вайсфельд «Объектно-ориентированное мышление»

8. Приложение. Листинг

Main.c

```
#include "shapes.hpp"
#include <SFML/Graphics.hpp>
#include <time.h>

#define FIGURS 1

int main(void)
{
    srand(time(NULL));
    sf::Vector2i win(720, 480);
    sf::RenderWindow window(sf::VideoMode(win.x, win.y), "human");

    Human shapes(10.0);
    // Human shapes;
    std::vector<Shape *> child = shapes.getChild();

    shapes.setPosition(rand() % win.x, rand() % win.y);
    shapes.moveElements(&shapes);

    // sf::Clock clock;
    while (window.isOpen())
    {
        sf::Event e;
        while (window.pollEvent(e))
            if (e.type == sf::Event::Closed)
                window.close();

        if (sf::Keyboard::isKeyPressed(sf::Keyboard::Left))
            shapes.control(win, LEFT, &shapes);

        if (sf::Keyboard::isKeyPressed(sf::Keyboard::Right))
            shapes.control(win, RIGHT, &shapes);

        if (sf::Keyboard::isKeyPressed(sf::Keyboard::Up))
            shapes.control(win, TOP, &shapes);

        if (sf::Keyboard::isKeyPressed(sf::Keyboard::Down))
            shapes.control(win, BOT, &shapes);

        window.clear(sf::Color::Black);

        window.draw(shapes);
        for (int i = 0; i < 5; i++)
            window.draw(*child[i]);

        window.display();
    }
    return EXIT_SUCCESS;
}
```

Shapes.cpp

```
#include "shapes.hpp"
#include <cmath>
#include <SFML/Graphics/ConvexShape.hpp>
```

```
// shape
Shape::Shape() : sf::ConvexShape()
{
    speed.x = 0;
    speed.y = 0;
}
void Shape::setSpeed(sf::Vector2f _speed)
{
    speed = _speed;
}
void Shape::setSpeed(float _x, float _y)
{
    setSpeed(sf::Vector2f(_x, _y));
}
sf::Vector2f Shape::getSpeed(void)
{
    return speed;
}

void Shape::move(sf::Vector2i win, sf::Vector2f offset)
{
    setFillColor(sf::Color::Green);
}

void Shape::move(int winX, int winY, float offsetX, float offsetY)
{
    move(sf::Vector2i(winX, winY), sf::Vector2f(offsetX, offsetY)); //
Основное движение
}

void Shape::move(sf::Vector2i win, float offsetX, float offsetY)
{
    move(win, sf::Vector2f(offsetX, offsetY));
}
void Shape::move(int winX, int winY, sf::Vector2f offset)
{
    move(sf::Vector2i(winX, winY), offset);
}

// ellipse
Ellipse::Ellipse(sf::Vector2f _radius) : Shape()
{
    setRadius(_radius);
}

Ellipse::Ellipse(float _radiusX, float _radiusY) : Shape()
{
    setRadius(_radiusX, _radiusY);
}

Ellipse::Ellipse() : Shape()
{
    setFillColor(sf::Color::White);
    setRadius(35, 35);
}

void Ellipse::updatePoints()
{
    if (getPointCount() != pointCount)
```

```
        Shape::setPointCount(pointCount);

    float rad, x, y;
    for (int i = 0; i < pointCount; i++)
    {
        rad = (360 / pointCount * i) / (360 / M_PI / 2);
        x = cos(rad) * radius.x;
        y = sin(rad) * radius.y;
        Shape::setPoint(i, sf::Vector2f(x, y));
    }
}

void Ellipse::setRadius(sf::Vector2f _radius)
{
    radius.x = (_radius.x) ? _radius.x : 0;
    radius.y = (_radius.y) ? _radius.y : 0;

    updatePoints();
}

void Ellipse::setRadius(float _radiusX, float _radiusY)
{
    setRadius(sf::Vector2f(_radiusX, _radiusY));
}

sf::Vector2f Ellipse::getRadius(void)
{
    return radius;
}

void Ellipse::move(sf::Vector2i win, sf::Vector2f offset)
{
    sf::Vector2f pos = getPosition();

    if (pos.x < 50 && getSpeed().x < 0)
        return;
    else if (pos.x > 670 && getSpeed().x > 0)
        return;
    else if (pos.y < 30 && getSpeed().y < 0)
        return;
    else if (pos.y > 460 && getSpeed().y > 0)
        return;

    setPosition(getPosition() + offset);
}

// rectangle
Rectangle::Rectangle(sf::Vector2f _size) : Shape()
{
    setSize(_size);
}

Rectangle::Rectangle(float _w, float _h) : Shape()
{
    setSize(_w, _h);
}

Rectangle::Rectangle() : Shape()
{
```

```
    setFillColor(sf::Color::Yellow);
    setSize(30, 100);
    setOrigin(getSize().x / 2, getSize().y / 2);
}

void Rectangle::updatePoints()
{
    if (getPointCount() != 4)
        Shape::setPointCount(4);
    Shape::setPoint(0, sf::Vector2f(0, 0));
    Shape::setPoint(1, sf::Vector2f(size.x, 0));
    Shape::setPoint(2, sf::Vector2f(size.x, size.y));
    Shape::setPoint(3, sf::Vector2f(0, size.y));
}

void Rectangle::setSize(sf::Vector2f _size)
{
    size.x = (_size.x) ? _size.x : 0;
    size.y = (_size.y) ? _size.y : 0;

    updatePoints();
}

void Rectangle::setSize(float _w, float _h)
{
    size.x = (_w) ? _w : 0;
    size.y = (_h) ? _h : 0;

    updatePoints();
}

sf::Vector2f Rectangle::getSize(void)
{
    return size;
}

void Rectangle::move(sf::Vector2i win, sf::Vector2f offset)
{
    sf::Vector2f pos = getPosition();
    if (pos.x < 50 && getSpeed().x < 0)
        return;
    else if (pos.x > 670 && getSpeed().x > 0)
        return;
    else if (pos.y < 30 && getSpeed().y < 0)
        return;
    else if (pos.y > 460 && getSpeed().y > 0)
        return;

    setPosition(getPosition() + offset);
}

Human::Human() : speed{4}
{
    // speed = 3.0;
    setFillColor(sf::Color::Green);
    setSize(70, 140);
    child.push_back(new Rectangle());
    child.push_back(new Rectangle());
    child.push_back(new Rectangle());
```

```
    child.push_back(new Rectangle());
    child.push_back(new Ellipse());
}

Human::Human(float Speed)
{
    speed = Speed;
    setFillColor(sf::Color::Green);
    setSize(70, 140);
    child.push_back(new Rectangle());
    child.push_back(new Rectangle());
    child.push_back(new Rectangle());
    child.push_back(new Rectangle());
    child.push_back(new Ellipse());
}

std::vector<Shape *> Human::getChild()
{
    return child;
}

void Human::moveElements(Shape* shapes)
{
    child[0]->setPosition(shapes->getPosition().x , shapes->getPosition().y +
140); //leg 1
    child[1]->setPosition(shapes->getPosition().x + 40, shapes-
>getPosition().y + 140);//leg 2
    child[2]->setPosition(shapes->getPosition().x - 30, shapes-
>getPosition().y);//arm 1
    child[3]->setPosition(shapes->getPosition().x + 70, shapes-
>getPosition().y);//arm 2
    child[4]->setPosition(shapes->getPosition().x + 19, shapes-
>getPosition().y - 85);//head
}

void Human::control(sf::Vector2i win, int Direction, Human* shapes)
{
    switch (Direction)
    {
    case LEFT:
        shapes->setSpeed(speed * -1, 0);
        break;

    case RIGHT:
        shapes->setSpeed(speed, 0);
        break;

    case TOP:
        shapes->setSpeed(0, speed * -1);
        break;

    case BOT:
        shapes->setSpeed(0, speed);
        break;

    default:
        return;
    }
```

```
    shapes->move(win, shapes->getSpeed());
    shapes->moveElements(shapes);
}
```

Shapes.hpp

```
#pragma once

#include <SFML/Graphics.hpp>
#include <iostream>

enum {
    LEFT = 1,
    RIGHT,
    TOP,
    BOT
};

class Shape : public sf::ConvexShape
{
private:
    sf::Vector2f speed;

public:
    Shape();
    void setSpeed(sf::Vector2f _speed);
    void setSpeed(float _x, float _y);
    sf::Vector2f getSpeed(void);
    virtual void move(sf::Vector2i win, sf::Vector2f offset);
    virtual void move(int winX, int winY, float offsetX, float offsetY);
    virtual void move(sf::Vector2i win, float offsetX, float offsetY);
    virtual void move(int winX, int winY, sf::Vector2f offset);
};

class Ellipse : public Shape
{
private:
    sf::Vector2f radius;
    const int pointCount = 90;
    void updatePoints(void);
    void setPointCount();
    void setPoint();

public:
    Ellipse(sf::Vector2f _radius);
    Ellipse(float _radiusX, float _radiusY);
    Ellipse();
    void setRadius(sf::Vector2f _radius);
    void setRadius(float _radiusX, float _radiusY);
    sf::Vector2f getRadius(void);
    void move(sf::Vector2i win, sf::Vector2f offset);
};

class Rectangle : public Shape
{
private:
    sf::Vector2f size;
    void updatePoints();
```

```
    void setPointCount();
    void setPoint();

public:
    Rectangle(sf::Vector2f _size);
    Rectangle(float _w, float _h);
    Rectangle();
    void setSize(sf::Vector2f _size);
    void setSize(float _w, float _h);
    sf::Vector2f getSize(void);
    void move(sf::Vector2i win, sf::Vector2f offset);
};

// Human
class Human : public Rectangle
{
private:
    float speed;
    std::vector<Shape *> child;
public:
    Human();
    Human(float Speed);
    void moveElements(Shape* shapes);
    void control(sf::Vector2i win, int Direction, Human* shapes);
    std::vector<Shape *> getChild();
};
```