Московский Авиационный Институт
(Национальный Исследовательский Университет)

Кафедра 806 «Вычислительная информатика и программирование»
Факультет: «Информационные технологии и прикладная математика»

Лабораторная работа
Дисциплина: «Объектно-ориентированное программирование»
III семестр
Задание 7: «Проектирование структуры классов»

Группа:	М8О-206Б-18, №27
Студент:	Шорохов Алексей Павлович
Преподаватель:	Журавлёв Андрей Андреевич
Оценка:	
Дата:	.12.2019

Москва, 2019

Задание

27.	Прямоугольник	Трапеция	Ромб

Адрес репозитория на GitHub

Код программы на C++

CMakeLists.txt
```
cmake_minimum_required(VERSION 3.0)
project(lab7)
set(CMAKE_CXX_STANDARD_REQUIRED YES)
set(CMAKE_CXX_STANDARD 14)
add_executable(lab7
  Source.cpp
  stdl.cpp
  circle.cpp
  brokenLine.cpp
  polygon.cpp
  rectangle.cpp
  rhombus.cpp
  trapezoid.cpp
  Document.cpp
)

add_subdirectory(lib/SDL2/)
target_link_libraries(lab7 SDL2-static)
target_include_directories(lab7 PRIVATE ${SDL2_INCLUDE_DIR})

add_subdirectory(lib/imgui/)
target_include_directories(imgui PRIVATE lib/SDL2/include/)
target_link_libraries(lab7 imgui)
```

Document.h
```
#pragma once

#ifndef D_DOCUMENT_H
#define D_DOCUMENT_H 1

#include <array>
#include <fstream>
#include <iostream>
#include <memory>
#include <vector>
```

```
#include <stack>

#include "sdl.h"
#include "imgui.h"

#include "rectangle.h"
#include "rhombus.h"
#include "trapezoid.h"
#include "brokenLine.h"
#include "circle.h"
#include "polygon.h"

struct Command;
struct CommandAdd;
struct CommandRemove;
struct Document;


struct Document {
public:
	Document() = default;

	void addFigure(std::unique_ptr<figure> fig);
	void removeFigure(int id);
	void removeByClick(vertex v);
	void undo();
	void Save(std::ofstream& os);
	void Load(std::ifstream& is);
	void render(const sdl::renderer& renderer);
	void clear();

	std::vector<std::shared_ptr<figure>> figures;
	std::stack<std::unique_ptr<Command>> commandStack;
};



struct Command {

	virtual ~Command() = default;

	virtual void undo() = 0;

};
```

```
struct CommandAdd : Command {


	int index__;
	Document * doc__ = new Document();

	CommandAdd(int index, Document * doc) : index__(index), doc__(doc) {}

	void undo() {
		(doc__ -> figures).erase((doc__ -> figures).begin() + index__);
	}

};

struct CommandRemove : Command {

	Document * doc__;

	int index__;
	std::shared_ptr<figure> figure__ = nullptr;

	CommandRemove(int index, std::shared_ptr<figure> figure_, Document *
doc) : index__(index), figure__(figure_), doc__(doc) {}

	void undo() {
		if (index__ > (doc__ -> figures).size() - 1)
			(doc__ -> figures).push_back(std::move(figure__));
		else
			(doc__ -> figures).insert((doc__ -> figures).begin() + index__,
std::move(figure__));
	}

};

#endif
```

Document.cpp

```
#include "Document.h"


void Document::addFigure(std::unique_ptr<figure> fig) {
	figures.emplace_back(std::move(fig));

	commandStack.push(std::make_unique<CommandAdd>(figures.size() - 1,
this));
```

```
}
void Document::removeFigure(int id) {
	//commandStack.push(std::make_unique<remove_command>(remove_com
mand(this, std::move(figures[id]), id)));
	//figures.pop_back();

	commandStack.push(std::make_unique<CommandRemove>(id, figures[id],
this));
	figures.erase(figures.begin() + id);

}
void Document::undo() {
	if (commandStack.size()) {

		commandStack.top() -> undo();

		commandStack.pop();

	}
}

void Document::removeByClick(vertex v) {
	std::vector<int> toDelete;
	for (int i = 0; i < figures.size(); i++) {
		if (figures[i] -> isPointInside(v)) {
			toDelete.push_back(i);
			//std::string type_ = active_builder -> getType();

			commandStack.push(std::make_unique<CommandRemove>(i,
figures[i], this));


		}
	}

	for (int i = 0; i < toDelete.size(); i++) {
		figures.erase(figures.begin() + toDelete[i] - i);
	}
}

void Document::render(const sdl::renderer& renderer) {
	for (const std::shared_ptr<figure>& figure : figures) {
		figure -> render(renderer);
	}
```

```
}

void Document::Save(std::ofstream& os) {
	for (const std::shared_ptr<figure>& figure : figures) {
		figure -> save(os);
	}
}

void Document::Load(std::ifstream& is) {

	figures.clear();
	std::string type;

	while(std::getline(is, type)) {
		if (type == "rectangle") {
			std::array<vertex, 4> vrt;
			for (int i = 0; i < 4; i++) {
				is >> vrt[i];
			}
			std::vector<int> colorTmp(3);
			for (int i = 0; i < 3; i++) {
				is >> colorTmp[i];
			}
			std::unique_ptr<figure> rect =
std::make_unique<rectangle>(vrt);
			rect -> setColor(colorTmp);
			figures.emplace_back(std::move(rect));
		} else if (type == "rhombus") {
			std::array<vertex, 4> vrt;
			for (int i = 0; i < 4; i++) {
				is >> vrt[i];
			}
			std::vector<int> colorTmp(3);
			for (int i = 0; i < 3; i++) {
				is >> colorTmp[i];
			}
			std::unique_ptr<figure> rhom =
std::make_unique<rhombus>(vrt);
			rhom -> setColor(colorTmp);
			figures.emplace_back(std::move(rhom));
		} else if (type == "trapezoid") {
			std::array<vertex, 4> vrt;
			for (int i = 0; i < 4; i++) {
				is >> vrt[i];
			}
```

```
                        std::vector<int> colorTmp(3);
                        for (int i = 0; i < 3; i++) {
                                is >> colorTmp[i];
                        }
                        std::unique_ptr<figure> trap =
std::make_unique<trapezoid>(vrt);
                        trap -> setColor(colorTmp);
                        figures.emplace_back(std::move(trap));
                } else if (type == "polygon") {
                        int sz;
                        is >> sz;
                        std::vector<vertex> vrt(sz);

                        for (int i = 0; i < sz; i++) {
                                is >> vrt[i];
                        }
                        std::vector<int> colorTmp(3);
                        for (int i = 0; i < 3; i++) {
                                is >> colorTmp[i];
                        }
                        std::unique_ptr<figure> poly =
std::make_unique<polygon>(vrt);
                        poly -> setColor(colorTmp);
                        figures.emplace_back(std::move(poly));
                } else if (type == "brokenLine") {
                        int sz;
                        is >> sz;
                        std::vector<vertex> vrt(sz);

                        for (int i = 0; i < sz; i++) {
                                is >> vrt[i];
                        }
                        std::vector<int> colorTmp(3);
                        for (int i = 0; i < 3; i++) {
                                is >> colorTmp[i];
                        }
                        std::unique_ptr<figure> bl =
std::make_unique<brokenLine>(vrt);
                        bl -> setColor(colorTmp);
                        figures.emplace_back(std::move(bl));
                } else if (type == "circle") {
                        std::vector<vertex> vrt(2);
                        is >> vrt[0] >> vrt[1];

                        std::vector<int> colorTmp(3);
```

```
                    for (int i = 0; i < 3; i++) {
                        is >> colorTmp[i];
                    }

                    std::unique_ptr<figure> crcl = std::make_unique<circle>(vrt);
                    crcl -> setColor(colorTmp);
                    figures.emplace_back(std::move(crcl));
                }

        }

}

void Document::clear() {

        while ( ! commandStack.empty() )
        {
        commandStack.pop();
        }

        figures.clear();

}
```

Source.cpp

```
#include <array>
//#include <fstream>
//#include <iostream>
#include <memory>
#include <vector>
#include <stack>

#include "sdl.h"
#include "imgui.h"

#include "rectangle.h"
#include "rhombus.h"
#include "trapezoid.h"
#include "brokenLine.h"
#include "circle.h"
#include "polygon.h"

#include "Document.h"

int main() {
```

```
	sdl::renderer renderer("Editor");
	bool quit = false;

	std::unique_ptr<builder> active_builder = nullptr;
	bool active_deleter = false;
	const int32_t file_name_length = 128;
	char file_name[file_name_length] = "";
	int32_t remove_id = 0;
	std::vector<int> color(3);

	Document currentDocument;

	while (!quit) {
		renderer.set_color(0, 0, 0);
		renderer.clear();

		sdl::event event;

		while (sdl::event::poll(event)) {
			sdl::quit_event quit_event;
			sdl::mouse_button_event mouse_button_event;
			if (event.extract(quit_event)) {
				quit = true;
				break;
			} else if (event.extract(mouse_button_event)) {
				if (active_builder && mouse_button_event.button() ==
sdl::mouse_button_event::left && mouse_button_event.type() ==
sdl::mouse_button_event::down) {
					std::unique_ptr<figure> figure = active_builder-
>add_vertex(vertex{mouse_button_event.x(), mouse_button_event.y()});
					if (figure) {
						figure -> setColor(color);


	currentDocument.addFigure(std::move(figure));

						active_builder = nullptr;
					}
				}
				if (active_builder && mouse_button_event.button() ==
sdl::mouse_button_event::right && mouse_button_event.type() ==
sdl::mouse_button_event::down) {
					std::unique_ptr<figure> figure = active_builder-
>add_vertex(vertex{-1, -1});
					if (figure) {
```

```
                                        figure -> setColor(color);


        currentDocument.addFigure(std::move(figure));

                                        active_builder = nullptr;
                                }
                        }
                        if (active_deleter && mouse_button_event.button() ==
sdl::mouse_button_event::left && mouse_button_event.type() ==
sdl::mouse_button_event::down) {


        currentDocument.removeByClick(vertex{mouse_button_event.x(),
mouse_button_event.y()});

                                active_deleter = false;
                        }
                }
        }

        currentDocument.render(renderer);

        /*

        TODO

        pointer to canvas in commands;

        Loader & saver

        */


        ImGui::Begin("Menu");
        if (ImGui::Button("New canvas")) {
                currentDocument.clear();
        }

        ImGui::InputText("File name", file_name, file_name_length - 1);

        if (ImGui::Button("Save")) {
                std::ofstream os(file_name);

                if (os) {
```

```
                currentDocument.Save(os);
            }

        }

        ImGui::SameLine();

        if (ImGui::Button("Load")) {
            std::ifstream is(file_name);

            if (is) {
                currentDocument.Load(is);
            }

        }


        ImGui::InputInt("R", &color[0]);
        ImGui::InputInt("G", &color[1]);
        ImGui::InputInt("B", &color[2]);

        if (ImGui::Button("Rectangle")) {
            active_builder = std::make_unique<rectangle_builder>();
        }
        if (ImGui::Button("Rhombus")) {
            active_builder = std::make_unique<rhombus_builder>();
        }
        if (ImGui::Button("Trapezoid")) {
            active_builder = std::make_unique<trapezoid_builder>();
        }
        if (ImGui::Button("Broken Line")) {
            active_builder = std::make_unique<brokenLine_builder>();
        }
        if (ImGui::Button("Circle")) {
            active_builder = std::make_unique<circle_builder>();
        }
        if (ImGui::Button("Polygon")) {
            active_builder = std::make_unique<polygon_builder>();
        }

        ImGui::InputInt("Remove id", &remove_id);
        if (ImGui::Button("Remove")) {
            if (remove_id >= 0 && remove_id <
(currentDocument.figures).size()) {
```

```
                        currentDocument.removeFigure(remove_id);

                }
            }
            if (ImGui::Button("Remove by click")) {
                active_deleter = true;
            }
            if (ImGui::Button("UNDO")) {

                currentDocument.undo();
            }

            ImGui::End();

            renderer.present();
        }


}
```

Builder.h

```
#ifndef D_BUILDER_H
#define D_BUILDER_H 1

#include "figure.h"

struct builder {
    virtual std::unique_ptr<figure> add_vertex(const vertex& v) = 0;
    virtual std::string getType() = 0;

    virtual ~builder() = default;

};

#endif //!D_BUILDER_H
```

Vertex.h

```
#ifndef D_VERTEX_H
#define D_VERTEX_H 1

#include <memory>
#include <fstream>
#include <iostream>
```

```
struct vertex {
	int32_t x, y;

};

inline std::istream& operator>> (std::istream& is, vertex& p) {
	is >> p.x >> p.y;
	return is;
}

#endif // !D_VERTEX_H
```

Trapezoid.h

```
#ifndef D_TRAPEZOID_H
#define D_TRAPEZOID_H 1

#include "builder.h"
#include "figure.h"

struct trapezoid : figure {
	trapezoid(const std::array<vertex, 4>& vertices);

	void setColor(std::vector<int> color) override;

	void render(const sdl::renderer& renderer) const override;

	void save(std::ostream& os) const override;

	bool isPointInside(vertex v) const override;

private:
	std::vector<int> color_;
	std::array<vertex, 4> vertices_;

};

struct trapezoid_builder : builder {
	std::unique_ptr<figure> add_vertex(const vertex& v) override;

	std::string getType();

private:
	int32_t n_ = 0;
	std::array<vertex, 4> vertices_;
```

```
};

#endif // !D_TRAPEZOID_H
```

Trapezoid.cpp

```
#include "trapezoid.h"

trapezoid::trapezoid(const std::array<vertex, 4>& vertices) : vertices_(vertices) {}

void trapezoid::setColor(std::vector<int> color) {
	for (int i = 0; i < 3; i++) {
		color_.push_back(color[i]);
	}
}

void trapezoid::render(const sdl::renderer& renderer) const {
	renderer.set_color(color_[0], color_[1], color_[2]);
	for(int32_t i = 0; i < 4; ++i){
		renderer.draw_line(vertices_[i].x, vertices_[i].y,
			vertices_[(i + 1) % 4].x, vertices_[(i + 1) % 4].y);
	}
}

void trapezoid::save(std::ostream& os) const {
	os << "trapezoid\n";
	for(int32_t i = 0; i < 4; ++i){
		os << vertices_[i].x << ' ' << vertices_[i].y << '\n';
	}
	os << color_[0] << ' ' << color_[1] << ' ' << color_[2] << '\n';
}

bool trapezoid::isPointInside(vertex v) const {
	int x = v.x;
	int y = v.y;
	int i1, i2, n, N, S, S1, S2, S3;
	bool flag;
	N = 4;
	for (n = 0; n < N; n++) {
		flag = false;
		i1 = n < N-1 ? n + 1 : 0;
		while (flag == false) {
		i2 = i1 + 1;
		if (i2 >= N)
			i2 = 0;
```

```
                if (i2 == (n < N-1 ? n + 1 : 0))
                        break;
                        S = abs (vertices_[i1].x * (vertices_[i2].y - vertices_[n ].y) +
vertices_[i2].x * (vertices_[n ].y - vertices_[i1].y) + vertices_[n].x  *
(vertices_[i1].y - vertices_[i2].y));
                S1 = abs (vertices_[i1].x * (vertices_[i2].y - y) + vertices_[i2].x * (y
- vertices_[i1].y) + x       * (vertices_[i1].y - vertices_[i2].y));
                S2 = abs (vertices_[n ].x * (vertices_[i2].y - y) + vertices_[i2].x * (y
- vertices_[n ].y) + x       * (vertices_[n ].y - vertices_[i2].y));
                S3 = abs (vertices_[i1].x * (vertices_[n ].y - y) + vertices_[n ].x * (y
- vertices_[i1].y) + x       * (vertices_[i1].y - vertices_[n ].y));
                        if (S == S1 + S2 + S3) {
                        flag = true;
                        break;
                }
                i1 = i1 + 1;
                if (i1 >= N)
                        i1 = 0;
                }
                if (flag == false)
                break;
        }
        return flag;
}


std::unique_ptr<figure> trapezoid_builder::add_vertex(const vertex& v) {
        vertices_[n_] = v;
        n_ += 1;
        if(n_ != 4){
                return nullptr;
        }
        return std::make_unique<trapezoid>(vertices_);
}

std::string trapezoid_builder::getType() {
        return "trapezoid";
}
```

Rhombus.h

```
#ifndef D_RHOMBUS_H
#define D_RHOMBUS_H 1

#include "figure.h"
#include "builder.h"
```

```
struct rhombus : figure {
	rhombus(const std::array<vertex, 4>& vertices);

	void setColor(std::vector<int> color) override;

	void render(const sdl::renderer& renderer) const override;

	void save(std::ostream& os) const override;


	bool isPointInside(vertex v) const override;

private:
	std::vector<int> color_;
	std::array<vertex, 4> vertices_;

};

struct rhombus_builder : builder {
	std::unique_ptr<figure> add_vertex(const vertex& v) override;

	std::string getType();
private:
	int32_t n_ = 0;
	std::array<vertex, 4> vertices_;

};


#endif // !D_RHOMBUS_H
```

Rhombus.cpp

```
#include "rhombus.h"

rhombus::rhombus(const std::array<vertex, 4>& vertices) : vertices_(vertices) {}

void rhombus::setColor(std::vector<int> color) {
	for (int i = 0; i < 3; i++) {
		color_.push_back(color[i]);
	}
}

void rhombus::render(const sdl::renderer& renderer) const {
	renderer.set_color(color_[0], color_[1], color_[2]);
	for (int32_t i = 0; i < 4; ++i) {
		renderer.draw_line(vertices_[i].x, vertices_[i].y,
```

```
                vertices_[(i + 1) % 4].x, vertices_[(i + 1) % 4].y);
        }
}

void rhombus::save(std::ostream& os) const {
        os << "rhombus\n";
        for (int32_t i = 0; i < 4; ++i) {
                os << vertices_[i].x << ' ' << vertices_[i].y << '\n';
        }
        os << color_[0] << ' ' << color_[1] << ' ' << color_[2] << '\n';
}


bool rhombus::isPointInside(vertex v) const {
        int x = v.x;
        int y = v.y;
        int i1, i2, n, N, S, S1, S2, S3;
        bool flag;
        N = 4;
        for (n = 0; n < N; n++) {
                flag = false;
                i1 = n < N-1 ? n + 1 : 0;
                while (flag == false) {
                i2 = i1 + 1;
                if (i2 >= N)
                        i2 = 0;
                if (i2 == (n < N-1 ? n + 1 : 0))
                        break;
                        S = abs (vertices_[i1].x * (vertices_[i2].y - vertices_[n ].y) +
vertices_[i2].x * (vertices_[n ].y - vertices_[i1].y) + vertices_[n].x  *
(vertices_[i1].y - vertices_[i2].y));
                S1 = abs (vertices_[i1].x * (vertices_[i2].y - y) + vertices_[i2].x * (y
- vertices_[i1].y) + x       * (vertices_[i1].y - vertices_[i2].y));
                S2 = abs (vertices_[n ].x * (vertices_[i2].y - y) + vertices_[i2].x * (y
- vertices_[n ].y) + x       * (vertices_[n ].y - vertices_[i2].y));
                S3 = abs (vertices_[i1].x * (vertices_[n ].y - y) + vertices_[n ].x * (y
- vertices_[i1].y) + x       * (vertices_[i1].y - vertices_[n ].y));
                        if (S == S1 + S2 + S3) {
                        flag = true;
                        break;
                }
                i1 = i1 + 1;
                if (i1 >= N)
                        i1 = 0;
                }
```

```
            if (flag == false)
            break;
        }
        return flag;
}


std::unique_ptr<figure> rhombus_builder::add_vertex(const vertex& v) {
        vertices_[n_] = v;
        n_ += 1;
        if(n_ != 4){
            return nullptr;
        }
        return std::make_unique<rhombus>(vertices_);
}

std::string rhombus_builder::getType() {
        return "rhombus";
}
```

Rectangle.h

```
#ifndef D_RECTANGLE_H
#define D_RECTANGLE_H 1

#include "figure.h"
#include "builder.h"



struct rectangle : figure {
        rectangle(const std::array<vertex, 4>& vertices);

        std::string getType();

        void setColor(std::vector<int> color) override;

        void render(const sdl::renderer& renderer) const override;

        void save(std::ostream& os) const override;


        bool isPointInside(vertex v) const override;

private:
        std::vector<int> color_;
        std::array<vertex, 4> vertices_;
```

```
};

struct rectangle_builder : builder {
	std::unique_ptr<figure> add_vertex(const vertex& v) override;

	std::string getType();

private:
	int32_t n_ = 0;
	std::array<vertex, 4> vertices_;

};



#endif // !D_RECTANGLE_H
```

Rectangle.cpp

```
#include "rectangle.h"

rectangle::rectangle(const std::array<vertex, 4>& vertices) : vertices_(vertices) {}

std::string getType() {
	return "rectangle";
}

void rectangle::setColor(std::vector<int> color) {
	for (int i = 0; i < 3; i++) {
		color_.push_back(color[i]);
	}
}

void rectangle::render(const sdl::renderer& renderer) const {
	renderer.set_color(color_[0], color_[1], color_[2]);
	for (int32_t i = 0; i < 4; ++i) {
		renderer.draw_line(vertices_[i].x, vertices_[i].y,
			vertices_[(i + 1) % 4].x, vertices_[(i + 1) % 4].y);
	}
}

void rectangle::save(std::ostream& os) const {
	os << "rectangle\n";
	for (int32_t i = 0; i < 4; ++i) {
		os << vertices_[i].x << ' ' << vertices_[i].y << '\n';
	}
```

```
        os << color_[0] << ' ' << color_[1] << ' ' << color_[2] << '\n';
}


bool rectangle::isPointInside(vertex v) const {
        int x = v.x;
        int y = v.y;
        int i1, i2, n, N, S, S1, S2, S3;
        bool flag;
        N = 4;
        for (n = 0; n < N; n++) {
                flag = false;
                i1 = n < N-1 ? n + 1 : 0;
                while (flag == false) {
                i2 = i1 + 1;
                if (i2 >= N)
                        i2 = 0;
                if (i2 == (n < N-1 ? n + 1 : 0))
                        break;
                        S = abs (vertices_[i1].x * (vertices_[i2].y - vertices_[n ].y) +
vertices_[i2].x * (vertices_[n ].y - vertices_[i1].y) + vertices_[n].x  *
(vertices_[i1].y - vertices_[i2].y));
                S1 = abs (vertices_[i1].x * (vertices_[i2].y - y) + vertices_[i2].x * (y
- vertices_[i1].y) + x       * (vertices_[i1].y - vertices_[i2].y));
                S2 = abs (vertices_[n ].x * (vertices_[i2].y - y) + vertices_[i2].x * (y
- vertices_[n ].y) + x       * (vertices_[n ].y - vertices_[i2].y));
                S3 = abs (vertices_[i1].x * (vertices_[n ].y - y) + vertices_[n ].x * (y
- vertices_[i1].y) + x       * (vertices_[i1].y - vertices_[n ].y));
                        if (S == S1 + S2 + S3) {
                        flag = true;
                        break;
                }
                i1 = i1 + 1;
                if (i1 >= N)
                        i1 = 0;
                }
                if (flag == false)
                break;
        }
        return flag;
}

std::unique_ptr<figure> rectangle_builder::add_vertex(const vertex& v) {
        vertices_[n_] = v;
        n_ += 1;
```

```
        if(n_ != 4){
                return nullptr;
        }
        return std::make_unique<rectangle>(vertices_);
}

std::string rectangle_builder::getType() {
        return "rectangle";
}
```

brokenLine.h

```
#ifndef D_BROKENLINE_H
#define D_BROKENLINE_H 1

#include "figure.h"
#include "builder.h"


struct brokenLine : figure {

        brokenLine(const std::vector<vertex>& vertices);

        void setColor(std::vector<int> color) override;

        void render(const sdl::renderer& renderer) const override;
        void save(std::ostream& os) const override;


        bool isPointInside(vertex v) const override;

private:
        std::vector<int> color_;
        std::vector<vertex> vertices_;


};

struct brokenLine_builder : builder {
        std::unique_ptr<figure> add_vertex(const vertex& v) override;

        std::string getType();
private:

        std::vector<vertex> vertices_;

};
```

```
#endif
```

brokenLine.cpp

```
#include "brokenLine.h"

brokenLine::brokenLine(const std::vector<vertex>& vertices) : vertices_(vertices)
{}

void brokenLine::setColor(std::vector<int> color) {
	for (int i = 0; i < 3; i++) {
		color_.push_back(color[i]);
	}
}

void brokenLine::render(const sdl::renderer& renderer) const {
	renderer.set_color(color_[0], color_[1], color_[2]);
	for (int32_t i = 0; i < vertices_.size() - 1; ++i) {
		renderer.draw_line(vertices_[i].x, vertices_[i].y,
			vertices_[(i + 1)].x, vertices_[(i + 1)].y);
	}
}

void brokenLine::save(std::ostream& os) const {
	os << "brokenLine\n";
	os << vertices_.size() << '\n';
	for (int32_t i = 0; i < vertices_.size(); ++i) {
		os << vertices_[i].x << ' ' << vertices_[i].y << '\n';
	}
	os << color_[0] << ' ' << color_[1] << ' ' << color_[2] << '\n';
}


bool brokenLine::isPointInside(vertex v) const {
	return false;
}

std::unique_ptr<figure> brokenLine_builder::add_vertex(const vertex& v) {
	if (v.x != -1 && v.y != -1) {
		vertices_.push_back(v);
		return nullptr;
	}

	return std::make_unique<brokenLine>(vertices_);
}
```

```
std::string brokenLine_builder::getType() {
	return "brokenLine";
}
```

Polygon.h

```
#ifndef D_POLYGON_H
#define D_POLYGON_H 1

#include "figure.h"
#include "builder.h"

struct polygon : figure {

	polygon(const std::vector<vertex>& vertices);

	std::string getType();

	void setColor(std::vector<int> color) override;

	void render(const sdl::renderer& renderer) const override;

	void save(std::ostream& os) const override;

	bool isPointInside(vertex v) const override;

private:
	std::vector<int> color_;
	std::vector<vertex> vertices_;

};

struct polygon_builder : builder {
	std::unique_ptr<figure> add_vertex(const vertex& v) override;
	std::string getType();

private:

	std::vector<vertex> vertices_;

};


#endif
```

Polygon.cpp

```
#include "polygon.h"
```

```
polygon::polygon(const std::vector<vertex>& vertices) : vertices_(vertices) {}

std::string polygon::getType() {
	return "polygon";
}

void polygon::setColor(std::vector<int> color) {
	for (int i = 0; i < 3; i++) {
		color_.push_back(color[i]);
	}
}

void polygon::render(const sdl::renderer& renderer) const {
	renderer.set_color(color_[0], color_[1], color_[2]);
	for (int32_t i = 0; i < vertices_.size() - 1; ++i) {
		renderer.draw_line(vertices_[i].x, vertices_[i].y,
			vertices_[(i + 1)].x, vertices_[(i + 1)].y);
	}
	renderer.draw_line(vertices_[vertices_.size() - 1].x, vertices_[vertices_.size()
- 1].y, vertices_[0].x, vertices_[0].y);
}

void polygon::save(std::ostream& os) const {
	os << "polygon\n";
	os << vertices_.size() << '\n';
	for (int32_t i = 0; i < vertices_.size(); ++i) {
		os << vertices_[i].x << ' ' << vertices_[i].y << '\n';
	}
	os << color_[0] << ' ' << color_[1] << ' ' << color_[2] << '\n';
}


bool polygon::isPointInside(vertex v) const {
	int x = v.x;
	int y = v.y;
	int i1, i2, n, N, S, S1, S2, S3;
	bool flag;
	N = vertices_.size();
	for (n = 0; n < N; n++) {
		flag = false;
		i1 = n < N-1 ? n + 1 : 0;
		while (flag == false) {
		i2 = i1 + 1;
		if (i2 >= N)
			i2 = 0;
```

```
                if (i2 == (n < N-1 ? n + 1 : 0))
                        break;
                        S = abs (vertices_[i1].x * (vertices_[i2].y - vertices_[n ].y) +
vertices_[i2].x * (vertices_[n ].y - vertices_[i1].y) + vertices_[n].x  *
(vertices_[i1].y - vertices_[i2].y));
                S1 = abs (vertices_[i1].x * (vertices_[i2].y - y) + vertices_[i2].x * (y
- vertices_[i1].y) + x       * (vertices_[i1].y - vertices_[i2].y));
                S2 = abs (vertices_[n ].x * (vertices_[i2].y - y) + vertices_[i2].x * (y
- vertices_[n ].y) + x       * (vertices_[n ].y - vertices_[i2].y));
                S3 = abs (vertices_[i1].x * (vertices_[n ].y - y) + vertices_[n ].x * (y
- vertices_[i1].y) + x       * (vertices_[i1].y - vertices_[n ].y));
                        if (S == S1 + S2 + S3) {
                        flag = true;
                        break;
                }
                i1 = i1 + 1;
                if (i1 >= N)
                        i1 = 0;
                }
                if (flag == false)
                break;
        }
        return flag;
}


std::unique_ptr<figure> polygon_builder::add_vertex(const vertex& v) {
        if (v.x != -1 && v.y != -1) {
                vertices_.push_back(v);
                return nullptr;
        }

        return std::make_unique<polygon>(vertices_);
}

std::string polygon_builder::getType() {
        return "polygon";
}
```

Circle.h

```
#ifndef D_CIRCLE_H
#define D_CIRCLE_H 1

#include "figure.h"
#include "builder.h"
#include <math.h>
```

```
struct circle : figure {
	circle(const std::vector<vertex>& vertices);

	void setColor(std::vector<int> color) override;

	void render(const sdl::renderer& renderer) const override;

	void save(std::ostream& os) const override;

	bool isPointInside(vertex v) const override;

private:
	std::vector<int> color_;
	std::vector<vertex> vertices_;
	int radius;
};

struct circle_builder : builder {

	std::unique_ptr<figure> add_vertex(const vertex& v) override;

	std::string getType() override;

private:

	int32_t n_ = 0;

	std::vector<vertex> vertices_;

};


#endif
```

Circle.cpp

```
#include "circle.h"

circle::circle(const std::vector<vertex>& vertices) : vertices_(vertices) {
	radius = (int) sqrt((vertices_[1].y - vertices_[0].y) * (vertices_[1].y -
vertices_[0].y) + (vertices_[1].x - vertices_[0].x) * (vertices_[1].x -
vertices_[0].x));
}

void circle::setColor(std::vector<int> color) {
	for (int i = 0; i < 3; i++) {
```

```
            color_.push_back(color[i]);
        }
}

void circle::render(const sdl::renderer& renderer) const {
        renderer.set_color(color_[0], color_[1], color_[2]);



            /*

            *         *
              ________
                       |
                       *

            */

        int32_t centreX = vertices_[0].x;
        int32_t centreY = vertices_[0].y;

        const int32_t diameter = (radius * 2);

        int32_t x = (radius - 1);
        int32_t y = 0;
        int32_t tx = 1;
        int32_t ty = 1;
        int32_t error = (tx - diameter);

        while (x >= y)
        {
            //  Each of the following renders an octant of the circle
        renderer.draw_point(centreX + x, centreY - y);
        renderer.draw_point(centreX + x, centreY + y);
        renderer.draw_point(centreX - x, centreY - y);
        renderer.draw_point(centreX - x, centreY + y);
        renderer.draw_point(centreX + y, centreY - x);
        renderer.draw_point(centreX + y, centreY + x);
        renderer.draw_point(centreX - y, centreY - x);
        renderer.draw_point(centreX - y, centreY + x);

        if (error <= 0)
        {
        ++y;
        error += ty;
        ty += 2;
```

```
        }

        if (error > 0)
        {
        --x;
        tx += 2;
        error += (tx - diameter);
        }
        }
        }
}

void circle::save(std::ostream& os) const {
        os << "circle\n";

        os << vertices_[0].x << ' ' << vertices_[0].y << '\n';
        os << vertices_[1].x << ' ' << vertices_[1].y << '\n';

        os << color_[0] << ' ' << color_[1] << ' ' << color_[2] << '\n';
}


        /*          *
            /
                /*
                        /_|


        */

bool circle::isPointInside(vertex v) const {
        int distance = (int) sqrt((v.y - vertices_[0].y) * (v.y - vertices_[0].y) + (v.x -
vertices_[0].x) * (v.x - vertices_[0].x));
        if (distance <= radius)
                return true;
        return false;
}


std::unique_ptr<figure> circle_builder::add_vertex(const vertex& v) {
        vertices_.push_back(v);
        n_ += 1;
        if (n_ != 2) {
                return nullptr;
        }
        return std::make_unique<circle>(vertices_);
```

```
}

std::string circle_builder::getType() {
	return "circle";
}
```

Figure.h

```
#ifndef D_FIGURE_H
#define D_FIGURE_H 1

#include "vertex.h"

#include "sdl.h"
#include <array>
#include <vector>
#include <memory>
//#include <iostream>
//#include <fstream>
#include <string>


struct figure {
	virtual void render(const sdl::renderer& renderer) const = 0;
	virtual void save(std::ostream& os) const = 0;
	virtual bool isPointInside(vertex v) const = 0;
	virtual void setColor(std::vector<int> color) = 0;
	virtual ~figure() = default;

};

#endif // !D_FIGURE_H
```

Sdl.h

```
#ifndef D_SDL_H_
#define D_SDL_H_ 1

#include <string>

#include "SDL_events.h"
#include "SDL_render.h"
#include "SDL_video.h"

namespace sdl {

struct sdl {
  sdl();
```

```
  ~sdl();

};

struct renderer {
  renderer(const std::string& window_name);
  ~renderer();

  // set color for subsequent operations
  void set_color(uint8_t r, uint8_t g, uint8_t b) const;
  // fill screen with current color
  void clear() const;
  // draw segment with current color
  void draw_line(int32_t x1, int32_t y1, int32_t x2, int32_t y2) const;
  void draw_point(int32_t x, int32_t y) const;

  // every command draws to a temporary buffer
  // this function swaps temporary buffer containing new frame with current frame
  void present() const;

private:
  sdl system;
  SDL_Window* window_;
  SDL_Renderer* renderer_;

};

struct quit_event {
  quit_event() = default;
  quit_event(const SDL_QuitEvent& e);

private:
  SDL_QuitEvent event_;

};

struct mouse_button_event {
  mouse_button_event() = default;
  mouse_button_event(const SDL_MouseButtonEvent& e);

  static constexpr uint32_t down = SDL_MOUSEBUTTONDOWN;
  static constexpr uint32_t up = SDL_MOUSEBUTTONUP;

  static constexpr uint8_t left = SDL_BUTTON_LEFT;
  static constexpr uint8_t right = SDL_BUTTON_RIGHT;
```

```
  // button up or down
  uint32_t type() const;
  // left or right button
  uint8_t button() const;
  // distance from left border in pixels
  int32_t x() const;
  // distance from top border in pixels
  int32_t y() const;

private:
  SDL_MouseButtonEvent event_;

};

struct event {
  // try to convert generic event to some specific event
  bool extract(quit_event& event) const;
  bool extract(mouse_button_event& event) const;

  // try to get next event
  static bool poll(event& e);

private:
  SDL_Event event_;

};

} // namespace sdl

#endif // D_SDL_H_
```

Sdl.cpp

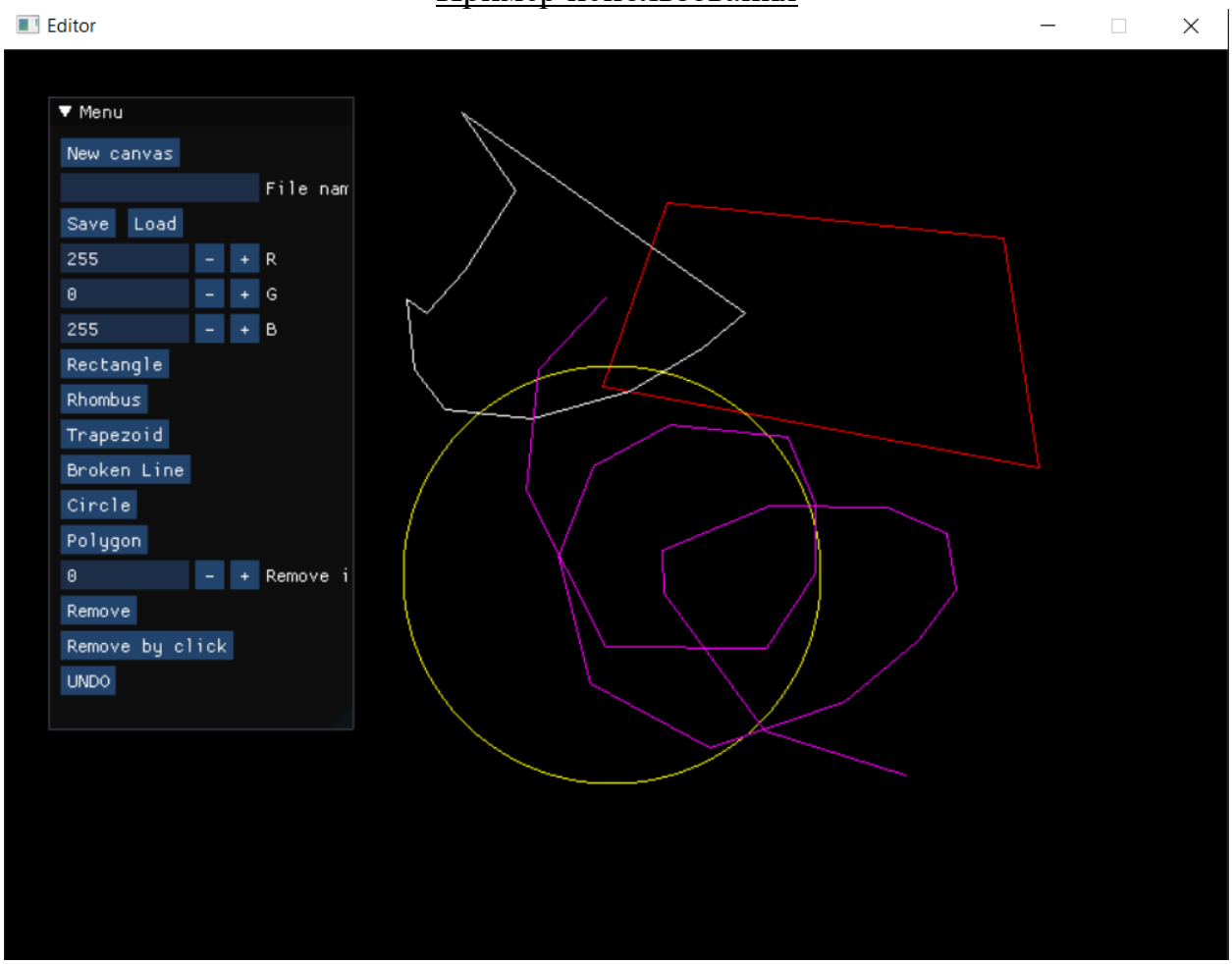
```
#include "sdl.h"

#include <SDL.h>

#include "imgui.h"
#include "imgui_sdl.h"
#include "imgui_impl_sdl.h"

namespace sdl {

sdl::sdl() {
  SDL_Init(SDL_INIT_VIDEO);
}
```

```
sdl::~sdl() {
  SDL_Quit();
}

renderer::renderer(const std::string& window_name):
    window_(SDL_CreateWindow(window_name.data(),
SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED,
        800, 600, 0)),
    renderer_(SDL_CreateRenderer(window_, -1,
SDL_RENDERER_SOFTWARE)) {
  ImGui::CreateContext();
  ImGui_ImplSDL2_Init(window_);
  ImGuiSDL::Initialize(renderer_, 800, 600);
  ImGui_ImplSDL2_NewFrame(window_);
  ImGui::NewFrame();
}

renderer::~renderer() {
  ImGuiSDL::Deinitialize();
  ImGui::DestroyContext();
  SDL_DestroyRenderer(renderer_);
  SDL_DestroyWindow(window_);
}

void renderer::set_color(uint8_t r, uint8_t g, uint8_t b) const {
  SDL_SetRenderDrawColor(renderer_, r, g, b, 255);
}

void renderer::clear() const {
  SDL_RenderClear(renderer_);
}

void renderer::draw_line(int32_t x1, int32_t y1, int32_t x2, int32_t y2) const {
  SDL_RenderDrawLine(renderer_, x1, y1, x2, y2);
}

void renderer::draw_point(int32_t x, int32_t y) const {
  SDL_RenderDrawPoint(renderer_, x, y);
}

void renderer::present() const {
  ImGui::Render();
  ImGuiSDL::Render(ImGui::GetDrawData());
  SDL_RenderPresent(renderer_);
```

```
  ImGui_ImplSDL2_NewFrame(window_);
  ImGui::NewFrame();
}

quit_event::quit_event(const SDL_QuitEvent& e): event_(e) {}

mouse_button_event::mouse_button_event(const SDL_MouseButtonEvent& e):
event_(e) {}

uint32_t mouse_button_event::type() const {
  return event_.type;
}

uint8_t mouse_button_event::button() const {
  return event_.button;
}

int32_t mouse_button_event::x() const {
  return event_.x;
}

int32_t mouse_button_event::y() const {
  return event_.y;
}

bool event::extract(quit_event& event) const {
  if(event_.type == SDL_QUIT){
    event = event_.quit;
    return true;
  }
  return false;
}

bool event::extract(mouse_button_event& event) const {
  if(event_.type == SDL_MOUSEBUTTONDOWN || event_.type ==
SDL_MOUSEBUTTONUP){
    event = event_.button;
    return true;
  }
  return false;
}

bool event::poll(event& e) {
  bool result = SDL_PollEvent(&e.event_);
  if(result){
```

```
    ImGui_ImplSDL2_ProcessEvent(&e.event_);
  }
  return result;
}

} // namespace sdl
```

Пример использования



Объяснение результатов

Программа представляет собой визуальное приложение, способное строить прямоугольник, ромб, трапецию, ломаную линию, многоугольник и круг. Возможно удаление по индексу, клику, операция undo, загрузка и сохранение в файл.

Вывод

Я познакомился с визуальными библиотеками в C++, углубил свои знания в области полиморфизма, познакомился с написанием и сборкой объемных проектов, а так же подключением сторонних библиотек.