

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ
ФЕДЕРАЦИИ МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

ЛАБОРАТОРНАЯ РАБОТА №7
по курсу объектно-ориентированное программирование I семестр, 2019/20
уч. год

Студент Попов Данила Андреевич, группа М8О-208Б-18

Преподаватель Журавлёв Андрей Андреевич

Условие

Спроектировать графический редактор с графическим интерфейсом.

Редактор должен соответствовать следующему функционалу:

1. создание нового документа
2. импорт документа из файла
3. экспорт документа в файл
4. создание/удаление графических примитивов
5. отображение документа на экране
6. операция undo

Описание программы

Исходный код лежит в 8 файлах:

1. app/main.cpp: точка входа в программу
2. src/application.cpp: определение центрального класса
3. src/application.hpp: объявление центрального класса
4. src/builders.hpp: обработчики пользовательского ввода
5. src/figures.hpp: определение графических примитивов
6. src/editor/brush.hpp: определение кисти
7. src/editor/builder.hpp: определение интерфейса обработчика пользовательского ввода
8. src/editor/drawable.hpp: определение интерфейса графического примитива
9. src/editor/figure.hpp: определение интерфейса графической фигуры
10. src/editor/storage.hpp: хранилище текущего состояния канваса
11. src/geom/algorithm.hpp
12. src/geom/point.hpp
13. src/geom/polygon.hpp
14. src/system/application_base.cpp: определение базового класса приложения

15. `src/system/application_base.hpp`: объявление базового класса приложения
16. `src/system/renderer.hpp`: класс отрисовщика
17. `src/system/sdl2.cpp`: sdl2 C++ wrapper
18. `src/system/sdl2.hpp`

Дневник отладки

Не смог осилить выпадающие окошки. Сохранение и открытие документов происходит через командную строку.

Недочёты

Сериализатор до жути тривиальный.

Выводы

ImGui как панацея от всех болезней. Открыл для себя паттерн команд, которые можно отменять. Так же можно было бы сделать Redo, добавив пару индексов. Познакомился с миром git submodule при попытках подключить SDL2 и ImGui.

Исходный код

main.cpp

```
#include "application.hpp"

oop::application g_application;

int main(const int argc, char* argv[]) {
    return g_application.start(argc, argv);
}
```

src/application.cpp

```
#include "application.hpp"

#include "imgui.h"

#include <iostream>
#include <vector>

#include "builders.hpp"

using namespace oop;

static void fill_with_style_color(SDL_Renderer* renderer);

struct command_add final : editor::i_command
{
    command_add(const editor::fig_ptr& fig)
        : fig_{ fig }
    {}

    bool commit(std::vector<editor::fig_ptr>& figs) override
    {
        figs.push_back(fig_);
        return true;
    }

    void reset(std::vector<editor::fig_ptr>& figs) override
    {
        figs.erase(figs.end() - 1);
    }

private:
    editor::fig_ptr fig_;
};

struct command_remove final : editor::i_command
{
    command_remove(const editor::vec2& p)
        : p{ p }
        , ix{ 0 }
    {}
}
```

```

bool commit(std::vector<editor::fig_ptr>& figs) override
{
    const size_t size = figs.size();
    for (size_t i = 1; i <= size; ++i)
    {
        if (figs[size - i]->inside(p))
        {
            fig_ = figs[size - i];
            ix = figs.erase(figs.begin() + (size - i)) - figs.begin();
            return true;
        }
    }
    return false;
}

void reset(std::vector<editor::fig_ptr>& figs) override
{
    figs.insert(figs.begin() + ix, fig_);
}

private:
    editor::vec2 p;
    editor::fig_ptr fig_;
    size_t ix;
};

application::application()
    : builder_{new idle_builder{}}
{
    ImGui::StyleColorsLight();
}

void application::process_event(const SDL_Event& event)
{
    static int i = 0;
    ImGuiIO& io = ImGui::GetIO();
    if (io.WantCaptureMouse && (event.type == SDL_MOUSEBUTTONDOWN || event.type == SDL_MOUSEBUTTONUP || io.WantCaptureKeyboard))
    {
        return;
    }

    if (typeid(*builder_) == typeid(idle_builder))

```

```

{
    if (event.type == SDL_MOUSEBUTTONDOWN)
    {
        auto& button = event.button;
        editor::cmd_ptr cmd{ new command_remove({ button.x, button.y }) };
        storage_.commit(cmd);
    }
}
else if (builder_->next(event.button))
{
    const editor::fig_ptr fig = builder_->extract();
    fig->color = brush_;

    const editor::cmd_ptr cmd(new command_add{ fig });
    storage_.commit(cmd);

    builder_.reset(new idle_builder{});
}

if (event.type == SDL_KEYDOWN)
{
    if (event.key.keysym.scancode == SDL_SCANCODE_U)
    {
        storage_.undo();
    }
}
}

void application::construct_frame()
{
    fill_with_style_color(renderer_);
    storage_.draw(renderer_);

    auto [r, g, b] = brush_.convert_u8();
    SDL_SetRenderDrawColor(renderer_, r, g, b, SDL_ALPHA_OPAQUE);
    builder_->draw(renderer_);

    construct_toolbar();
}

void application::construct_toolbar()
{

```

```

auto const window_flags = 0
    | ImGuiWindowFlags_NoTitleBar
    | ImGuiWindowFlags_NoScrollbar
    | ImGuiWindowFlags_MenuBar
    | ImGuiWindowFlags_NoMove
    | ImGuiWindowFlags_NoResize
    | ImGuiWindowFlags_NoCollapse
    | ImGuiWindowFlags_NoNav
    | ImGuiWindowFlags_NoBackground
    | ImGuiWindowFlags_NoBringToFrontOnFocus;

int w, h;
SDL_GetWindowSize(window_, &w, &h);
ImGui::SetNextWindowSize({ float(w), 0 });
ImGui::SetNextWindowPos({ 0, 0 });

if (!ImGui::Begin("toolbar", nullptr, window_flags))
{
    ImGui::End();
    return;
}

if (ImGui::BeginMenuBar())
{
    if (ImGui::BeginMenu("file"))
    {
        if (ImGui::MenuItem("open"))        open();
        if (ImGui::MenuItem("save"))        save();
        if (ImGui::MenuItem("save as ...")) save_as();
        ImGui::EndMenu();
    }

    if (ImGui::BeginMenu("figure"))
    {
        if (ImGui::MenuItem("tetragon")) builder_.reset(new polygon_builder<4>);
        if (ImGui::MenuItem("pentagon")) builder_.reset(new polygon_builder<5>);
        if (ImGui::MenuItem("hexagon"))  builder_.reset(new polygon_builder<6>);
        if (ImGui::MenuItem("shape"))    builder_.reset(new shape_builder{});
        if (ImGui::MenuItem("circle"))   builder_.reset(new circle_builder{});
        ImGui::EndMenu();
    }
}

```



```

        ImGui::SameLine();
        ImGui::ColorEdit3("", brush_.rgb, ImGuiColorEditFlags_NoInputs);

        ImGui::EndMenuBar();
    }

    ImGui::End();
}

void application::open()
{
    std::getline(std::cin, filename_);
    storage_.clear();

    std::fstream f;
    f.open(filename_, std::ios_base::in);
    while (f)
    {
        editor::fig_ptr fig;

        std::string header;
        f >> header;
        if (header == shape::header)
        {
            fig.reset(new shape{});
        }
        else if (header == circle::header)
        {
            fig.reset(new circle{});
        }
        else {
            break;
        }

        fig->deserialize(f);
        storage_.push_back(fig);
    }
}

void application::save()
{
    if (filename_.empty())

```

```

    {
        save_as();
        return;
    }

    storage_.save(filename_);
}

void application::save_as()
{
    std::getline(std::cin, filename_);
    storage_.save(filename_);
}

void fill_with_style_color(SDL_Renderer* renderer)
{
    const auto& c = ImGui::GetStyle().Colors[ImGuiCol_WindowBg];
    SDL_SetRenderDrawColor(renderer, 255 * c.x, 255 * c.y, 255 * c.z, 255 * c.w);
    SDL_RenderClear(renderer);
}

```

src/application.hpp

```
#pragma once

#include "system/application_base.hpp"

#include <string>
#include <vector>
#include <memory>

#include "editor/brush.hpp"
#include "editor/builder.hpp"
#include "editor/storage.hpp"

namespace oop
{
    class application final : public system::application_base
    {
    public:
        application();

    private:
        void process_event(const SDL_Event& event) override;
        void construct_frame() override;

        /*!
        * @brief constructs toolbar
        */
        void construct_toolbar();

        // Toolbar file actions
        void open();
        void save();
        void save_as();

        editor::brush brush_;
        std::string filename_ = "";
        std::unique_ptr<editor::i_builder> builder_;
        editor::storage storage_;
    };
}
```

src/builders.hpp

```
#pragma once

#include <optional>
#include <cmath>

#include "editor/builder.hpp"
#include "editor/figure.hpp"
#include "system/sdl2.hpp"
#include "figures.hpp"

namespace oop
{
    struct idle_builder final
        : editor::i_builder
    {
        void draw(system::renderer&) override
        {
            // Placeholder
        }

        bool next(const SDL_MouseButtonEvent&) override
        {
            return false;
        }

        editor::fig_ptr extract() override
        {
            return nullptr;
        }
    };

    struct shape_builder final
        : editor::i_builder
    {
        void draw(system::renderer& renderer) override
        {
            if (vertices_.empty())
            {
                return;
            }
        }
    }
}
```

```

    const size_t size = vertices_.size();
    for (size_t i = 0; i < size - 1; i++)
    {
        const auto& v1 = vertices_[i], v2 = vertices_[i + 1];
        SDL_RenderDrawLine(renderer, v1.x, v1.y, v2.x, v2.y);
    }

    auto const pos = ImGui::GetMousePos();
    auto const last = vertices_[size - 1];
    SDL_RenderDrawLine(renderer, last.x, last.y, int(pos.x), int(pos.y));
}

bool next(const SDL_MouseButtonEvent& event) override
{
    if (event.type == SDL_MOUSEBUTTONDOWN)
    {
        vertices_.emplace_back(event.x, event.y);
        if (event.button == SDL_BUTTON_RIGHT) {
            return true;
        }
    }
    return false;
}

editor::fig_ptr extract() override final
{
    return editor::fig_ptr{new shape(std::move(vertices_))};
}

private:
    std::vector<editor::vec2> vertices_;
};

struct circle_builder final
    : editor::i_builder
{
    void draw(system::renderer& renderer) override
    {
        if (center_.has_value())
        {
            auto const & center = *center_;
            auto const pos      = ImGui::GetMousePos();

```

```

        auto const x      = pos.x - center.x;
        auto const y      = pos.y - center.y;
        auto const rad     = sqrt(x * x + y * y);
        draw_circle(renderer, int(center.x), int(center.y), int(rad));
    }
}

bool next(const SDL_MouseButtonEvent& event) override
{
    if (event.type == SDL_MOUSEBUTTONDOWN && event.button == SDL_BUTTON_LEFT)
    {
        if (center_.has_value())
        {
            auto const & center = *center_;
            auto const pos      = ImGui::GetMousePos();
            auto const x        = pos.x - center.x;
            auto const y        = pos.y - center.y;

            radius_ = sqrt(x * x + y * y);
            return true;
        }
        center_.emplace(event.x, event.y);
    }
    return false;
}

editor::fig_ptr extract() override
{
    return editor::fig_ptr{ new circle(center_.value(), radius_) };
}

private:
    float radius_ = 0;
    std::optional<editor::vec2> center_;
};

template<size_t N>
struct polygon_builder final : editor::i_builder {
    void draw(system::renderer& renderer) override
    {
        if (vertices_.empty())
        {

```

```

        return;
    }

    const size_t size = vertices_.size();
    for (size_t i = 0; i < size - 1; i++)
    {
        const auto& v1 = vertices_[i], v2 = vertices_[i + 1];
        SDL_RenderDrawLine(renderer, v1.x, v1.y, v2.x, v2.y);
    }

    auto const pos = ImGui::GetMousePos();
    auto const last = vertices_[size - 1];
    SDL_RenderDrawLine(renderer, last.x, last.y, int(pos.x), int(pos.y));
}

bool next(const SDL_MouseButtonEvent& event) override
{
    if (event.type == SDL_MOUSEBUTTONDOWN)
    {
        vertices_.emplace_back(event.x, event.y);
        if (vertices_.size() == N) {
            return true;
        }
    }
    return false;
}

editor::fig_ptr extract() override
{
    return editor::fig_ptr{ new shape(std::move(vertices_)) };
}

private:
    std::vector<editor::vec2> vertices_;
};
}

```

src/figures.hpp

```
#pragma once
```

```
#include <vector>
```

```
#include "geom/polygon.hpp"
```

```
#include "editor/figure.hpp"
```

```
#include "editor/brush.hpp"
```

```
#include "system/sdl2.hpp"
```

```
#include "geom/algorithm.hpp"
```

```
namespace oop
```

```
{
```

```
    template<size_t N>
```

```
    using polygon_vec2 = basic_polygon<editor::vec2, N>;
```

```
    struct shape final : editor::i_figure
```

```
    {
```

```
        static auto constexpr header = "SHAPE";
```

```
        shape() = default;
```

```
        explicit shape(std::vector<editor::vec2>&& storage)
```

```
            : points_(std::move(storage))
```

```
        {}
```

```
        bool inside(const editor::vec2& v)
```

```
        {
```

```
            return is_inside(points_.data(), points_.size(), v);
```

```
        }
```

```
        void serialize(std::ostream& file) override
```

```
        {
```

```
            auto [r, g, b] = color.convert_u8();
```

```
            file << header << " " << int(r) << " " << int(g) << " " << int(b) << " " <<
```

```
            for (auto& p : points_)
```

```
            {
```

```
                file << " " << p.x << " " << p.y;
```

```
            }
```

```
        }
```



```

void deserialize(std::istream& file) override
{
    points_.clear();

    file >> color;

    size_t size;
    file >> size;
    for (size_t i = 0; i < size; i++)
    {
        editor::vec2 v;
        file >> v.x >> v.y;
        points_.push_back(v);
    }
}

private:
void ondraw(system::renderer& renderer)
{
    size_t size = points_.size();
    for (size_t i = 0; i < size; ++i) {
        auto& u = points_[i];
        auto& v = points_[(i + 1) % size];

        SDL_RenderDrawLine(renderer, u.x, u.y, v.x, v.y);
    }
}

std::vector<editor::vec2> points_;
};

struct circle final : editor::i_figure
{
    static auto constexpr header = "CIRCLE";

    circle() = default;

    explicit circle(const editor::vec2& pos, int radius)
        : pos_(pos)
        , rad_(radius)
    {}
}

```

```

bool inside(const editor::vec2& p) override
{
    int x = pos_.x - p.x;
    int y = pos_.y - p.y;

    return sqrt(x * x + y * y) <= rad_;
}

void serialize(std::ostream& file) override
{
    auto [r, g, b] = color.convert_u8();
    file << header << " " << int(r) << " " << int(g) << " " << int(b) << " " <<
}

void deserialize(std::istream& file) override
{
    file >> color >> pos_.x >> pos_.y >> rad_;
}

private:
void ondraw(system::renderer& renderer)
{
    system::draw_circle(renderer, pos_.x, pos_.y, rad_);
}

editor::vec2 pos_;
int rad_;
};
}

```

src/editor/brush.hpp

```
#pragma once

#include <tuple>
#include <istream>

namespace oop::editor
{
    struct brush
    {
        union
        {
            float rgb[3];
            struct
            {
                float r, g, b;
            };
        };

        std::tuple<Uint8, Uint8, Uint8> convert_u8() const noexcept
        {
            return {r * 255, g * 255, b * 255};
        }
    };
}

inline std::istream& operator>>(std::istream& s, oop::editor::brush& brush)
{
    int r, g, b;
    s >> r >> g >> b;

    brush = oop::editor::brush{r / 255.0f, g / 255.0f, b / 255.0f };

    return s;
}
```

src/editor/builder.hpp

```
#pragma once

#include "editor/drawable.hpp"
#include "editor/figure.hpp"
#include "editor/storage.hpp"

namespace oop::editor
{
    struct i_builder: i_drawable
    {
        i_builder() = default;
        i_builder(const i_builder&) = default;
        i_builder(i_builder&&) noexcept = default;
        i_builder& operator=(const i_builder&) = default;
        i_builder& operator=(i_builder&&) noexcept = default;
        ~i_builder() = 0;

        /*!
         * @brief
         * Builds figure click-by-click
         *
         * @param event
         * Mouse event on editor
         *
         * @return true if figure complete
         */
        virtual bool next(const SDL_MouseButtonEvent& event) = 0;

        virtual fig_ptr extract() = 0;
    };

    inline i_builder::~i_builder() = default;
}
```

src/editor/drawable.hpp

```
#pragma once

#include "system/renderer.hpp"

namespace oop::editor
{
    struct i_drawable
    {
        i_drawable() = default;
        i_drawable(const i_drawable&) = default;
        i_drawable(i_drawable&&) noexcept = default;
        i_drawable& operator=(const i_drawable&) = default;
        i_drawable& operator=(i_drawable&&) noexcept = default;
        virtual ~i_drawable() = 0;

        virtual void draw(system::renderer& renderer) = 0;
    };

    inline i_drawable::~i_drawable() = default;
}
```

src/editor/figure.hpp

```
#pragma once

#include <cstdint>
#include <iostream>

#include "system/renderer.hpp"
#include "editor/drawable.hpp"
#include "editor/brush.hpp"

namespace oop::editor
{
    struct vec2
    {
        vec2() = default;
        vec2(const int32_t x, const int32_t y)
            : x{x}
            , y{y}
        {}

        union
        {
            int32_t points[2];
            struct {
                int32_t x;
                int32_t y;
            };
        };
    };

    struct i_figure : i_drawable
    {
        i_figure() = default;
        i_figure(const i_figure&) = default;
        i_figure(i_figure&&) noexcept = default;
        i_figure& operator=(const i_figure&) = default;
        i_figure& operator=(i_figure&&) noexcept = default;
        virtual ~i_figure() = 0;

        void draw(system::renderer& renderer) override final
        {
            auto [r, g, b] = color.convert_u8();
        }
    };
}
```

```

        SDL_SetRenderDrawColor(renderer, r, g, b, SDL_ALPHA_OPAQUE);
        ondraw(renderer);
    }

    virtual bool inside(const editor::vec2&)
    {
        return false;
    }

    virtual void serialize(std::ostream& file) = 0;

    virtual void deserialize(std::istream& file) = 0;

    brush color = { 0, 0, 0 };

private:
    virtual void ondraw(system::renderer& renderer) = 0;
};

inline i_figure::~i_figure() = default;
}

```

src/editor/storage.hpp

```
#pragma once

#include <memory>
#include <vector>
#include <algorithm>
#include <stdexcept>
#include <fstream>
#include <vector>

#include "editor/drawable.hpp"
#include "editor/figure.hpp"

namespace oop::editor
{
    using fig_ptr = std::shared_ptr<i_figure>;

    struct i_command
    {
        virtual ~i_command() = 0;

        virtual bool commit(std::vector<fig_ptr>& figs) = 0;
        virtual void reset(std::vector<fig_ptr>& figs) = 0;
    };

    inline i_command::~i_command() = default;

    typedef std::shared_ptr<i_command> cmd_ptr;

    struct storage final : i_drawable
    {
        void draw(system::renderer& renderer) override
        {
            for (auto fig : *this)
            {
                fig->draw(renderer);
            }
        }

        void commit(const cmd_ptr& cmd)
        {
            if (cmd->commit(figures_))
            {
            }
        }
    };
}
```



```

        {
            commands_.push_back(cmd);
        }
    }

    void undo()
    {
        if (commands_.empty())
        {
            return;
        }

        const auto last = commands_.end() - 1;
        (*last)->reset(figures_);
        commands_.erase(last);
    }

    void clear()
    {
        commands_.clear();
        figures_.clear();
    }

    void push_back(fig_ptr fig)
    {
        figures_.push_back(fig);
    }

    void save(std::string_view filename)
    {
        std::fstream f;
        f.open(filename.data(), std::ios_base::out);

        for (auto& fig : *this)
        {
            fig->serialize(f);
            f << std::endl;
        }
    }

    using const_iterator = std::vector<fig_ptr>::const_iterator;

```

```

    const_iterator begin() const
    {
        return figures_.begin();
    }

    const_iterator end() const
    {
        return figures_.end();
    }

    /*!
     * @brief undo last change in associated storage
     */
    void undo(std::vector<fig_ptr>& storage);
private:
    std::vector<cmd_ptr> commands_;
    std::vector<fig_ptr> figures_;
};
}

```

src/geom/algorithm.hpp

```
#pragma once
```

```
#include <type_traits>
```

```
#include <tuple>
```

```
#include <utility>
```

```
#include <ostream>
```

```
#include <cmath>
```

```
#include "point.hpp"
```

```
#include "../editor/figure.hpp"
```

```
namespace detail {
```

```
    template<size_t _Off, size_t ... _Ix>
```

```
    std::index_sequence<(_Off + _Ix)...> add_offset(std::index_sequence<_Ix...>) {  
        return {};
```

```
    }
```

```
    template<size_t _Off, size_t _N>
```

```
    auto make_index_sequence_with_offset() {  
        return add_offset<_Off>(std::make_index_sequence<_N>{});  
    }
```

```
    template<typename _T, size_t... _Ix>
```

```
    double area2d(const _T& tuple, std::index_sequence<_Ix...>) {  
        using vertex = std::remove_const_t<std::remove_reference_t<decltype(std::get<0>(tuple))>>;  
        static_assert(std::is_same_v<vertex, point2d>, "incorrect type");
```

```
        auto constexpr tuple_size = std::tuple_size<_T>{}();
```

```
        auto constexpr x = 0;
```

```
        auto constexpr y = 1;
```

```
        using std::get;
```

```
        double result = ((get<_Ix>(tuple)[x] * (get<_Ix + 1>(tuple)[y] - get<_Ix - 1>(tuple)[y]) -  
            (get<_Ix>(tuple)[y] * (get<_Ix + 1>(tuple)[x] - get<_Ix - 1>(tuple)[x])) / 2;  
        auto constexpr first = 0;  
        auto constexpr last = tuple_size - 1;  
        result += get<first>(tuple)[x] * (get<first + 1>(tuple)[y] - get<last>(tuple)[y]) -  
            (get<first>(tuple)[y] * (get<first + 1>(tuple)[x] - get<last>(tuple)[x])) / 2;  
        result += get<last>(tuple)[x] * (get<first>(tuple)[y] - get<last - 1>(tuple)[y]) -  
            (get<last>(tuple)[y] * (get<first>(tuple)[x] - get<last - 1>(tuple)[x])) / 2;
```

```
        return std::abs(result);
```

```

}

template<typename _T, std::size_t... _Ix>
auto center2d(const _T& tuple, std::index_sequence<_Ix...>) {
    using vertex = std::remove_const_t<std::remove_reference_t<decltype(std::get<0>(
        static_assert(std::is_same_v<vertex, point2d>, "incorrect type");

    auto constexpr tuple_size = std::tuple_size<_T>{}();
    auto constexpr x = 0;
    auto constexpr y = 1;

    vertex result = (std::get<_Ix>(tuple) + ...);
    result[x] /= tuple_size;
    result[y] /= tuple_size;

    return result;
}

template<typename _T, std::size_t... _Ix>
auto print_points2d(std::ostream& out, const _T& tuple, std::index_sequence<_Ix...>) {
    auto constexpr tuple_size = std::tuple_size<_T>{}();
    (out << ... << std::get<_Ix>(tuple));
}

}

template<typename _T>
double area2d(const _T& tuple) {
    auto constexpr tuple_size = std::tuple_size<_T>{}();
    using vertex = std::remove_reference_t<decltype(std::get<0>(tuple))>;
    return detail::area2d(tuple, detail::make_index_sequence_with_offset<1, tuple_size -
}

template<typename _T>
auto center2d(const _T& tuple) {
    auto constexpr tuple_size = std::tuple_size<_T>{}();
    return detail::center2d(tuple, std::make_index_sequence<tuple_size>{});
}

template<typename _T>
auto print2d(std::ostream& stream, const _T& tuple) {
    auto constexpr tuple_size = std::tuple_size<_T>{}();

```

```

using std::endl;

stream << "\ntype:  ";
switch (tuple_size) {
case 4:
    stream << "rhombus" << endl; break;
case 5:
    stream << "pentagon" << endl; break;
case 6:
    stream << "hexagon" << endl; break;
default:
    stream << "unknown" << endl;
}

stream << "center: " << center2d(tuple) << endl
    << "area:  " << area2d(tuple) << endl
    << "points: ";
detail::print_points2d(stream, tuple, std::make_index_sequence<tuple_size>{});
stream << endl << endl;
}

#include <algorithm>

// Define Infinite (Using INT_MAX caused overflow problems)
#define INF 10000

//struct Point
//{
//    int x;
//    int y;
//};

typedef oop::editor::vec2 Point;

// Given three colinear points p, q, r, the function checks if
// point q lies on line segment 'pr'
bool on_segment(Point p, Point q, Point r)
{
    using namespace std;
    if (q.x <= max(p.x, r.x) && q.x >= min(p.x, r.x) &&
        q.y <= max(p.y, r.y) && q.y >= min(p.y, r.y))
        return true;
}

```

```

        return false;
    }

    // To find orientation of ordered triplet (p, q, r).
    // The function returns following values
    // 0 --> p, q and r are colinear
    // 1 --> Clockwise
    // 2 --> Counterclockwise
    int orientation(Point p, Point q, Point r)
    {
        int val = (q.y - p.y) * (r.x - q.x) -
            (q.x - p.x) * (r.y - q.y);

        if (val == 0) return 0; // colinear
        return (val > 0) ? 1 : 2; // clock or counterclock wise
    }

    // The function that returns true if line segment 'p1q1'
    // and 'p2q2' intersect.
    bool do_intersect(Point p1, Point q1, Point p2, Point q2)
    {
        // Find the four orientations needed for general and
        // special cases
        int o1 = orientation(p1, q1, p2);
        int o2 = orientation(p1, q1, q2);
        int o3 = orientation(p2, q2, p1);
        int o4 = orientation(p2, q2, q1);

        // General case
        if (o1 != o2 && o3 != o4)
            return true;

        // Special Cases
        // p1, q1 and p2 are colinear and p2 lies on segment p1q1
        if (o1 == 0 && on_segment(p1, p2, q1)) return true;

        // p1, q1 and p2 are colinear and q2 lies on segment p1q1
        if (o2 == 0 && on_segment(p1, q2, q1)) return true;

        // p2, q2 and p1 are colinear and p1 lies on segment p2q2
        if (o3 == 0 && on_segment(p2, p1, q2)) return true;
    }

```

```

    // p2, q2 and q1 are colinear and q1 lies on segment p2q2
    if (o4 == 0 && on_segment(p2, q1, q2)) return true;

    return false; // Doesn't fall in any of the above cases
}

// Returns true if the point p lies inside the polygon[] with n vertices
bool is_inside(Point polygon[], int n, Point p)
{
    // There must be at least 3 vertices in polygon[]
    if (n < 3) return false;

    // Create a point for line segment from p to infinite
    Point extreme = { INF, p.y };

    // Count intersections of the above line with sides of polygon
    int count = 0, i = 0;
    do
    {
        int next = (i + 1) % n;

        // Check if the line segment from 'p' to 'extreme' intersects
        // with the line segment from 'polygon[i]' to 'polygon[next]'
        if (do_intersect(polygon[i], polygon[next], p, extreme))
        {
            // If the point 'p' is colinear with line segment 'i-next',
            // then check if it lies on segment. If it lies, return true,
            // otherwise false
            if (orientation(polygon[i], p, polygon[next]) == 0)
                return on_segment(polygon[i], p, polygon[next]);

            count++;
        }
        i = next;
    } while (i != 0);

    // Return true if count is odd, false otherwise
    return count & 1; // Same as (count%2 == 1)
}

```

src/geom/point.hpp

```
#pragma once

#include <iostream>
#include <cstdint>
#include <cassert>
#include <cmath>

template<typename _Type, size_t _Dimensions>
struct point {
    static_assert(_Dimensions != 0, "can not create 0d point");

    using type = _Type;
    using reference = type&;
    using const_reference = const type&;
    using pointer = type*;
    using const_pointer = const type*;

    using iterator = pointer;
    using const_iterator = const_pointer;

    type dots[_Dimensions];

    type& operator[](size_t ix) noexcept {
        return dots[ix];
    }

    const type& operator[](size_t ix) const noexcept {
        return const_cast<point&>(*this).operator[](ix);
    }

    iterator begin() noexcept {
        return &dots[0];
    }

    const_iterator begin() const noexcept {
        return const_cast<point&>(*this).begin();
    }

    iterator end() noexcept {
        return &dots[_Dimensions];
    }
}
```



```

const_iterator end() const noexcept {
    return const_cast<point*>(*this).end();
}

static constexpr size_t size() noexcept {
    return _Dimensions;
}

point operator+(const point& other) const {
    point result = *this;

    for (size_t i = 0; i < result.size(); i++) {
        result[i] += other[i];
    }

    return result;
}

point operator-(const point& other) const {
    point result = *this;

    for (size_t i = 0; i < result.size(); i++) {
        result[i] -= other[i];
    }

    return result;
}
};

template<typename _Type, size_t _Dims>
std::ostream& operator<<(std::ostream& stream, const point<_Type, _Dims>& p) {
    stream << "{ ";
    for (const auto& d : p) {
        stream << d << " ";
    }
    stream << "}";

    return stream;
}

template<typename _Type, size_t _Dims>

```

```

std::istream& operator>>(std::istream& stream, point<_Type, _Dims>& p) {
    for (auto& d : p) {
        stream >> d;
    }

    return stream;
}

// Examples:
using point2d = point<double, 2>;

inline double distance(const point2d& left, const point2d& right) {
    double x = left[0] - right[0];
    double y = left[1] - right[1];
    return std::sqrt((x * x) + (y * y));
}

```

src/geom/polygon.hpp

```
#pragma once
```

```
#include <cstdint> // size_t  
#include <tuple>  
#include <type_traits>  
#include <istream>  
#include <ostream>  
#include <stdexcept>  
#include <vector>
```

```
/*  
    basic_polygon_traits  
*/
```

```
template<typename _Vertex>  
struct basic_polygon_traits {  
    using vertex      = _Vertex;  
    using pointer      = vertex*;  
    using const_pointer = const vertex*;  
    using reference    = vertex&;  
    using const_reference = const vertex&;  
  
    using iterator      = pointer;  
    using const_iterator = const_pointer;  
};
```

```
/*  
    basic_polygon class  
    tuple-like  
    structured binding is available  
*/
```

```
template<typename _Vertex, size_t _NumOfPoints>  
class basic_polygon {  
    static_assert(_NumOfPoints >= 3, "can not create polygon from points when there are  
    using traits = basic_polygon_traits<_Vertex>;  
  
    struct tag_prepare_initializer{};  
    struct tag_emplace_initializer{};  
public:  
    using vertex      = typename traits::vertex;  
    using pointer      = typename traits::pointer;  
    using const_pointer = typename traits::const_pointer;
```

```

using reference      = typename traits::reference;
using const_reference = typename traits::const_reference;

using iterator       = typename traits::iterator;
using const_iterator = typename traits::const_iterator;

// constructors
basic_polygon() = default;

basic_polygon(std::istream& stream) {
    for (auto& point : points) {
        stream >> point;
    }
    if (stream.fail()) {
        throw std::runtime_error("bad polygon initialization");
    }
}

basic_polygon(const vertex& v) noexcept {
    for (auto& point : points) {
        point = v;
    }
}

basic_polygon(const std::vector<vertex>& v) noexcept {
    if (v.size() < _NumOfPoints) {
        throw std::runtime_error("too few vertices for initialization");
    }
}

// element getters
reference at(size_t ix) {
    return points[ix];
}

const_reference at(size_t ix) const {
    return const_cast<basic_polygon*>(*this).at(ix);
}

reference operator[](size_t ix) {
    return at(ix);
}

```

```

const_reference operator[](size_t ix) const {
    return const_cast<basic_polygon*>(*this)[ix];
}

// iterators
iterator begin() {
    return &points[0];
}
const_iterator begin() const {
    // cast const to mutable and use non-const begin
    return const_cast<basic_polygon*>(*this).begin();
}

/* NEVER DEREFERENCE */
iterator end() {
    return &points[_NumOfPoints];
}
/* NEVER DEREFERENCE */
const_iterator end() const {
    // cast const to mutable and use non-const end
    return const_cast<basic_polygon*>(*this).end();
};

// structured binding
template<size_t _Ix>
constexpr auto& get() & {
    // check out of bounds
    if constexpr (_Ix < _NumOfPoints) {
        return points[_Ix];
    }
    else {
        // generate compile-time error
        static_assert(_Ix < _NumOfPoints, "ix is out of range");
    }
}

template<size_t _Ix>
constexpr auto const& get() const& {

```

```

        // cast const to mutable and use non-const get
        // which does no effect on storage
        return const_cast<basic_polygon*>(*this).get<_Ix>();
    }

    template<size_t _Ix>
    constexpr auto&& get() && {
        // cast lvalue reference to rvalue and return it
        return std::move(this->get<_Ix>());
    }

    constexpr size_t size() const {
        return _NumOfPoints;
    }

private:
    vertex points[_NumOfPoints];

    template<size_t _Ix, typename _V, size_t _N>
    friend constexpr auto std::get(const basic_polygon<_V, _N>& polygon);
};

// std types spetializations for structured binding of basic_polygon
namespace std {
    template<size_t _Ix, typename _Vertex, size_t _NumOfPoints>
    constexpr auto get(const basic_polygon<_Vertex, _NumOfPoints>& polygon) {
        return polygon.points[_Ix];
    }

    template<typename _Vertex, size_t _NumOfPoints>
    struct tuple_size<::basic_polygon<_Vertex, _NumOfPoints>>
        : integral_constant<size_t, _NumOfPoints> {};

    template<size_t _Ix, typename _Vertex, size_t _NumOfPoints>
    struct tuple_element<_Ix, ::basic_polygon<_Vertex, _NumOfPoints>> {
        using type = typename basic_polygon_traits<_Vertex>::vertex;
    };
} // namespace std

```

src/system/application_base.cpp

```
#include "application_base.hpp"

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

#include <atomic>
#include <stdexcept>

using namespace oop::system;

std::atomic_bool application_base::running_ = false;

application_base::application_base()
    : window_{nullptr}
    , renderer_{nullptr}
{
    auto expected = false;
    if (auto const ok = running_.compare_exchange_strong(expected, true); !ok)
    {
        throw std::runtime_error("application: already running");
    }

    window_ = SDL_CreateWindow("", SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED, 800,
    renderer_ = SDL_CreateRenderer(window_, -1, SDL_RENDERER_SOFTWARE);

    ImGui::CreateContext();

    // ImGui_ImplSDL2_InitForOpenGL is a wrapper for ImGui_ImplSDL2_Init which ignores s
    ImGui_ImplSDL2_InitForOpenGL(window_, nullptr);

    ImGuiSDL::Initialize(renderer_, 800, 600);
    ImGui_ImplSDL2_NewFrame(window_);
    ImGui::NewFrame();
}

application_base::~application_base()
{
    ImGuiSDL::Deinitialize();
    ImGui::DestroyContext();
    SDL_DestroyRenderer(renderer_);
}
```

```

        SDL_DestroyWindow(window_);
        running_.store(false);
    }

    int application_base::start(std::string_view name, int argc, char* argv[])
    {
        //    configure();
        run();
        return 0;
    }

    void application_base::run()
    {
        while (!done_)
        {
            process_events();
            construct_frame();
            update();
        }
    }

    void application_base::process_events()
    {
        SDL_Event event;
        while (SDL_PollEvent(&event))
        {
            ImGui_ImplSDL2_ProcessEvent(&event);
            process_event(event);

            if (event.type == SDL_QUIT)
            {
                done_ = true;
            }
        }
    }

    void application_base::process_event(const SDL_Event& event)
    {
        // Placeholder
    }

    void application_base::construct_frame()

```



```
{  
    // Placeholder  
}  
  
void application_base::update() const  
{  
    renderer_.update(window_);  
}
```

src/system/application_base.hpp

```
#pragma once

#include <atomic>
#include <string>

#include "system/renderer.hpp"

namespace oop::system
{
    class application_base
    {
    protected:
        SDL_Window* window_;
        renderer    renderer_;

    public:
        /*!
         * @brief application entry point
         *
         * Starts new application instance; one per process.
         *
         *
         * @throws std::runtime_error
         * If here is already another instance exception will be throum.
         */
        int start(std::string_view name, int argc, char* argv[]);
        int start(int argc, char* argv[])
        {
            return start("SDL2 Window", argc, argv);
        }

    protected:
        application_base();
        virtual ~application_base();

    private:
        void run();
        void update() const;
        void process_events();

        virtual void process_event(const SDL_Event& event);
    };
}
```

```

    virtual void construct_frame();

public:
    application_base(const application_base&) = delete;
    application_base(application_base&&) noexcept = delete;

    application_base& operator=(const application_base&) = delete;
    application_base& operator=(application_base&&) noexcept = delete;

private:
    static std::atomic_bool running_;
    bool done_ = false;
};
}

```

src/system/renderer.hpp

```
#pragma once

#include <SDL.h>

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

namespace oop::system
{
    struct renderer
    {
        renderer() = default;

        explicit renderer(SDL_Renderer* renderer)
            : renderer_{ renderer }
        {}

        renderer& operator=(SDL_Renderer* renderer)
        {
            renderer_ = renderer;
            return *this;
        }

        void update(SDL_Window* window) const {
            ImGui::Render();
            ImGuiSDL::Render(ImGui::GetDrawData());
            SDL_RenderPresent(renderer_);
            ImGui_ImplSDL2_NewFrame(window);
            ImGui::NewFrame();
        }

        [[nodiscard]]
        SDL_Renderer& operator*() const noexcept
        {
            return *renderer_;
        }

        SDL_Renderer& operator->() const noexcept
        {
            return *renderer_;
        }
    };
}
```

```

    }

    [[nodiscard]]
    operator SDL_Renderer*() const noexcept
    {
        return renderer_;
    }

private:
    SDL_Renderer* renderer_;
};
}

```

src/system/sdl2.cpp

```
#include "sdl2.hpp"
```

```
void oop::system::draw_circle(SDL_Renderer* surface, int n_cx, int n_cy, int radius)
{
    // if the first pixel in the screen is represented by (0,0) (which is in sdl)
    // remember that the beginning of the circle is not in the middle of the pixel
    // but to the left-top from it:

    double error = -radius;
    auto x = radius - 0.5;
    auto y = 0.5;
    const auto cx = n_cx - 0.5;
    const auto cy = n_cy - 0.5;

    while (x >= y)
    {
        SDL_RenderDrawPoint(surface, int(cx + x), int(cy + y));
        SDL_RenderDrawPoint(surface, int(cx + y), int(cy + x));

        if (x != 0)
        {
            SDL_RenderDrawPoint(surface, int(cx - x), int(cy + y));
            SDL_RenderDrawPoint(surface, int(cx + y), int(cy - x));
        }

        if (y != 0)
        {
            SDL_RenderDrawPoint(surface, int(cx + x), int(cy - y));
            SDL_RenderDrawPoint(surface, int(cx - y), int(cy + x));
        }

        if (x != 0 && y != 0)
        {
            SDL_RenderDrawPoint(surface, int(cx - x), int(cy - y));
            SDL_RenderDrawPoint(surface, int(cx - y), int(cy - x));
        }

        error += y;
        ++y;
        error += x;
    }
}
```

```
        if (error >= 0)
        {
            --x;
            error -= x;
            error -= x;
        }
    }
}
```

src/system/sdl2.hpp

```
#pragma once
```

```
#include "SDL.h"
```

```
namespace oop::system {  
    void draw_circle(SDL_Renderer* surface, int n_cx, int n_cy, int radius);  
}
```