CG Lab Project1: OpenGL 基础绘制 实验报告

袁晨圃 2023K8009929012

Repo cauphenuny/ucas-graphics

一、 实验环境

1.1 依赖项

基础依赖：比较新的 C++编译器、`xmake`、`opengl`、`glfw`

为了让 coding 比较舒服，加了点别的库：`fmtlib`, `spdlog`, `magic_enum`

```lua
-- xmake.lua
add_rules("mode.debug", "mode.release")
set_languages("c99", "c++20")
add_requires("opengl", {system = true})
add_requires("glut", {system = true})
add_requires("glfw3", {system = true})
add_requires("spdlog", {system = true})
add_requires("magic_enum")

target("project1")
    set_kind("binary")
    add_files("src/basics/*.cpp")
    add_packages("opengl")
    add_packages("glut")
    add_packages("spdlog")
    add_packages("glfw3")
    add_packages("magic_enum")
    if is_plat("macosx") then
        add_frameworks("Cocoa", "CoreFoundation", "IOKit")
    end
```

由于在我的 macOS 环境下 xmake/vcpkg 安装的 `opengl/glfw/spdlog` 库有点问题，所以改成用系统的库了，在编译前需要先安装这些依赖。

header-only 的 `magic_enum` 没有问题，能直接从 xmake 装。

1.2 编译 & 运行

```shell
xmake
xmake run project1
```

会先启动一个显示小电脑的窗口，关闭后，进入画板，可以画图

启动的时候可以选择主题

```shell
xmake run project1 catppuccin # default
xmake run project1 xterm-dark
xmake run project1 catppuccin-dark
```

二、 实验效果

2.1 demo

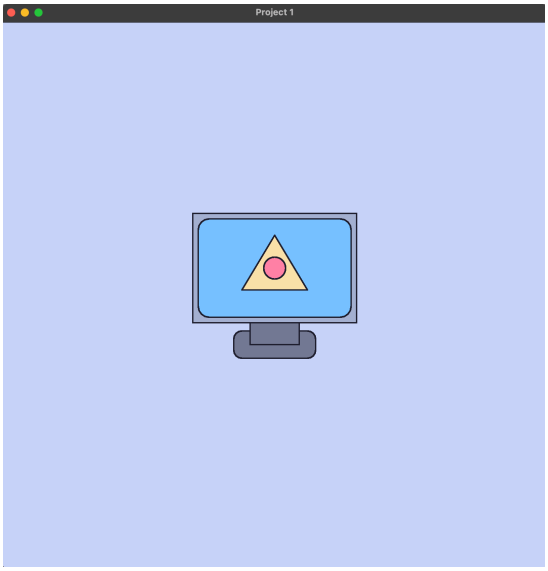

图 1　computer

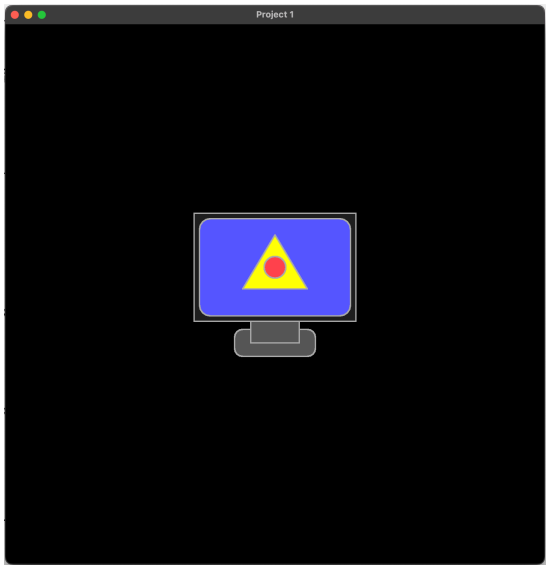

图 2　computer, theme=xterm-dark

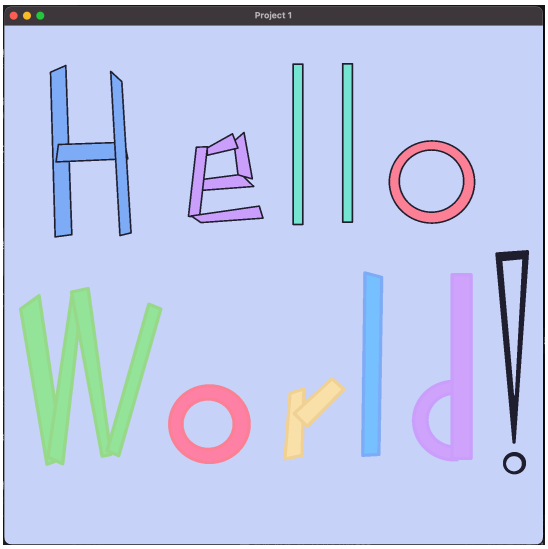

图 3　画板画图 demo

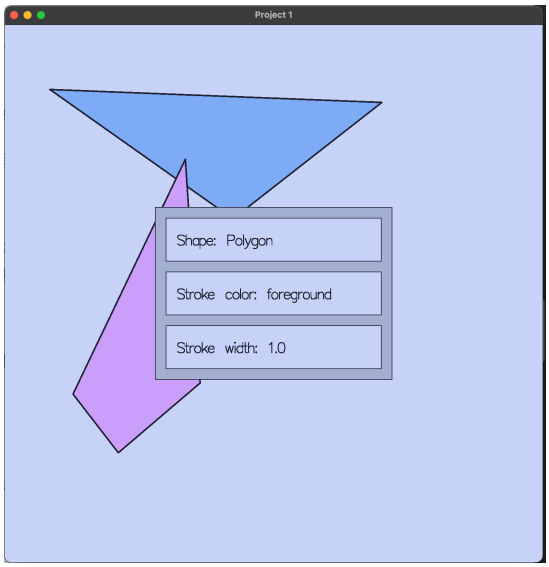

图 4　画板选项栏

2.2 操作说明

画板功能：

- 支持绘制直线、折线、矩形、圆形、三角形、多边形：鼠标左键选点，右键或者 ESC 提交当前图形
- 支持撤销操作：按 Backspace 撤销上一个图形
- 支持的绘制选项：描边颜色、描边宽度、填充颜色、是否圆角（矩形）。按空格打开菜单或者绘制完图形后自动打开填充颜色选项栏，菜单打开后使用左键切换选项，右键/ESC 退出。

三、 代码明细

3.1 整体架构

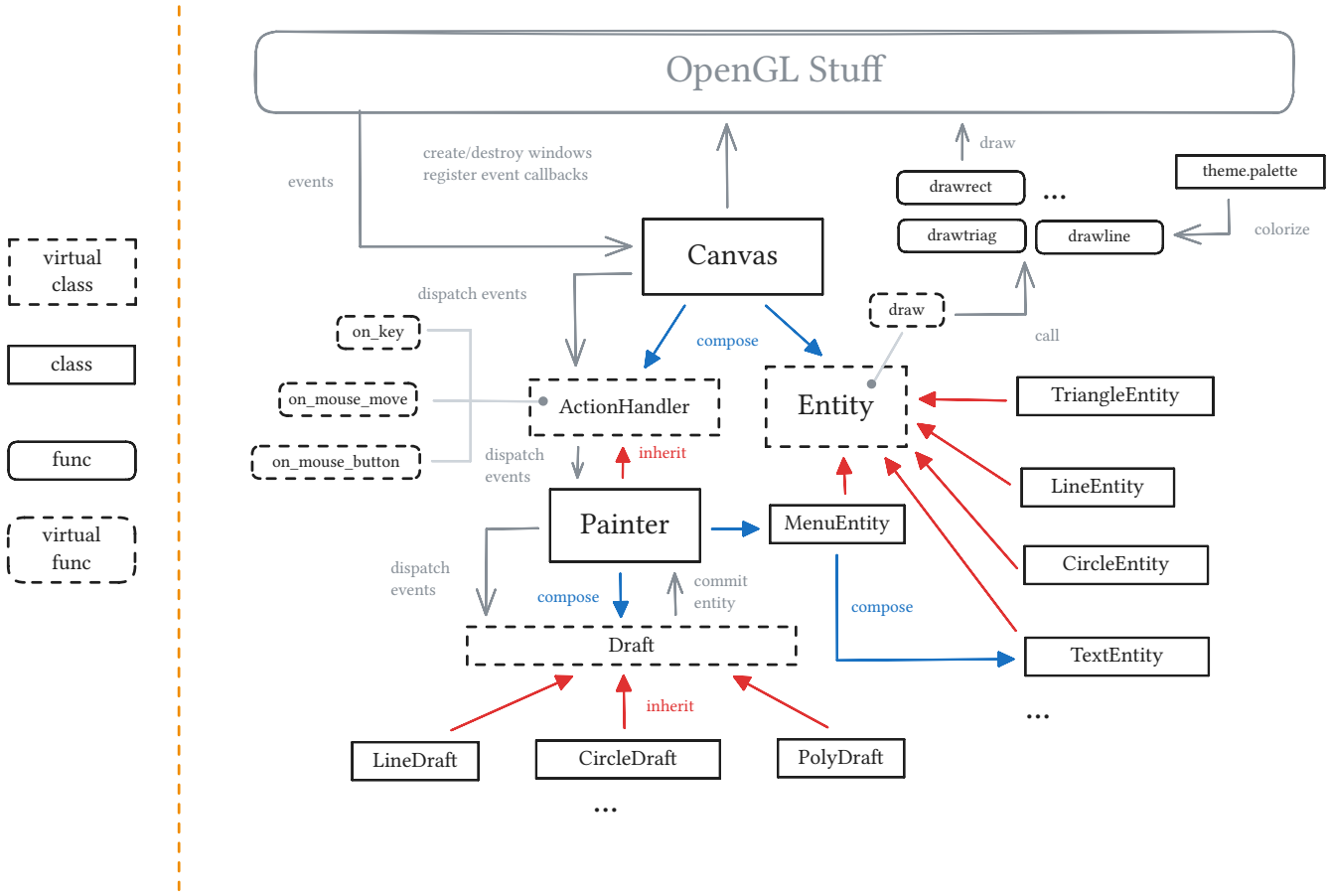

图 5 架构图

```
basics/
├── coord.hpp:    坐标表示 Vertex2d
├── color.hpp:    颜色表示 Color, 预设的主题调色盘 Palette
├── draw.hpp:     图形绘制函数 draw::line, draw::circle, ...
├── entity.hpp:   图形实体基类 Entity 及其派生类 Line, Rectangle, Circle, ...
├── canvas.hpp:   Canvas 负责 GLFW 初始化及主循环
├── drafts.hpp:   草图管理，显示正在绘制的图形以及生成完整图形给 painter
├── painter.hpp:  画板主逻辑，处理用户输入并调用 drafts 和 canvas
└── main.cpp:     程序入口，处理命令行参数并启动小电脑和画板
```

3.2 基础图形绘制

3.2.1 坐标和颜色

使用两个类 `Vertex2d` 和 `Color` 分别表示二维坐标和颜色

```cpp
struct Vertex2d {
    GLdouble x, y;
    Vertex2d(GLdouble x, GLdouble y) : x(x), y(y) {}
    template <typename T> Vertex2d(std::initializer_list<T> init); // supports {x, y}
};
```

```cpp
 struct Color {
     GLdouble red, green, blue, alpha{1.0};
```

```
    Color(GLubyte r, GLubyte g, GLubyte b, GLubyte a = 255)
        : red(r / 255.0), green(g / 255.0), blue(b / 255.0), alpha(a / 255.0) {}
    Color(GLdouble r, GLdouble g, GLdouble b, GLdouble a = 1.0)
        : red(r), green(g), blue(b), alpha(a) {}
    Color(HexColor hex);
    Color(const std::string_view color_name);
    Color(const char* color_name) : Color(std::string_view{color_name}) {}
};
```

这里传入 string_view 的构造函数可以根据调色盘返回颜色

```C++
enum class ColorID {
    WHITE = 0,  // background
    RED = 1,
    GREEN = 2,
    ...
};

struct Palette {
    unsigned int palette[18];
};

inline Color::Color(const std::string_view color_name) {
    std::string name{color_name};
    // convert to uppercase because we use uppercase in enum fields like "WHITE", "BLACK"
    std::transform(
        name.begin(), name.end(), name.begin(), [](unsigned char c) { return std::toupper(c); });
    auto color_enum = magic_enum::enum_cast<themes::ColorID>(name);
    if (color_enum.has_value()) {
        auto color = color_enum.value();
        unsigned int hex = current_theme.palette[static_cast<int>(color)];
        *this = Color(HexColor{hex});
    } else {
        throw std::runtime_error(fmt::format("Unknown color name: {}", color_name));
    }
}
```

因此，在代码中改变 current_theme 会改变主题，影响新构造出的颜色

3.2.2 窗口管理

通过全局单例类 GlfwContext 来初始化 glfw

```C++
inline struct GlfwContext {
    GlfwContext() {
        if (!glfwInit()) {
            throw std::runtime_error("glfw init failed");
        }
    }
    ~GlfwContext() { glfwTerminate(); }
} glfw_context;
```

通过 Canvas 类管理窗口和 OpenGL 上下文

```C++
struct CanvasParameters {
    std::string title;
    struct {
        int width{800}, height{800};
    } display_size;
    Color background{"background"};
```

```
    struct {
        GLdouble left{-5}, right{5}, bottom{-5}, top{5};
        GLdouble zNear{5}, zFar{15};
    } projection;
    struct {
        GLdouble eyeX, eyeY, eyeZ;
        GLdouble centerX, centerY, centerZ;
        GLdouble upX, upY, upZ;
    } view_point;
};

struct Canvas {
    const CanvasParameters params;
    GLFWwindow* window;
    ...
    Canvas(const CanvasParameters& params = CanvasParameters()) : params(params) {}

    void init() {
        auto& [title, display_size, bg, proj, view] = this->params;
        glfwMakeContextCurrent(this->window);
        glClearColor(bg.red, bg.green, bg.blue, bg.alpha);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        glOrtho(proj.left, proj.right, proj.bottom, proj.top, proj.zNear, proj.zFar);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
        gluLookAt(
            view.eyeX, view.eyeY, view.eyeZ, view.centerX, view.centerY, view.centerZ, view.upX,
            view.upY, view.upZ);
    }


    void window_init() {
        auto [w, h] = this->params.display_size;
        spdlog::info("creating GLFW window width: {}, height: {}", w, h);
        this->window = glfwCreateWindow(w, h, this->params.title.c_str(), NULL, NULL);
        if (!window) {
            throw std::runtime_error("glfw create window failed");
        }
    }

    ...

    // 主循环
    void spin() {
        // 初始化窗口
        this->window_init();
        this->init();

        // set_action_handler 是设置当前的 handler，但是窗口初始化后才能把当前handler attach到这个窗口
        if (this->action_handler) this->attach_handler(this->action_handler.get());
        while (!glfwWindowShouldClose(this->window)) {
            glClear(GL_COLOR_BUFFER_BIT);

            // 根据优先级给所有实体排序，然后依次渲染
            std::vector<Entity*> sorted_entities(entities.begin(), entities.end());
            std::sort(sorted_entities.begin(), sorted_entities.end(), [this](Entity* a, Entity* b) {
                return this->entity_attributes[a].priority < this->entity_attributes[b].priority;
            });
```

```
            for (auto entity : sorted_entities) {
                entity->draw();
            }

            glfwSwapBuffers(window);
            glfwPollEvents();
        }
        glfwDestroyWindow(this->window);
        this->window = nullptr;
    }
};
```

3.2.3 绘制图形

图形绘制函数都在 `draw` 命名空间下，这个命名空间下的函数不负责保存图形状态，只负责绘制，图形状态由 `Entity` 保存（类似 `torch.nn.functional`）

另外我发现直接画线的话线段的粗细不够，而 mac 上 `glLineWidth` 似乎有问题，OpenGL 标准没有要求实现 width $\neq 1$ 的 `glLineWidth`，所以代码里都是用 rect 模拟线段的

C++

```
// namespace draw
inline constexpr double kLineWidthScale = 0.03;

inline void rect_filled(const opengl::Vertex2d& center, double w, double h, opengl::Color color);
inline void
circle_filled(const opengl::Vertex2d& center, double radius, opengl::Color color, int segments);

inline void line(Vertex2d start, Vertex2d end, Color color, double width = 1.0) {
    double scaled_width = width * kLineWidthScale;
    if (scaled_width <= 0.0) {
        return;
    }

    double dx = end.x - start.x;
    double dy = end.y - start.y;
    double length = std::hypot(dx, dy);
    if (length <= std::numeric_limits<double>::epsilon()) {
        rect_filled(start, scaled_width, scaled_width, color);
        return;
    }

    Vertex2d center{(start.x + end.x) * 0.5, (start.y + end.y) * 0.5};
    double angle_deg = std::atan2(dy, dx) * 180.0 / std::numbers::pi_v<double>;

    glPushMatrix();
    glTranslated(center.x, center.y, 0.0);
    glRotated(angle_deg, 0.0, 0.0, 1.0);
    rect_filled(Vertex2d{0.0, 0.0}, length, scaled_width, color);
    glPopMatrix();
}
```

这里的 `glPushMatrix` 和 `glPopMatrix` 用来保存和恢复矩阵状态，避免对其他绘制造成影响，减少函数副作用

为了折线之间显示平滑，在拼接两边的线段处加了个圆角

```C++
// namespace draw
inline void
draw_polyline(const std::vector<Vertex2d>& points, bool closed, Color color, double width) {
    if (points.size() < 2) {
        return;
    }

    double joint_radius = 0.5 * width * kLineWidthScale;
    if (joint_radius > 0.0) {
        constexpr int kJointSegments = 18;
        for (const auto& p : points) {
            circle_filled(p, joint_radius, color, kJointSegments);
        }
    }

    for (std::size_t i = 1; i < points.size(); ++i) {
        line(points[i - 1], points[i], color, width);
    }
    if (closed) {
        line(points.back(), points.front(), color, width);
    }
}
```

用多个线段模拟画圆/画弧

```C++
// 画圆（描边）
// center: 圆心, radius: 半径, color: 颜色, segments: 分段数（越大越圆）
inline void circle_outline(
    const opengl::Vertex2d& center, double radius, opengl::Color color, int segments = 64,
    double line_stroke = 1.0) {
    if (radius <= 0.0) {
        return;
    }
    int segs = std::max(3, segments);
    std::vector<Vertex2d> points;
    points.reserve(segs);
    for (int i = 0; i < segs; ++i) {
        double angle = 2.0 * std::numbers::pi_v<double> * static_cast<double>(i) / segs;
        detail::append_point(
            points,
            Vertex2d{center.x + radius * std::cos(angle), center.y + radius * std::sin(angle)});
    }
    detail::draw_polyline(points, true, color, line_stroke);
}
```

文字这里比较 tricky，通过多次偏移画同一个字符来加粗字体

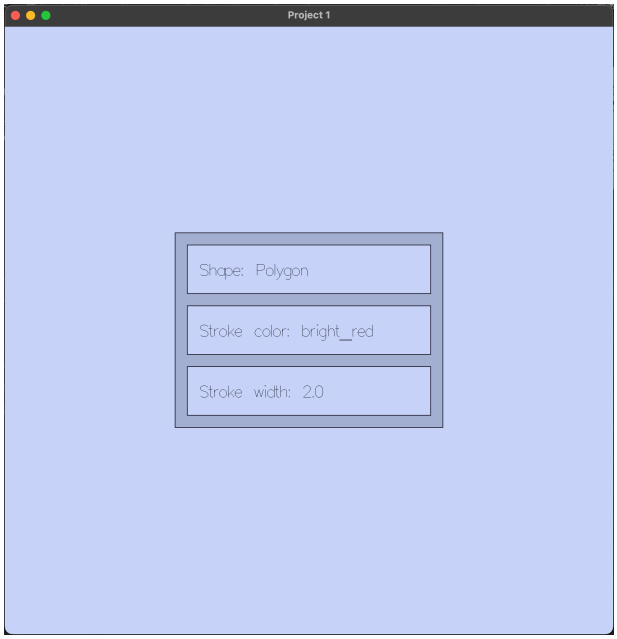

图 6 加粗前

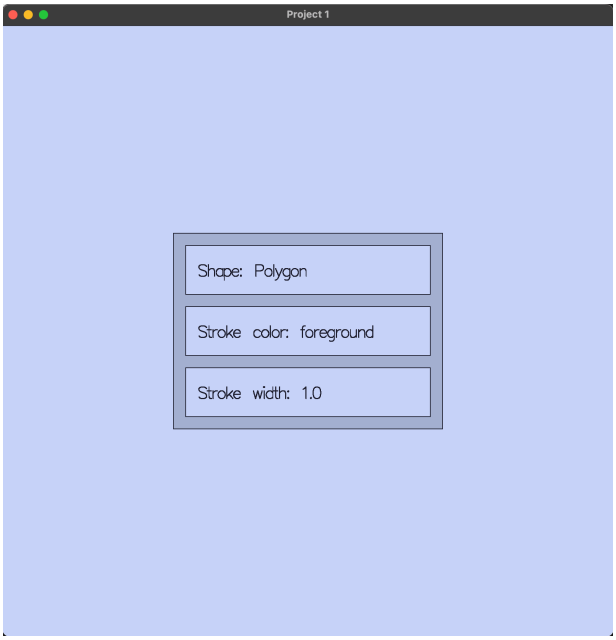

图 7 加粗后

```C++
inline void text(
    const opengl::Vertex2d& origin, const std::string& content, opengl::Color color,
    double scale = 1.0) {
    if (content.empty()) {
        return;
    }
    double normalized_scale = std::max(0.001, 0.0027 * scale);
    float line_width = std::max(1.0f, static_cast<float>(2.0 * scale));
    constexpr double kStrokeFontHeight = 110;
    glPushMatrix();
    glColor3d(color.red, color.green, color.blue);
    glTranslated(origin.x, origin.y - normalized_scale * kStrokeFontHeight * 0.5, 0.0);
    glScaled(normalized_scale, normalized_scale, 1.0);
    // draw multiple passes with symmetric offsets to thicken strokes without shifting position
    const double offset = 3.0;  // in stroke-font units (before scaling)
    const double step = 0.3;
    auto offsets =
        std::views::iota(-static_cast<int>(offset / step), static_cast<int>(offset / step) + 1) |
        std::views::transform([step](int idx) { return static_cast<double>(idx) * step; });
    for (double ox : offsets) {
        for (double oy : offsets) {
            glPushMatrix();
            glTranslated(ox, oy, 0.0);
            for (unsigned char ch : content) {
                glutStrokeCharacter(GLUT_STROKE_ROMAN, ch);
            }
            glPopMatrix();
        }
    }
    glPopMatrix();
}
```

3.3 实体管理

渲染函数肯定不能全部写在 Canvas 的主循环里，所以需要某个东西保存当前画了哪些形状，然后在 Canvas::spin 里逐个调用它们的 draw 方法

定义一个 Entity 抽象基类，所有图形实体都继承自它，类型擦除之后可以直接保存在 Canvas 的容器里

```C++
struct Entity {
    virtual void draw() const = 0;
    virtual std::string repr() const = 0;
    virtual ~Entity();
    Canvas* container;
    Entity(Canvas* canvas);
    auto attribute() const -> EntityAttribute;
    void set_priority(int) const;
};
```

EntityAttribute 目前只有一个 priority 字段，用来控制渲染顺序，数值越小越先渲染，越高显示优先级越高

下面是个例子：Line 和 LineEntity 分别表示线段的配置和实体

```C++
 struct Line {
     Vertex2d start;
     Vertex2d end;
     Color color{"foreground"};
     double stroke{1.0};
     using EntityType = LineEntity;
 };

 struct LineEntity : Entity {
     Line config;
     LineEntity(Canvas* canvas, Line config) : Entity(canvas), config(config) {}
     void draw() const override {
         draw::line(config.start, config.end, config.color, config.stroke);
     }
     std::string repr() const override {
         return fmt::format(
             "Line(start={}, end={}, color={}, stroke={})", config.start, config.end, config.color,
             config.stroke);
     }
 };
```

那生命周期怎么管理呢？原先的设计是 Canvas 创建后返回一个 handler，然后类外部通过操作 handler 来释放实体。但是这样其实跟把 Entity 本身作为一个 handler 没什么区别，代码复杂度还更高，所以现在生命周期直接由 Entity 类本身管理，用 this 作为 key，Entity 析构时自动从 Canvas 容器里删除自己

Canvas 里面只保存指向 Entity 的裸指针，不负责实体的内存管理

```C++
inline Entity::Entity(Canvas* canvas) : container(canvas) { this->container->add_entity(this); }
inline Entity::~Entity() {
    if (this->container) this->container->delete_entity(this);
    spdlog::info("entity {} destructed", (void*)this);
}
```

如果 Canvas 先于 Entity 析构，也是安全的，因为只有 Canvas 会调用 draw()，只要把 Entity 里面的 container 指针清了让 Entity 析构的时候不在 container (已失效) 中注销自己就行

唯一需要担心的问题是 Entity 内存泄露，但用 std::unique_ptr 管理所有权可以解决这一问题。

由于配置类里面定义了实体类的类型 using EntityType = LineEntity;，所以可以通过模板函数 Canvas::draw(some_config) 来添加实体，不需要指定模版参数 draw<SomeShape>(SomeShapeConfig(.key1 = ...)) 了

```C++
auto Canvas::draw(auto config) {
    auto entity = std::make_unique<typename decltype(config)::EntityType>(this, config);
```

```
    spdlog::info("draw: {}, id={}, canvas={}", entity->repr(), (void*)entity.get(), (void*)this);
    return entity;
}
```

使用 Config 类创建实体可以做到一种类似 python kwargs 风格的调用

例如创建小电脑的代码如下：

```C++
struct Computer {
    std::set<std::unique_ptr<Entity>> parts;
    Computer(Canvas* canvas, double center_x = 0.0, double center_y = 0.0) {
        auto screen_x = center_x, screen_y = center_y + 0.5;
        double screen_w = 3, screen_h = 2;
        double base_delta = 0.4;
        double base_x = screen_x, base_y = screen_y - screen_h / 2 - base_delta;
        double base_w = 1.5, base_h = 0.5;
        double margin = 0.2;
        auto base_color = mix("foreground", "background", 0.5);
        auto margin_color = mix("foreground", "background", 0.8);
        parts.insert(canvas->draw(
            Rectangle{
                .center = {base_x, base_y},
                .width = base_w,
                .height = base_h,
                .corner_radius = 0.15,
                .fill_color = base_color,
            }));
        parts.insert(canvas->draw(
            Rectangle{
                .center = {base_x, screen_y - screen_h / 2},
                .width = base_w * 0.6,
                .height = base_delta * 2,
                .fill_color = base_color,
            }));
        parts.insert(canvas->draw(
            Rectangle{
                .center = {screen_x, screen_y},
                .width = screen_w,
                .height = screen_h,
                .fill_color = margin_color,
            }));
        parts.insert(canvas->draw(
            Rectangle{
                .center = {screen_x, screen_y},
                .width = screen_w - margin,
                .height = screen_h - margin,
                .corner_radius = 0.2,
                .fill_color = "bright_blue",
            }));
        double triangle_l1 = 0.6, triangle_l2 = 0.4;
        parts.insert(canvas->draw(
            Triangle{
                .p1 = {screen_x - triangle_l1, screen_y - triangle_l2},
                .p2 = {screen_x + triangle_l1, screen_y - triangle_l2},
                .p3 = {screen_x, screen_y + triangle_l2 * 1.5},
                .fill_color = "bright_yellow",
            }));
        double circle_r = 0.2;
        parts.insert(canvas->draw(
            Circle{
```

```
                .center = {screen_x, screen_y},
                .radius = circle_r,
                .fill_color = "bright_red",
            }));
    }
};
```

以典型的 Rectangle 为例：

```C++
struct Rectangle {
    Vertex2d center;
    double width;
    double height;
    std::optional<double> corner_radius{};
    Color color{"foreground"};  // stroke color
    std::optional<Color> fill_color = Color{"foreground"};
    double stroke{1.0};
    using EntityType = RectangleEntity;
};
```

有以下字段：center, width, height, corner_radius, color, fill_color, stroke，其中 corner_radius 和 fill_color 是可选的，传入 std::nullopt 表示不使用圆角或填充颜色

3.4 处理事件输入

输入事件通过 ActionHandler 接口在 Canvas 和业务层之间传递。Canvas::attach_handler 会把一组 GLFW 回调绑定到当前窗口，同时把 ActionHandler 指针塞进 glfwSetWindowUserPointer，因此所有键盘/鼠标事件都会被转发到 on_key、on_mouse_button、on_mouse_move。

```C++
struct ActionHandler {
    virtual void on_key(int key, int action) = 0;
    virtual void on_mouse_button(int button, int action, int mods) = 0;
    virtual void on_mouse_move(double xpos, double ypos) = 0;
    void attach(Canvas* canvas);
};
```

通过 UserPointer，GLFW 的纯函数回调可以找到具体的 ActionHandler 实例：

```C++
void Canvas::attach_handler(ActionHandler* handler) {
    auto key_callback = [](GLFWwindow* window, int key, int scancode, int action, int mods) {
        ActionHandler* self = static_cast<ActionHandler*>(glfwGetWindowUserPointer(window));
        self->on_key(key, action);
    };
    auto mouse_button_callback = [](GLFWwindow* window, int button, int action, int mods) {
        ActionHandler* self = static_cast<ActionHandler*>(glfwGetWindowUserPointer(window));
        self->on_mouse_button(button, action, mods);
    };
    auto mouse_move_callback = [](GLFWwindow* window, double xpos, double ypos) {
        ActionHandler* self = static_cast<ActionHandler*>(glfwGetWindowUserPointer(window));
        self->on_mouse_move(xpos, ypos);
    };
    glfwSetWindowUserPointer(this->window, handler);
    glfwSetKeyCallback(this->window, key_callback);
    glfwSetMouseButtonCallback(this->window, mouse_button_callback);
    glfwSetCursorPosCallback(this->window, mouse_move_callback);
    handler->attach(this);
}
```

`Painter` 继承自 `ActionHandler`：

- 按下 `Space` 打开主菜单，`Esc` 关闭菜单或取消草稿，`Backspace` 撤销最近一次提交；
- 左键点击会查询当前鼠标位置，转换为世界坐标后交给草稿；右键点击等价于发送 `Esc`，用于快速终止当前操作；
- 鼠标移动持续刷新草稿的预览图形。

像素到世界坐标的转换由 `Painter::cursor_to_world` 完成，它根据投影体的左右/上下边界线性插值：

```C++
Vertex2d cursor_to_world(double xpos, double ypos) const {
    double nx = xpos / canvas->params.display_size.width;
    double ny = ypos / canvas->params.display_size.height;
    const auto& proj = canvas->params.projection;
    double world_x = proj.left + nx * (proj.right - proj.left);
    double world_y = proj.top - ny * (proj.top - proj.bottom);
    return Vertex2d{world_x, world_y};
}
```

整条输入链路是 “GLFW → Canvas → ActionHandler/Painter → Draft”。`Canvas` 仅负责把事件转发出去。

3.5 画板设计

`Painter` 是画板系统的核心，负责：

1. 维护当前草稿 `current_draft` 以及绘制历史 `drawn_entities`，支持撤销/重建；
2. 管理样式（描边颜色/宽度、填充颜色、矩形圆角等）的循环选项，并能随时刷新草稿和最后一次实体；
3. 渲染及驱动菜单覆盖层，响应用户切换形状或参数的需求；
4. 把草稿提交的实体设置合适的优先级后交给 `Canvas`。

历史记录中的每一项会保存实体指针、优先级、图形类型以及一个 `rebuild` lambda。这样当用户在菜单里修改样式时，只需调用 `rebuild_last_entity()` 用最新样式重新生成一次实体即可；优先级（绘制顺序）仍然保持不变。

```C++
struct HistoryEntry {
    std::unique_ptr<Entity> entity;
    int priority;
    ShapeType shape;
    std::function<std::unique_ptr<Entity>(Canvas*, const DraftStyle&)> rebuild;
};

void Painter::handle_draft_commit(DraftCommit commit_info) {
    if (!commit_info.entity) return;
    int priority = allocate_committed_priority();
    commit_info.entity->set_priority(priority);
    drawn_entities.push_back(HistoryEntry{
        .entity = std::move(commit_info.entity),
        .priority = priority,
        .shape = commit_info.shape_type,
        .rebuild = std::move(commit_info.rebuild),
    });
    if (has_shape_specific_options(commit_info.shape_type)) {
        open_shape_menu();
    } else if (menu_state.kind == MenuKind::ShapeSpecific) {
        close_menu();
    }
```

```
23 }
```

3.5.1 菜单操作

菜单由 MenuState + MenuOverlayEntity 组合实现。MenuState 记录锚点、宽度、内边距以及 MenuItem 列表，MenuOverlayEntity 则根据状态绘制背景板、按钮和文字。为了保证菜单始终盖在最上层，实体优先级固定设置为 200000。

- 主菜单（Space）：展示 Shape / Stroke color / Stroke width 三项，可循环当前形状类型和描边样式，同时立即刷新草稿的预览实体。
- 形状菜单（自动）：当最近提交的形状支持填充或额外参数（矩形圆角）时弹出，允许修改 Fill、Corner radius 等。每次点击都会执行对应 action，再调用 rebuild_last_entity() 把形状按最新样式重建。
- 关闭逻辑：点击任意空白区域或按 Esc/RightClick 会把 MenuState.visible 置为 false，菜单实体仍然存在但不再渲染，下次按 Space 会重用。

菜单命中检测通过 MenuItem::contains 完成。鼠标事件会先尝试命中菜单，如果命中则执行 action 并刷新 layout，否则继续流向草稿逻辑，实现了 UI 与绘图操作的统一输入链。

```cpp
void Painter::ensure_menu_layer() {
    if (menu_layer || !canvas) return;
    MenuOverlay overlay{ .state = &menu_state };
    menu_layer = canvas->draw(overlay);
    menu_layer->set_priority(200000);
}

std::vector<MenuItem> Painter::build_main_menu_items() {
    return {
        MenuItem{
            .label = fmt::format("Shape: {}", magic_enum::enum_name(active_shape)),
            .action = [this]() { cycle_shape_type(); },
        },
        MenuItem{
            .label = fmt::format("Stroke color: {}", stroke_palette[stroke_color_index]),
            .action = [this]() { cycle_stroke_color(); },
        },
        MenuItem{
            .label = fmt::format(
                "Stroke width: {:.1f}", stroke_width_options[stroke_width_index]),
            .action = [this]() { cycle_stroke_width(); },
        },
    };
}
```

3.5.2 草稿处理

草稿层把“正在交互中的形状”与“已经提交的实体”分离：

- DraftContext：封装 Canvas*、预览颜色、样式提供器、提交回调以及“工作优先级”分配器；
- Draft 基类暴露 on_mouse_button/move、on_key、refresh_style、reset 等接口，派生类只需关心几何逻辑。

```cpp
struct DraftContext {
    Canvas* canvas{nullptr};
    Color preview_color{mix("foreground", "background", 0.8)};
    std::function<DraftStyle()> style_provider;
    std::function<void(DraftCommit)> commit_callback;
    std::function<int()> working_priority_allocator;
```

```
};

class Draft {
public:
    explicit Draft(DraftContext ctx) : ctx_(std::move(ctx)) {}
    virtual ~Draft() = default;

    virtual std::string name() const = 0;
    virtual void on_mouse_button(int button, int action, int mods, const Vertex2d& world) = 0;
    virtual void on_mouse_move(const Vertex2d& world) = 0;
    virtual void on_key(int key, int action) {}
    virtual void refresh_style() {}
    virtual void reset() = 0;

protected:
    DraftStyle current_style() const {
        return ctx_.style_provider ? ctx_.style_provider() : DraftStyle{};
    }
    Color preview_color() const { return ctx_.preview_color; }
    Canvas* canvas() const { return ctx_.canvas; }
    int allocate_working_priority() const {
        return ctx_.working_priority_allocator ? ctx_.working_priority_allocator() : 0;
    }
    void commit(DraftCommit commit_info) {
        if (ctx_.commit_callback) ctx_.commit_callback(std::move(commit_info));
    }

private:
    DraftContext ctx_;
};
```

具体草稿：

- `LineDraft` / `RectangleDraft` / `CircleDraft`：两次点击确定关键点，过程中生成淡色预览实体；
- `PolygonDraft` / `PolylineDraft`：不断追加顶点，`Enter` 提交，`Esc` 取消；
- 所有预览实体的优先级都由 `working_priority_allocator` 发放，数值极大，确保不会被历史中已有实体挡住。

当草稿决定提交时，会创建一个 `DraftCommit`：

```C++
struct DraftCommit {
    std::unique_ptr<Entity> entity;  // 首次绘制结果
    std::function<std::unique_ptr<Entity>(Canvas*, const DraftStyle&)> rebuild;
    ShapeType shape_type;
};
```

```C++
DraftContext Painter::make_draft_context() {
    DraftContext ctx;
    ctx.canvas = canvas;
    ctx.preview_color = preview_color;
    ctx.style_provider = [this]() { return current_style(); };
    ctx.commit_callback = [this](DraftCommit commit_info) {
        handle_draft_commit(std::move(commit_info));
    };
    ctx.working_priority_allocator = [this]() { return allocate_working_priority(); };
    return ctx;
}
```

`Painter` 收到提交后会：

1. 分配一个递增的绘制优先级并写入实体；
2. 把实体、优先级、形状类型和 `rebuild` lambda 推入历史栈；
3. 根据形状类型决定是否打开“形状菜单”；
4. 若用户按 `Backspace`，仅需弹出栈顶即可，实体析构时会自动从 `Canvas` 中注销。

这种架构使得添加新形状非常简单：实现一个新的 `Draft` 子类即可复用 `Painter` 的系统、菜单、历史记录和预览优先级机制。