

CG Lab Project1: OpenGL 基础绘制 实验报告

袁晨圖 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

```
1  add_rules("mode.debug", "mode.release")
2  set_languages("c99", "c++20")
3  add_requires("opengl", {system = true})
4  add_requires("glut", {system = true})
5  add_requires("glfw3", {system = true})
6  add_requires("spdlog", {system = true})
7  add_requires("magic_enum")
8
9  target("project1")
10     set_kind("binary")
11     add_files("src/basics/*.cpp")
12     add_packages("opengl")
13     add_packages("glut")
14     add_packages("spdlog")
15     add_packages("glfw3")
16     add_packages("magic_enum")
17     if is_plat("macosx") then
18         add_frameworks("Cocoa", "CoreFoundation", "IOKit")
19     end
```

Lua

代码 1 xmake.lua

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

```
1  xmake
2  xmake run project1
```

Shell

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

```
1  xmake run project1 catppuccin # default
2  xmake run project1 xterm-dark
3  xmake run project1 catppuccin-dark
```

Shell

二、实验效果

2.1 demo

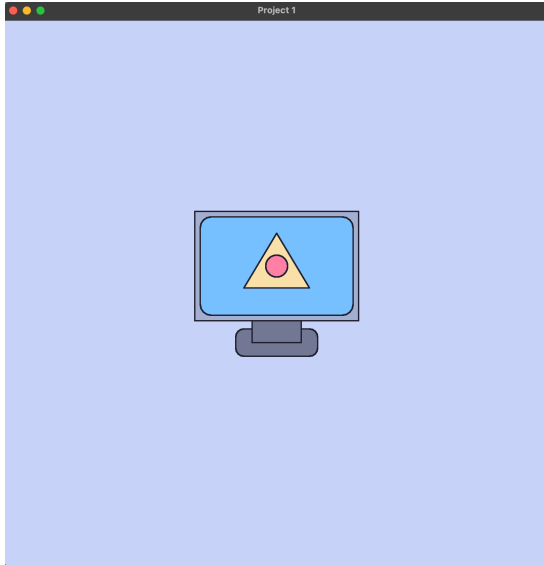


图 1 computer

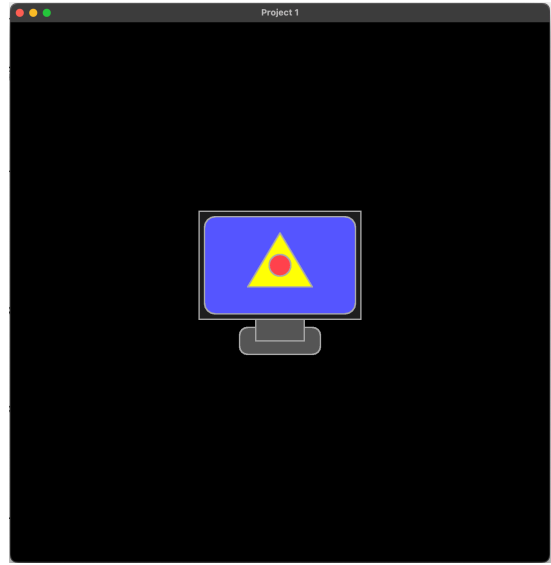


图 2 computer, theme=xterm-dark



图 3 画板画图 demo

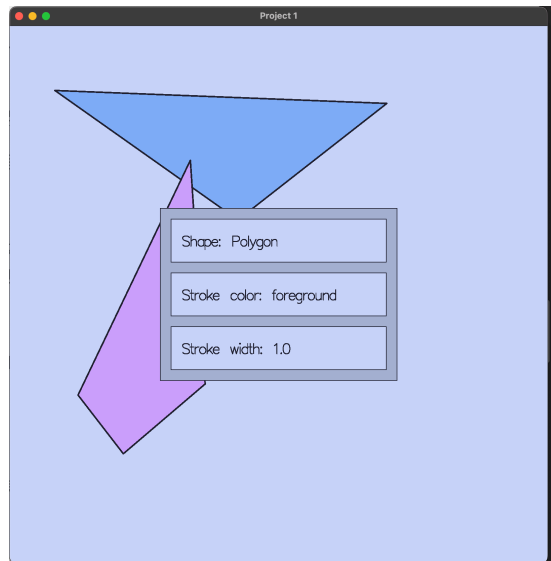


图 4 画板选项栏

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

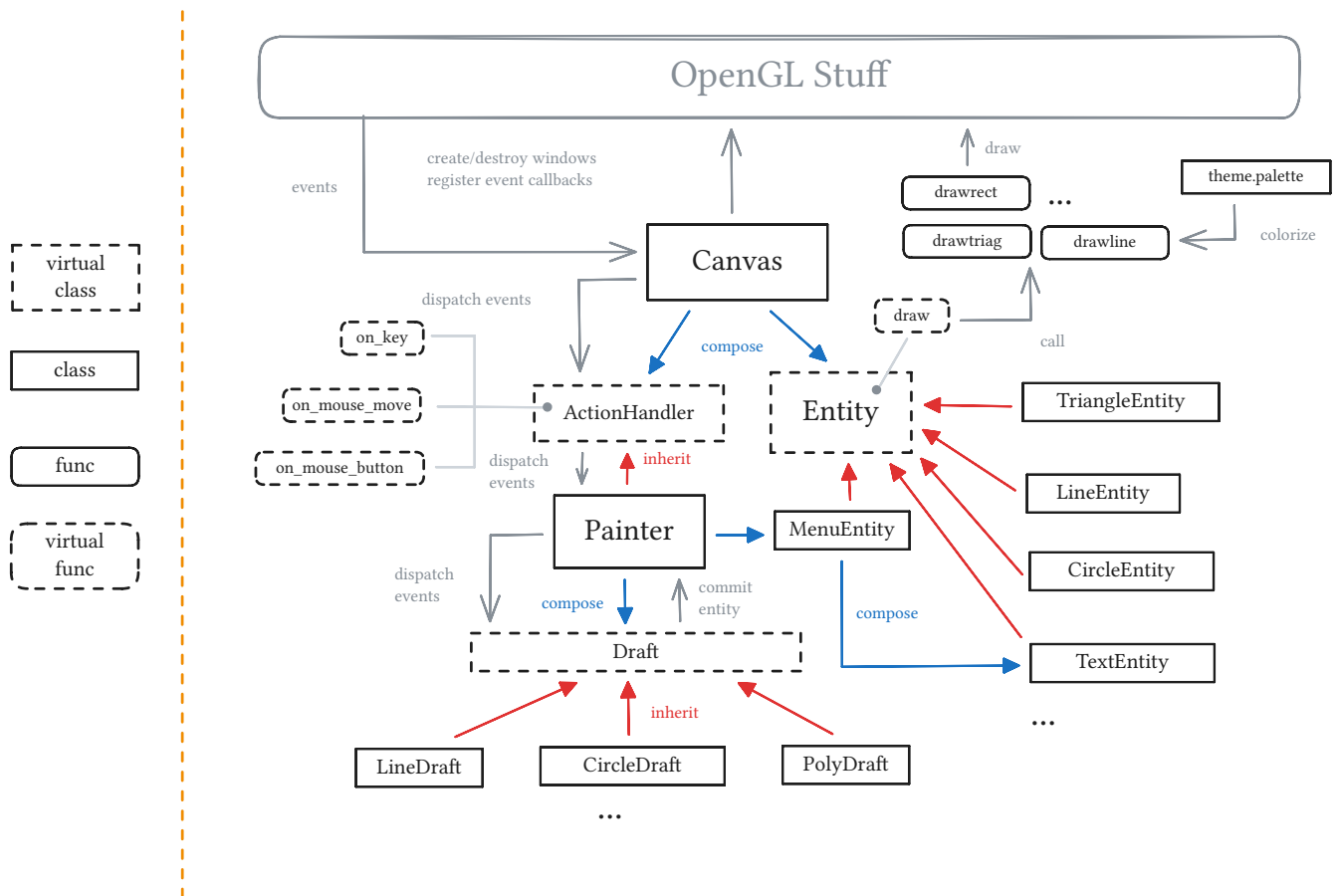


图 5 架构图

```

1 basics/
2 |— coord.hpp: 坐标表示 Vertex2d
3 |— color.hpp: 颜色表示 Color, 预设的主题调色盘 Palette
4 |— draw.hpp: 图形绘制函数 draw::line, draw::circle, ...
5 |— entity.hpp: 图形实体基类 Entity 及其派生类 Line, Rectangle, Circle, ...
6 |— canvas.hpp: Canvas 负责 GLFW 初始化及主循环
7 |— 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
8 |— painter.hpp: 画板主逻辑, 处理用户输入并调用 drafts 和 canvas
9 |— 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

```

1 struct Vertex2d {
2     GLdouble x, y;
3     Vertex2d(GLdouble x, GLdouble y) : x(x), y(y) {}
4     template <typename T> Vertex2d(std::initializer_list<T> init); // supports {x, y}

```

C++

```
5 };
```

```
1 struct Color {
2     GLdouble red, green, blue, alpha{1.0};
3     Color(GLubyte r, GLubyte g, GLubyte b, GLubyte a = 255)
4         : red(r / 255.0), green(g / 255.0), blue(b / 255.0), alpha(a / 255.0) {}
5     Color(GLdouble r, GLdouble g, GLdouble b, GLdouble a = 1.0)
6         : red(r), green(g), blue(b), alpha(a) {}
7     Color(HexColor hex);
8     Color(const std::string_view color_name);
9     Color(const char* color_name) : Color(std::string_view{color_name}) {}
10 };
```

C++

这里传入 string_view 的构造函数可以根据调色盘返回颜色

```
1 enum class ColorID {
2     WHITE = 0, // background
3     RED = 1,
4     GREEN = 2,
5     ...
6 };
7
8 struct Palette {
9     unsigned int palette[18];
10 };
11
12 inline Color::Color(const std::string_view color_name) {
13     std::string name{color_name};
14     // convert to uppercase because magic_enum expects enum names like "WHITE", "BLACK"
15     std::transform(
16         name.begin(), name.end(), name.begin(), [](unsigned char c) { return
17             std::toupper(c); });
18     auto color_enum = magic_enum::enum_cast<themes::ColorID>(name);
19     if (color_enum.has_value()) {
20         auto color = color_enum.value();
21         unsigned int hex = current_theme.palette[static_cast<int>(color)];
22         *this = Color(HexColor{hex});
23     } else {
24         throw std::runtime_error(fmt::format("Unknown color name: {}", color_name));
25     }
26 }
```

C++

因此，在代码中改变 current_theme 会立刻改变主题，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

```
1 inline struct GlfwContext {
2     GlfwContext() {
3         if (!glfwInit()) {
```

C++

```

4         throw std::runtime_error("glfw init failed");
5     }
6 }
7 ~GlfwContext() { glfwTerminate(); }
8 } glfw_context;

```

通过 Canvas 类管理窗口和 OpenGL 上下文

```

1  struct CanvasParameters {
2      std::string title;
3      struct {
4          int width{800}, height{800};
5      } display_size;
6      Color background{"background"};
7      struct {
8          GLdouble left{-5}, right{5}, bottom{-5}, top{5};
9          GLdouble zNear{5}, zFar{15};
10     } projection;
11     struct {
12         GLdouble eyeX, eyeY, eyeZ;
13         GLdouble centerX, centerY, centerZ;
14         GLdouble upX, upY, upZ;
15     } view_point;
16 };
17
18 struct Canvas {
19     const CanvasParameters params;
20     GLFWwindow* window;
21     ...
22     Canvas(const CanvasParameters& params = CanvasParameters()) : params(params) {}
23
24     void init() {
25         auto& [title, display_size, bg, proj, view] = this->params;
26         glfwMakeContextCurrent(this->window);
27         glClearColor(bg.red, bg.green, bg.blue, bg.alpha);
28         glMatrixMode(GL_PROJECTION);
29         glLoadIdentity();
30         glOrtho(proj.left, proj.right, proj.bottom, proj.top, proj.zNear, proj.zFar);
31         glMatrixMode(GL_MODELVIEW);
32         glLoadIdentity();
33         gluLookAt(
34             view.eyeX, view.eyeY, view.eyeZ, view.centerX, view.centerY, view.centerZ,
35             view.upX,
36             view.upY, view.upZ);
37     }
38
39     void window_init() {

```

```

40     auto [w, h] = this->params.display_size;
41     spdlog::info("creating GLFW window width: {}, height: {}", w, h);
42     this->window = glfwCreateWindow(w, h, this->params.title.c_str(), NULL, NULL);
43     if (!window) {
44         throw std::runtime_error("glfw create window failed");
45     }
46 }
47
48 ...
49
50 // 主循环
51 void spin() {
52     // 初始化窗口
53     this->window_init();
54     this->init();
55     if (this->action_handler) this->attach_handler(this->action_handler.get());
56     while (!glfwWindowShouldClose(this->window)) {
57         glClear(GL_COLOR_BUFFER_BIT);
58
59         // 根据优先级给所有实体排序, 然后依次渲染
60         std::vector<Entity*> sorted_entities(entities.begin(), entities.end());
61         std::sort(sorted_entities.begin(), sorted_entities.end(), [this](Entity* a,
        Entity* b) {
62             return this->entity_attributes[a].priority < this->
        entity_attributes[b].priority;
63         });
64         for (auto entity : sorted_entities) {
65             entity->draw();
66         }
67
68         glfwSwapBuffers(window);
69         glfwPollEvents();
70     }
71     glfwDestroyWindow(this->window);
72     this->window = nullptr;
73 }
74 };

```

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, 所以代码里都是用 rect 模拟线段的

```

1 // namespace draw
2 inline constexpr double kLineWidthScale = 0.03;
3
4 inline void rect_filled(const opengl::Vertex2d& center, double w, double h, opengl::Color
color);

```

```
5 inline void
6 circle_filled(const opengl::Vertex2d& center, double radius, opengl::Color color, int
  segments);
7
8 inline void line(Vertex2d start, Vertex2d end, Color color, double width = 1.0) {
9     double scaled_width = width * kLineWidthScale;
10    if (scaled_width <= 0.0) {
11        return;
12    }
13
14    double dx = end.x - start.x;
15    double dy = end.y - start.y;
16    double length = std::hypot(dx, dy);
17    if (length <= std::numeric_limits<double>::epsilon()) {
18        rect_filled(start, scaled_width, scaled_width, color);
19        return;
20    }
21
22    Vertex2d center{(start.x + end.x) * 0.5, (start.y + end.y) * 0.5};
23    double angle_deg = std::atan2(dy, dx) * 180.0 / std::numbers::pi_v<double>;
24
25    glPushMatrix();
26    glTranslated(center.x, center.y, 0.0);
27    glRotated(angle_deg, 0.0, 0.0, 1.0);
28    rect_filled(Vertex2d{0.0, 0.0}, length, scaled_width, color);
29    glPopMatrix();
30 }
```

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

```
1 // namespace draw
2 inline void
3 draw_polyline(const std::vector<Vertex2d>& points, bool closed, Color color, double width)
  {
4     if (points.size() < 2) {
5         return;
6     }
7
8     double joint_radius = 0.5 * width * kLineWidthScale;
9     if (joint_radius > 0.0) {
10        constexpr int kJointSegments = 18;
11        for (const auto& p : points) {
12            circle_filled(p, joint_radius, color, kJointSegments);
```

```
13     }
14 }
15
16 for (std::size_t i = 1; i < points.size(); ++i) {
17     line(points[i - 1], points[i], color, width);
18 }
19 if (closed) {
20     line(points.back(), points.front(), color, width);
21 }
22 }
```

用多个线段模拟画圆/画弧

```
1 // 画圆（描边） C++
2 // center: 圆心, radius: 半径, color: 颜色, segments: 分段数（越大越圆）
3 inline void circle_outline(
4     const opengl::Vertex2d& center, double radius, opengl::Color color, int segments = 64,
5     double line_stroke = 1.0) {
6     if (radius <= 0.0) {
7         return;
8     }
9     int segs = std::max(3, segments);
10    std::vector<Vertex2d> points;
11    points.reserve(segs);
12    for (int i = 0; i < segs; ++i) {
13        double angle = 2.0 * std::numbers::pi_v<double> * static_cast<double>(i) / segs;
14        detail::append_point(
15            points,
16            Vertex2d{center.x + radius * std::cos(angle), center.y + radius *
17                    std::sin(angle)});
18    }
19    detail::draw_polyline(points, true, color, line_stroke);
20 }
```

文字这里比较 tricky，通过多次偏移画同一个字符来加粗字体

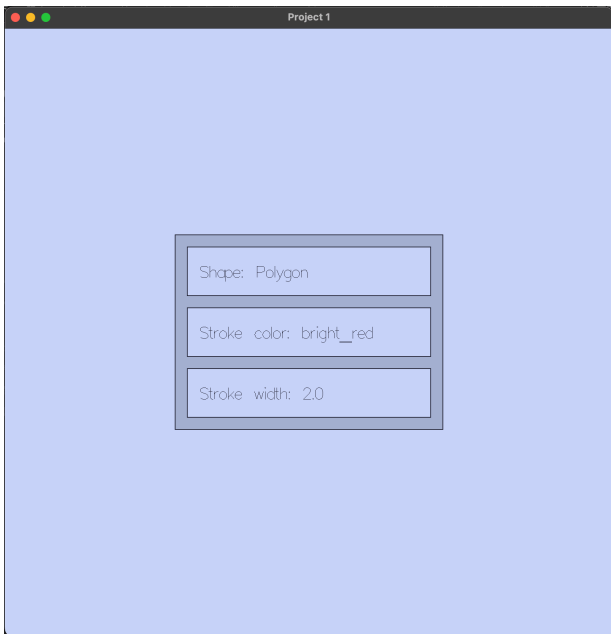


图 6 加粗前

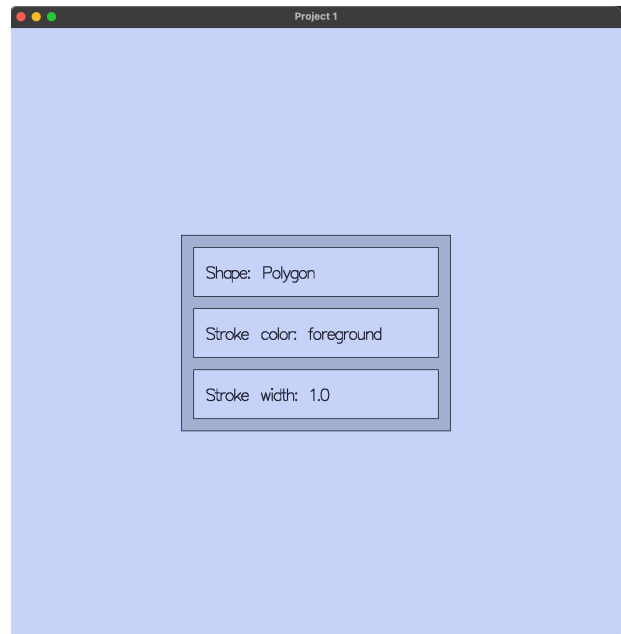


图 7 加粗后

```

1  inline void text(
2      const opengl::Vertex2d& origin, const std::string& content, opengl::Color color,
3      double scale = 1.0) {
4      if (content.empty()) {
5          return;
6      }
7      double normalized_scale = std::max(0.001, 0.0027 * scale);
8      float line_width = std::max(1.0f, static_cast<float>(2.0 * scale));
9      constexpr double kStrokeFontHeight = 110;
10     glPushMatrix();
11     glColor3d(color.red, color.green, color.blue);
12     glTranslated(origin.x, origin.y - normalized_scale * kStrokeFontHeight * 0.5, 0.0);
13     glScaled(normalized_scale, normalized_scale, 1.0);
14     // set a thicker line width for stroke font rendering
15     glLineWidth(line_width);
16     // draw multiple passes with symmetric offsets to thicken strokes without shifting
17     // position
18     const double offset = 3.0; // in stroke-font units (before scaling)
19     const double step = 0.3;
20     auto offsets =
21         std::views::iota(-static_cast<int>(offset / step), static_cast<int>(offset / step)
22             + 1) |
23         std::views::transform([step](int idx) { return static_cast<double>(idx) *
24             step; });
25     for (double ox : offsets) {
26         for (double oy : offsets) {
27             glPushMatrix();
28             glTranslated(ox, oy, 0.0);
29             for (unsigned char ch : content) {
30                 glutStrokeCharacter(GLUT_STROKE_ROMAN, ch);
31             }
32             glPopMatrix();
33         }
34     }
35     glPopMatrix();
36 }

```

```

28     }
29     glPopMatrix();
30 }
31 }
32 // restore default line width
33 glLineWidth(1.0f);
34 glPopMatrix();
35 }

```

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

```

1 struct Entity {
2     virtual void draw() const = 0;
3     virtual std::string repr() const = 0;
4     virtual ~Entity();
5     Canvas* container;
6     Entity(Canvas* canvas);
7     auto attribute() const -> EntityAttribute;
8     void set_priority(int) const;
9 };

```

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

```

1 struct Line {
2     Vertex2d start;
3     Vertex2d end;
4     Color color{"foreground"};
5     double stroke{1.0};
6     using EntityType = LineEntity;
7 };
8
9 struct LineEntity : Entity {
10     Line config;
11     LineEntity(Canvas* canvas, Line config) : Entity(canvas), config(config) {}
12     void draw() const override {
13         draw::line(config.start, config.end, config.color, config.stroke);
14     }
15     std::string repr() const override {
16         return fmt::format(
17             "Line(start={}, end={}, color={}, stroke={})", config.start, config.end,
18             config.color,
19             config.stroke);
19     }

```

```
20 };
```

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

```
1 inline Entity::Entity(Canvas* canvas) : container(canvas) { this->container->add_entity(this); }
2 inline Entity::~~Entity() {
3     if (this->container) this->container->delete_entity(this);
4     spdlog::info("entity {} destructed", (void*)this);
5 }
```

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 指针清了就行，唯一需要担心的是 Entity 内存泄露，但用 std::unique_ptr 可以解决这一问题。

由于配置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

```
1 auto Canvas::draw(auto config) {
2     auto entity = std::make_unique<typename decltype(config)::EntityType>(this, config);
3     spdlog::info("draw: {}, id={}, canvas={}", entity->repr(), (void*)entity.get(), (void*)this);
4     return entity;
5 }
```

使用 Config 结构体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：

```
1 struct Computer {
2     std::set<std::unique_ptr<Entity>> parts;
3     Computer(Canvas* canvas, double center_x = 0.0, double center_y = 0.0) {
4         auto screen_x = center_x, screen_y = center_y + 0.5;
5         double screen_w = 3, screen_h = 2;
6         double base_delta = 0.4;
7         double base_x = screen_x, base_y = screen_y - screen_h / 2 - base_delta;
8         double base_w = 1.5, base_h = 0.5;
9         double margin = 0.2;
10        auto base_color = mix("foreground", "background", 0.5);
11        auto margin_color = mix("foreground", "background", 0.8);
12        parts.insert(canvas->draw(
13            Rectangle{
14                .center = {base_x, base_y},
15                .width = base_w,
16                .height = base_h,
17                .corner_radius = 0.15,
18                .fill_color = base_color,
```

```

19     });
20     parts.insert(canvas->draw(
21         Rectangle{
22             .center = {base_x, screen_y - screen_h / 2},
23             .width = base_w * 0.6,
24             .height = base_delta * 2,
25             .fill_color = base_color,
26         });
27     parts.insert(canvas->draw(
28         Rectangle{
29             .center = {screen_x, screen_y},
30             .width = screen_w,
31             .height = screen_h,
32             .fill_color = margin_color,
33         });
34     parts.insert(canvas->draw(
35         Rectangle{
36             .center = {screen_x, screen_y},
37             .width = screen_w - margin,
38             .height = screen_h - margin,
39             .corner_radius = 0.2,
40             .fill_color = "bright_blue",
41         });
42     double triangle_l1 = 0.6, triangle_l2 = 0.4;
43     parts.insert(canvas->draw(
44         Triangle{
45             .p1 = {screen_x - triangle_l1, screen_y - triangle_l2},
46             .p2 = {screen_x + triangle_l1, screen_y - triangle_l2},
47             .p3 = {screen_x, screen_y + triangle_l2 * 1.5},
48             .fill_color = "bright_yellow",
49         });
50     double circle_r = 0.2;
51     parts.insert(canvas->draw(
52         Circle{
53             .center = {screen_x, screen_y},
54             .radius = circle_r,
55             .fill_color = "bright_red",
56         });
57 }
58 };

```

以典型的 Rectangle 为例:

```

1  struct Rectangle {
2      Vertex2d center;
3      double width;
4      double height;
5      std::optional<double> corner_radius{};

```

C++

```

6   Color color{"foreground"}; // stroke color
7   std::optional<Color> fill_color = Color{"foreground"};
8   double stroke{1.0};
9   using EntityType = RectangleEntity;
10 };

```

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。

```

1 struct ActionHandler {
2     virtual void on_key(int key, int action) = 0;
3     virtual void on_mouse_button(int button, int action, int mods) = 0;
4     virtual void on_mouse_move(double xpos, double ypos) = 0;
5     void attach(Canvas* canvas);
6 };

```

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

```

1 void Canvas::attach_handler(ActionHandler* handler) {
2     auto key_callback = [](GLFWwindow* window, int key, int scancode, int action, int
    mods) {
3         ActionHandler* self =
4             static_cast<ActionHandler*>(glfwGetWindowUserPointer(window));
5         self->on_key(key, action);
6     };
7     auto mouse_button_callback = [](GLFWwindow* window, int button, int action, int mods)
    {
8         ActionHandler* self =
9             static_cast<ActionHandler*>(glfwGetWindowUserPointer(window));
10        self->on_mouse_button(button, action, mods);
11    };
12    auto mouse_move_callback = [](GLFWwindow* window, double xpos, double ypos) {
13        ActionHandler* self =
14            static_cast<ActionHandler*>(glfwGetWindowUserPointer(window));
15        self->on_mouse_move(xpos, ypos);
16    };
17    glfwSetWindowUserPointer(this->window, handler);
18    glfwSetKeyCallback(this->window, key_callback);
19    glfwSetMouseButtonCallback(this->window, mouse_button_callback);
20    glfwSetCursorPosCallback(this->window, mouse_move_callback);
21    handler->attach(this);
22 }

```

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:

```
1 Vertex2d cursor_to_world(double xpos, double ypos) const {
2     double nx = xpos / canvas->params.display_size.width;
3     double ny = ypos / canvas->params.display_size.height;
4     const auto& proj = canvas->params.projection;
5     double world_x = proj.left + nx * (proj.right - proj.left);
6     double world_y = proj.top - ny * (proj.top - proj.bottom);
7     return Vertex2d{world_x, world_y};
8 }
```

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。

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。

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。

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`：

```
1 struct DraftCommit {  
2     std::unique_ptr<Entity> entity; // 首次绘制结果  
3     std::function<std::unique_ptr<Entity>(Canvas*, const DraftStyle&)> rebuild;  
4     ShapeType shape_type;  
5 };
```

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。