可视计算与交互概论 Tutorial for Lab Project

Lab 使用指南

这里是一个有关 Lab 框架的简单介绍，希望能够帮助大家更好地完成大作业。

编译

我们的 Lab 框架使用 **xmake** 进行编译。在 Lab 框架的基础上进行代码开发时，我们也推荐使用 xmake 来管理你的项目。想必在完成之前 Lab 的过程中，你也已经发现，xmake 根据当前目录中的 `xmake.lua` 文件对项目进行编译。虽然每个 Lab 要求完成的任务并不相同，但是它们的 `xmake.lua` 非常相似。通过理解这其中中每行命令的作用，相信你也可以写出自己的 `xmake.lua` 用于配制自己的项目。

xmake 命令简介

在这里我们将根据之前几次 Lab 的 `xmake.lua` 来对 xmake 的命令做一个简单的介绍。由于篇幅原因，显然没有办法覆盖 xmake 的所有特性，如果在项目中希望使用 xmake 更多的特性，欢迎阅读参考资料（**xmake官网**、**知乎教程**）或自行在网上搜索进行更深入的了解。

首先，在每个 `xmake.lua` 的开头都会有这样几行命令来做一些设置。

```
set_project("VCX-Labs")
set_version("2.0.0-alpha.1")
set_xmakever("2.6.9")
set_languages("cxx20")

add_rules("mode.debug", "mode.release", "mode.profile")
```

第一行我们通过 `set_project` 语句指定了整个项目的名字为 `VCX-Labs`；第二行我们通过 `set_version` 语句指定了项目的版本为 `2.0.0-alpha.1`；第三行我们通过 `set_xmakever` 语句设置了 xmake 版本为 `2.6.9`，当我们用于运行的 xmake 版本低于这个版本时，xmake 就会报错；第四行我们通过 `set_languages` 设置语言版本，在这里我们规定我们使用 c++20 的标准。

最后一行 `add_rules` 语句声明了三个 mode：debug 、 release 和 profile（默认 release）。debug 模式会关闭编译器优化并保留程序中的调试信息，而 release 模式会加入更激进的编译器优化，profile 模式则通常用于性能分析。这一行并不是必须的，如果不写这一语句，编译时实际执行的命令中将不含有任何额外的 flag。我们可以通过命令 `xmake f -m ...` 切换到我们需要的 mode，如 `xmake f -m debug` 就可以使我们从默认的 release 模式切换到 debug 模式。

接着我们会看到一系列用于引入依赖的语句。

```
add_requires("glad")
add_requires("glfw")
...
```

我们通过 `add_requires(<package>)` 来引入我们需要的依赖，这其中的 `<package>` 表示包的名称。值得注意的是，一个包在 xmake-xrepo 中的名字可能会跟想象中不同，这需要到 **xrepo** 网站查证（如 **macOS 可用包**、**windows 可用包**），或者在本地命令行运行 `xrepo search <package>` 来进行查证。当然 `add_requires` 语句仅仅帮我们引入了我们需要的依赖，要将这些第三方库导入到对应的 target 中，我们可以在后面会看到还需要使用 `add_packages` 语句。

最后我们会看到若干 target 语句块。

```
1   target("<target-name>")
2       set_kind("binary")
3       add_deps("lab-common")
4       add_packages("<package-name>", {public = true})
5       add_headerfiles("<header-dir>/*.h")
6       add_headerfiles("<header-dir>/*.hpp")
7       add_includedirs("<include-dir>")
8       add_files      ("<file-dir>/*.cpp")
9   ...
```

第一行会声明一个名为 `<target-name>` 的 taregt。在 xmake 中，所有的构建目标都用 target 来表示。如无额外声明，target 的名字将被默认作为库/可执行文件的名字的主要部分。例如，windows 上 binary 类型的 target helloworld 其输出文件名为 helloworld.exe，linux 上 static 类型的 target helloworld 其输出文件名为 libhelloworld.a。声明 target 之后进入该 target 的作用域，直到声明另一个 target 或者显式调用 `target_end` 之前，所有语句都是针对该target生效的。

第二行的 `set_kind` 语句限定 target 的类型为 binary。xmake 中 target 类型一共有5种：binary（默认），static，shared，headeronly，phony，分别对应可执行文件、静态库、动态链接库、纯头文件、伪 target。phony 类型的 target 仅用于添加一些可继承的 flag、协调 target 编译顺序，不会执行编译链接操作。

第三行的 `add_deps` 语句通常用于构建多个 target 之间的依赖关系。如我们这里的 `add_deps("lab-common")`，在实际编译过程中就会先链接 `lab-common`，然后再链接当前 target，并在当前 target 链接时自动链接到 liblab-common.a 或 liblab-common.dylib（取决于平台）。

> 由于 `lab-common` 中提供了许多通用的工具和接口，你会发现几乎每一个 lab 都会与之建立依赖。当你在开发自己的项目时，我们也非常推荐你使用 `lab-common`。

第四行的 `add_packages` 语句在之前提到过，可以用于将需要的第三方库导入到我们的 target 中。如果希望之后调用了 `add_deps("<target-name>")` 的其他 target 也能自动建立这样的依赖关系的话，可以通过加上 `{public = true}` 的命令来要求依赖的继承。

第五行和第六行的 `add_headerfiles` 语句用于为当前的 target 添加相应的头文件。第七行的 `add_includedirs` 语句用于为当前 target 添加搜索头文件的根目录。

第八行的 `add_files` 语句为当前的 target 加入了源文件。通过 `add_files` 加入的所有源文件都会被编译为中间文件，然后链接到目标文件中。另外，xmake 支持通配符匹配添加文件，借助这一点我们可以方便地添加我们需要的文件。

编译自己的项目

我们可以在 lab 的 `xmake.lua` 末尾加上自己的 target，如：

```
target("myproject")
    set_kind("binary")
    add_deps("lab-common")
    add_packages("<package-name>")
    add_headerfiles("<header-dir>/*.h")
    add_headerfiles("<header-dir>/*.hpp")
    add_files      ("<file-dir>/*.cpp")
```

这样当在命令行中运行 `xmake` 时，xmake 会将我们自己的项目编译为 myproject 或 myproject.exe（取决于平台）并存放在 `build` 文件夹对应的路径中（路径由编译时的 `plat`、`arch`、`mode`、`kind` 等参数决定）。这之后，在命令行中输入 `xmake run myproject` 就可以运行编译得到的可执行文件了。

当然，这只是一个最简单的使用 xmake 编译项目的实现。你也完全可以从头开始书写自己的 `xmake.lua` 进行编译，在其中加入更多 xmake 的功能。最后，我们相信，多多尝试，多多参考 lab 的实现，多多查看 xmake 的文档并利用网络搜索，你一定很快就会掌握 xmake 这个强大又便捷的工具！

代码框架

我们的代码通过 **imgui** 搭建 UI 界面，使用 **OpenGL** 提供 3D 的视觉效果。虽然代码复杂，但是由于经过了精心的设计和封装，在代码框架上进一步开发还是相对容易的。我们在这里将从之前几次 Lab 的代码入手，展示它们共同的代码逻辑，希望能够给你的项目开发带来灵感。

启动

你应该已经注意到了，每次 Lab 的核心代码都会出现在 `src/VCX/Labs/<lab-name>` 文件夹中，当我们在命令行中输入 `xmake run lab...` 后程序就会执行这个文件夹中 `main.cpp` 里的 `main()`。查看这个 `main()` 我们很容易发现，它实际运行的是 `Engine::RunApp()` 函数。~~接下来的过程，令当年稚嫩的笔者颇为震撼，是一个很好的锻炼c++能力的过程。~~ 在这里，我们将简单描述它的调用逻辑，如果对于这其中的细节不感兴趣，只希望了解如何在这之上进行开发，也可以直接快进到 **下一节**。

首先 `RunApp()` 函数会执行以下三个函数：

```
template<typename TApp>
    requires std::is_base_of_v<IApp, TApp> && std::is_constructible_v<TApp>
int RunApp(AppContextOptions const & options) {
    Internal::RunApp_Init(options);
    Internal::RunApp_Main(TApp());
    Internal::RunApp_Shutdown();
    return 0;
}
```

这其中 `Internal::RunApp_Init()` 和 `Internal::RunApp_Shutdown()` 负责启动前的初始化和运行结束后的关闭，这两部分如果不是对 Lab 代码希望进行比较彻底地修改可以不细读。在 `Internal::RunApp_Main()` 函数中我们要完成如下任务：

```
void RunApp_Main(IApp && app) {
    RunApp_InitGLFWWindowCallbacks(app);
    glfwShowWindow(g_glfwWindow);
    while (! glfwWindowShouldClose(g_glfwWindow)) {
        RunApp_Frame(app);
        glfwPollEvents();
    }
}
```

这里的大部分代码主要负责处理 OpenGL 的各类任务，在 `while` 循环内的 `RunApp_Frame()` 函数是我们核心代码运行所在，而在 `RunAPP_Frame()` 函数中，我们首先完成了一些 imgui 和 openGL 的处理，随后调用 `app.OnFrame()` 函数。注意到，这个函数在基类中是一个纯虚函数，在实现不同 Lab 的 `App` 类别时对其会有不同的实现，经过一番索引后我们可以发现最终它们都会执行 `UI::Setup()` 函数。

在 `UI::Setup()` 函数中，我们将通过 `setupSideWindow()` 中设置我们侧边栏的布局，在 `setupMainWindow()` 中设置我们主窗口的布局。而在这两个函数内部，我们又将调用大量的 imgui 代码用于 UI 的搭建，但是它们提供的是一个统一的基础结构。值得注意的是以下三个函数：

1. 在 `setupSideWindow()` 中，我们通过 `cases[caseId].get().OnSetupPropsUI();` 实现了对于不同的 case 情况我们可以设置不同的 sidebar。
2. 在 `setupMainWindow()` 中，我们通过 `casei.OnRender({ std::uint32_t(_layout.ContentChildSize.x), std::uint32_t(_layout.ContentChildSize.y) })` 返回了我们希望在主窗口上展示的结果。
3. 在 `setupMainWindow()` 中，我们通过 `casei.OnProcessInput({ ImGui::GetIO().MousePos.x - cornerPos.x, ImGui::GetIO().MousePos.y - cornerPos.y });` 处理了不同 case 情况对于外界交互的响应。

开发

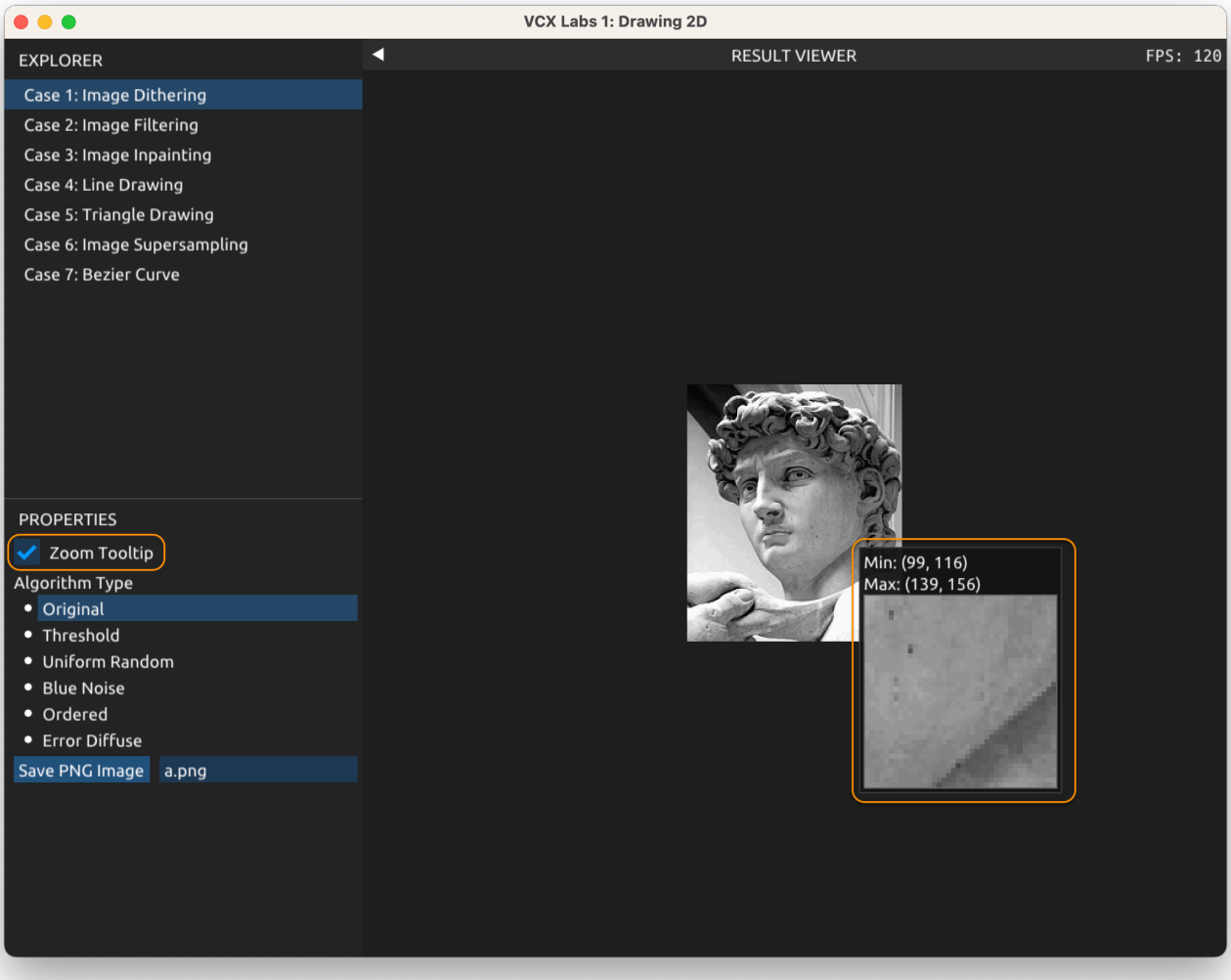
结合我们刚才的观察，我们就不难发现，我们 Lab 的开发方式都是类似的，首先定义一个新的 App 类，在 App 内我们定义自己的 Case 类，对于每类 Case 我们需要重载 `OnSetupPropsUI()`、`OnRender()` 和 `OnProcessInput()` 这三个函数，每一帧都会调用这三个函数，来实现我们希望在这个 Case 中做到的效果。

侧边栏的开发

侧边栏的开发通过 `onSetupPropsUI()` 函数实现。在这里，我们通常通过调用 imgui 的接口来创建我们需要的侧边栏按钮，并绑定变量实现一个可交互的效果。例如以下语句

```
ImGui::Checkbox("Zoom Tooltip", &_enableZoom);
```

我们将在侧边栏中创建一个 imgui 的 checkbox 对象，它显示的文本将是 `Zoom Tooltip`，并且它是否被勾选的状态会绑定给变量 `_enableZoom`。在当我们查看一些二维图片时，就可以通过这个按钮决定是否要使用放大镜的功能，这个功能对于完成了前几个 lab 的你来说一定并不陌生。



除此之外，imgui 还为我们提供了丰富多样的创建各类交互按钮的接口，在之前的 lab 中也有许多实际应用的例子，在进行项目的开发时可以多多借鉴，选择需要的接口进行使用。如果想实现之前 lab 中没有实现的效果，也可以查阅 imgui 官方代码 或在网上进行搜索。

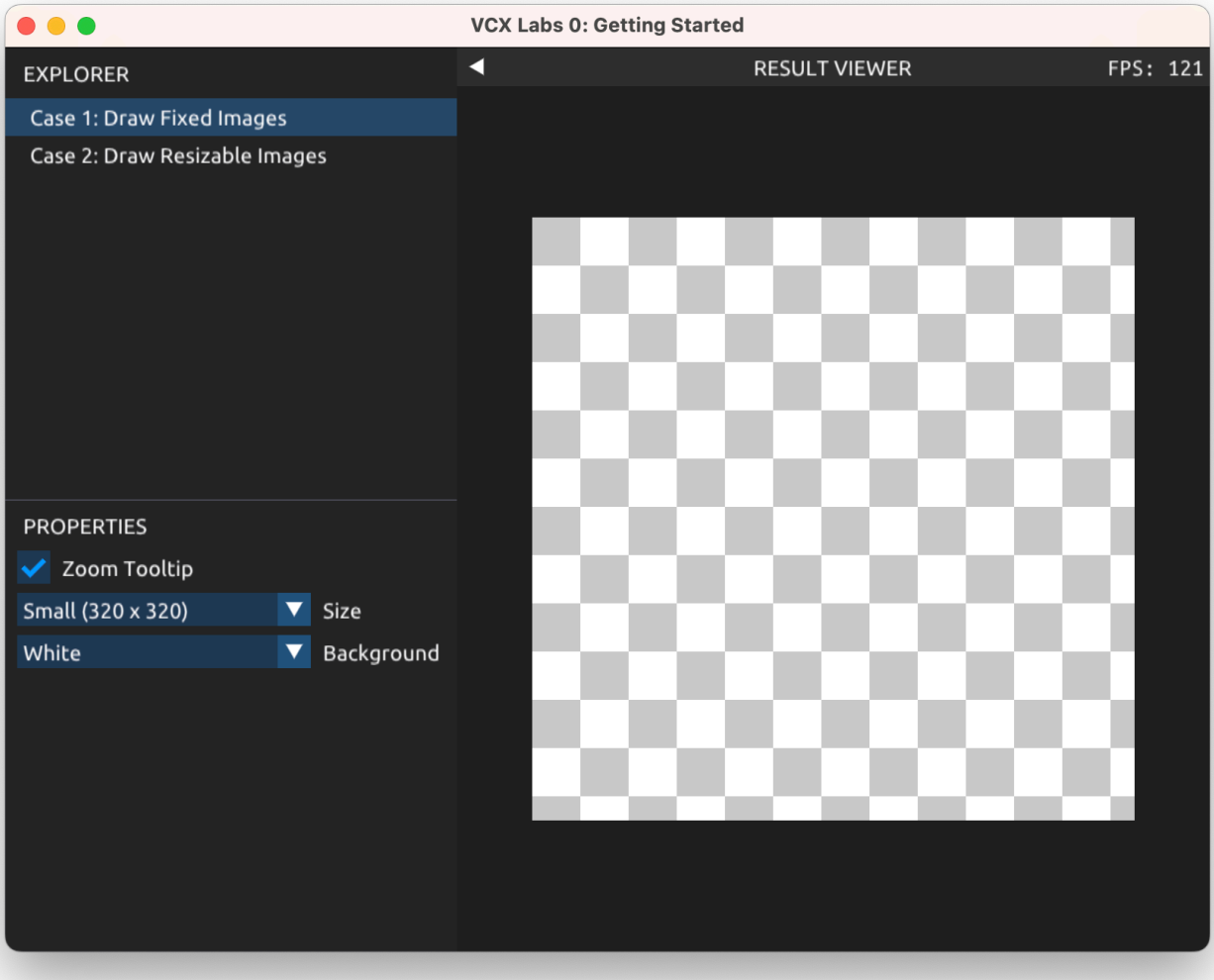
主窗口的开发

主窗口的开发通过 `OnRender()` 函数实现。每次调用 `OnRender()` 函数，它将返回一个 `Common::CaseRenderResult` 类型的结果。

当我们希望在主窗口呈现的内容是一个 2D 的内容时，我们可以构造一个 `Common::ImageRGB` 的 `image`，在 `image` 的对应像素染上对应的颜色，最后作为结果返还，我们就可以可视化我们需要的结果。例如，我们将 Lab0 中位于 `src/VCX/Labs/0-GettingStarted` 中的 `CaseFixed.cpp` 文件的 `CaseFixed::OnRender()` 修改为

```cpp
Common::CaseRenderResult CaseFixed::OnRender(std::pair<std::uint32_t, std::uint32_t> const desiredSize) {
    // 调用 CreateCheckboardImageRGB 来创建一个 400x400 的棋盘格的 image
    Common::ImageRGB image = Common::CreateCheckboardImageRGB(400, 400);
    // 将 image 更新到 _textures[0] 上
    _textures[0].Update(image);
    return Common::CaseRenderResult {
        .Fixed     = true,          // 代表生成的 2D 图像是固定的，即不会随着窗口大小的变化而拉伸的
        .Image     = _textures[0], // 将 _textures[0] 绑定到返回的结果上
        .ImageSize = {400, 400},    // 设置图像的大小为 400x400
    };
}
```

经过这样的修改之后，lab0 中的 CaseFixed 将一直展示一个 400x400 的棋盘格

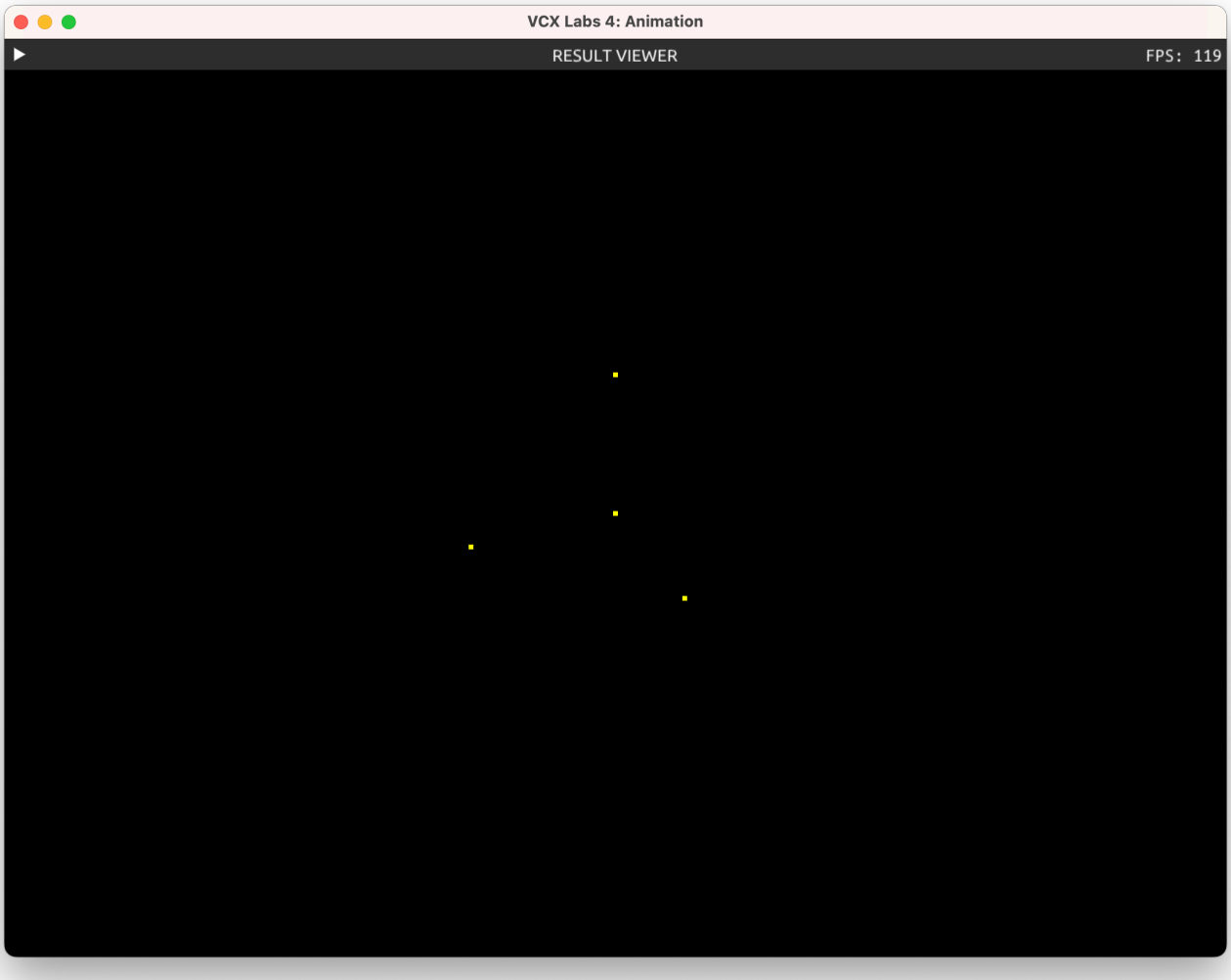


如果你希望可视化出别的不一样的效果，也可以通过同样的逻辑来完成。

主窗口除了可以提供一个 2D 内容的呈现之外，它也基于 OpenGL 提供三维可视化的展示。在这里，我们以 `src/VCX/Labs/4-Animation` 文件夹中的 `CaseMassSpring.cpp` 为例，将其中的 `CaseMassSpring::OnRender()` 函数修改为

```cpp
Common::CaseRenderResult CaseMassSpring::OnRender(std::pair<std::uint32_t, std::uint32_t> const desiredSize) {
    // 设置四个点的位置
    std::vector<glm::vec3> positions{glm::vec3(0, 0, 0), glm::vec3(1, 0, 0), glm::vec3(0, 1, 0), glm::vec3(0, 0, 1)};
    // 将这四个点的位置更新到 _particlesItem 中
    _particlesItem.UpdateVertexBuffer("position", Engine::make_span_bytes<glm::vec3>(positions));
    // 将 _frame 调整为 desiredSize
    _frame.Resize(desiredSize);
    // 调整相机位置
    _cameraManager.Update(_camera);
    _program.GetUniforms().SetByName("u_Projection", _camera.GetProjectionMatrix((float(desiredSize.first) / desiredSize.second)));
    _program.GetUniforms().SetByName("u_View"      , _camera.GetViewMatrix());
    gl_using(_frame);
    // 设置 OpenGL 中点的大小
    glPointSize(4.f);
    // 设置点的颜色，glm::vec3(1., 1., 0.) 即黄色
    _program.GetUniforms().SetByName("u_Color", glm::vec3(1., 1., 0.));
    _particlesItem.Draw({ _program.Use() });
    // 恢复 OpenGL 中点的大小为默认值
    glPointSize(1.f);

    return Common::CaseRenderResult {
        .Fixed     = false,
        .Flipped   = true,
        .Image     = _frame.GetColorAttachment(), // 获取 OpenGL 可视化得到的结果
        .ImageSize = desiredSize,
    };
}
```

这样我们的 lab4 的 CaseMassSpring 就会可视化四个黄色的点，效果如下图所示。

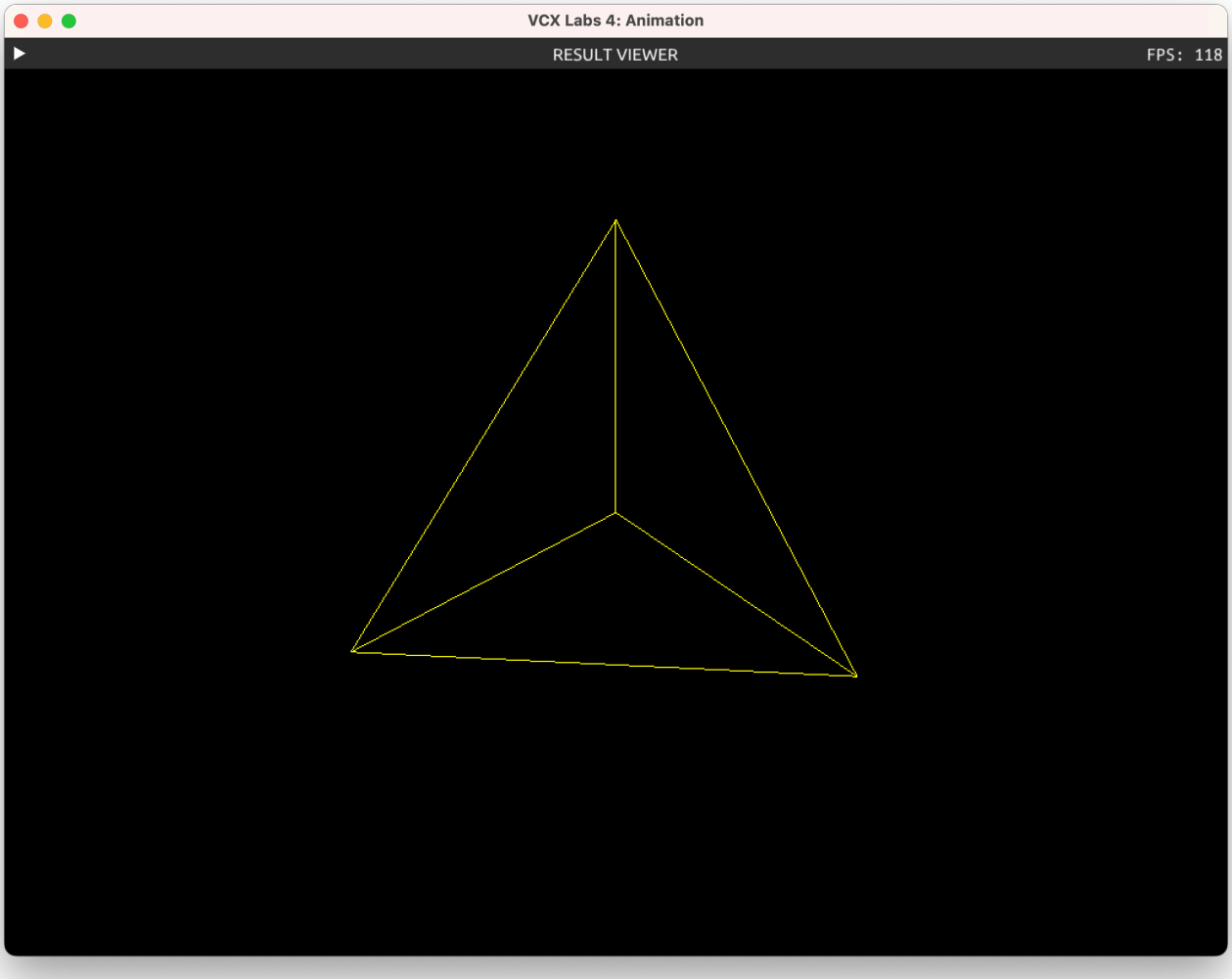


当然，我们不仅仅可以做到点的三维呈现，我们同样可以可视化线段、三角形等图形学中常见的基本元素。但是它们相比于点的呈现要更复杂一些。注意到我们这里使用了 `_particlesItem` 来加载点的位置，这是一个 `Engine::GL::UniqueRenderItem` 类的对象，如果需要可视化线段、三角形等元素，我们需要创建一个 `Engine::GL::UniqueIndexedRenderItem` 类的对象，顾名思义，这些对象还需要你提供 Indices，来说明点与点之间相连的关系。例如，我将 `CaseMassSpring::OnRender()` 函数修改为如下形式：

```
Common::CaseRenderResult CaseMassSpring::OnRender(std::pair<std::uint32_t, std::uint32_t> const desiredSize) {
    std::vector<glm::vec3> positions{glm::vec3(0, 0, 0), glm::vec3(1, 0, 0), glm::vec3(0, 1, 0), glm::vec3(0, 0, 1)};
    _springsItem.UpdateVertexBuffer("position", Engine::make_span_bytes<glm::vec3>(positions));
    // 通过 indices，_springsItem 可以得到绘制 Lines 时连接点的顺序。在这里，它将会两两一组进行连线，而如果初始化 _springsItem 时规定其 PrimitiveType 为三角形，那么就会每三个点一组进行连线得到三角形
    std::vector<uint32_t> indices{0, 1, 0, 2, 0, 3, 1, 2, 1, 3, 2, 3};
    _springsItem.UpdateElementBuffer(indices);

    _frame.Resize(desiredSize);

    _cameraManager.Update(_camera);

```

```
12      _program.GetUniforms().SetByName("u_Projection",
    _camera.GetProjectionMatrix((float(desiredSize.first) / desiredSize.second)));
13      _program.GetUniforms().SetByName("u_View"      , _camera.GetViewMatrix());
14
15      gl_using(_frame);
16
17      _program.GetUniforms().SetByName("u_Color", glm::vec3(1., 1., 0.));
18      _springsItem.Draw({ _program.Use() });
19
20      return Common::CaseRenderResult {
21          .Fixed     = false,
22          .Flipped   = true,
23          .Image     = _frame.GetColorAttachment(),
24          .ImageSize = desiredSize,
25      };
26  }
```

这样我们的 lab4 的 CaseMassSpring 就会可视化四个黄色的点相连得到的四面体，效果如下图所示。



了解了这些基本元素的可视化方法之后，我们就可以通过这些基本元素搭建出更加复杂的图形，做出更加丰富的可视化效果。

交互的开发

交互的开发通过 `onProcessInput()` 函数实现。Lab 的框架中也提供了一些基础的交互的处理方式，如在进行 2D 图像的可视化时，我们可以使用如下代码实现一个基本的交互

```
void OnProcessInput(ImVec2 const & pos) {
    auto         window  = ImGui::GetCurrentWindow();
    bool         hovered = false;
    bool         anyHeld = false;
    ImVec2 const delta   = ImGui::GetIO().MouseDelta;
    // 获取交互信息
    ImGui::ButtonBehavior(window->Rect(), window->GetID("##io"), &hovered, &anyHeld);
    if (! hovered) return;
    // 调整窗口大小
    if (ImGui::IsMouseDown(ImGuiMouseButton_Left) && delta.x != 0.f)
        ImGui::SetScrollX(window, window->Scroll.x - delta.x);
    if (ImGui::IsMouseDown(ImGuiMouseButton_Left) && delta.y != 0.f)
        ImGui::SetScrollY(window, window->Scroll.y - delta.y);
    // 决定是否调用放大镜功能
    if (_enableZoom && ! anyHeld && ImGui::IsItemHovered())
        Common::ImGuiHelper::ZoomTooltip(_textures[_sizeId], c_Sizes[_sizeId], pos);
}
```

而如果你使用的是 3D OpenGL 的可视化，则可以通过如下代码实现一个基本的交互

```
void OnProcessInput(ImVec2 const & pos) {
    _cameraManager.ProcessInput(_camera, pos);
}
```

通过这样的设置，你可以通过鼠标左键拖动实现对轨道相机的旋转，通过鼠标右键拖动移动相机焦点，通过鼠标滚轮径向移动相机（放大或缩小场景）

总结

总的来说，在 lab 代码的基础上进行开发是有一定的逻辑可以遵循的，但是这里介绍的方法也并不代表就是项目开发的唯一方式。我们希望提供的 lab 代码基础可以为你提供一个方便的可视化工具，从而可以让你把更多的精力用在算法的实现上，如果在这个过程中遇到了困难，也十分欢迎你及时向助教求助！

> ***温馨提示***：以上仅仅是对于 Lab 代码一个非常粗略的介绍，并不代表这就是它全部的功能，当你在完成自己的项目时，不要忘记，之前 5 个 lab 的实现才是最好的参考资料，找到其中你感兴趣的效果，仔细阅读代码理解它们是如何实现的，相信会给你自己的项目开发带来很大帮助！