第一章 绪论

1.1 研究背景

随着人类对于结构设计和评判的标准的提高，产品设计时如何进行参数分析和优化、如何在短时间内制定新工艺等问题成为新的挑战。正是由于计算机技术的飞速发展，才使得有限元方法的应用如此广泛和普及，使之成为最常用的分析工具，目前，国际上有 90%的机械产品和装备都要采用有限元方法进行分析，进而进行设计修改和优化。

工程上常应用经典的有限元方法计算拓扑优化中的未知结构响应，但是经典的有限元设计循环却存在如下缺点：整个设计循环涉及了 CAD/CAE、模拟仿真等众多学科领域，给各步骤相互之间的交流造成不便；在设计的过程中需要嵌入一些复杂功能的软件，而这些软件的操作条件和输入/输出格式都各不相同；在设计循环的每个步骤中的几何和物理表示形式都各不相同，而且某些步骤中得到的结果是精确值的近似表示，其数值缺陷严重影响了拓扑优化的有效性和效率。其中最后一点是 CAD/CAE 的无缝融合问题迟迟未得到真正解决的关键原因。显然，要克服工程设计和分析之间的障碍，就需要重建整个过程，但同时保持与现有技术的兼容性。其关键步骤在于确定一个唯一的几何模型，该模型可直接用作分析模型，或是从中可以自动建立几何精确的分析模型。这就需要将传统的有限元分析过程转变为基于 CAD 表达方式的分析过程。由于等几何分析（IGA）独特特点，即 CAD 模型和 CAE 模型可以统一为一个相同的数学模型，能够消除经典有限元方法对拓扑优化的负面影响，已成为一种很有前景的替代方法。

《中国制造 2025》提出，我国建设制造强国任务艰巨而紧迫。随着产业升级、数字化水平不断提高，国产 CAD、CAE 等软件技术也需要有全新的发展。等几何分析可视化技术将会成为降低专业人员门槛、提升软件操作可控性、缩短产品设计时间的关键步骤，正得到越来越多的关注和研究。本项目立足机械设计制造学科，以计算机图形学、可视化交互技术、等几何分析、拓扑优化为研究对象，有针对性地解决面向等几何拓扑优化的可视化问题。

1.2 等几何拓扑优化简介

拓扑优化的主要目的是在设计域寻求具有预期结构性能的最佳材料布局，它

源于一项开创性的工作，该工作讨论了材料经济性限制下的框架结构设计。就结构拓扑的表示模型而言，现有的优化方法主要可分为两个分支，分别是基于材料的模型（Material Description Models, MDMs）和基于边界的模型（Boundary Description Models, BDMs）。在前者所代表的拓扑优化方法中，设计域被离散成一系列带有密度属性的点或者单元，每个设计点或单元密度决定了设计与众相应位置处材料是否存在，该方法被称为基于密度的拓扑优化方法，包括了固体各向同性惩罚材料法（solid isotropic material with penalization Method, SIMP）和渐进结构优化法（Evolutionary Structural Optimization Method, ESO）等；而后者使用 BDMs 来表示结构拓扑，利用隐式或显示形式构造了一个更高维的函数，用于设计中拓扑的演变，且结构边界由函数的等值线或等值面定义，包括了水平集方法（Level Set Method, LSM）、相场方法（Phase Field Method）等。

在现有的拓扑优化工作中，经典的有限元方法（Finite Element Method, FEM）被用于求解数值分析中的未知结构相应。然而，有限元法在数值分析中有几个不足之处：（1）有限元网格只是集合的近似，而非精确表示；（2）相邻单元的低阶连续性；（3）获取高质量有限元网格相对低效。在这样的情况下，Huges 和他的同时提出了一种极具前景的 FEM 替代方案，称为等几何分析（IsoGeometric Analysis, IGA），而这可以完全消除 FEM 的上述问题。在等几何分析中，核心是将控制点和样条基函数在内的信息同时应用于几何表达欲数值分析，故几何模型和数值分析模型保持一致。结构几何和数值分析中的数学模型的统一为优化提供了极大的便利，也能够解决拓扑优化中出现的一些数值问题。

自从等几何分析问世之后，更多的研究人员开始开发新的拓扑优化方法，并使用等几何手段（而非传统的有限元手段）来实现拓扑优化的应用。将 IGA 引入拓扑优化的第一项工作可以追溯到 2010 年 Y-D Seo 等人的文章，其中讨论了 IGA 的形状优化应用及其对拓扑优化的扩展。后来，Y Wang 的工作讨论了如何使用修剪的样条曲线来呈现结构边界，然后提出了一套基于拓扑优化和 IGA 的等几何拓扑优化（Isogeometric Topology Optimization, ITO）的新框架，这为未来拓扑优化的发展打开了一个新窗口。在此之后，众多的研究工作持续开展，充分考虑了 IGA 在拓扑优化中的积极特性，也为开发更多更高效的 ITO 方法奠定了理论基础。IGA 取代拓扑优化中的经典有限元方法的方向受到越来越多的研究人

员的关注。

1.3 三维可视化图形工具 OpenGL 概述

OpenGL（Open Graphics Library）是一个跨编程语言、跨平台的编程图形程序接口，它将计算机的资源抽象称为一个个 OpenGL 的对象，对这些资源的操作抽象为一个个的 OpenGL 指令。OpenGL 图形库具有以下突出特点：

应用广泛：无论是 PC 机还是工作站，甚至是嵌入式设备，OpenGL 都能表现出它的高性能和强大威力。

跨平台性：OpenGL 能够在几乎所有的主流操作系统上运行。

可扩展性：通过 OpenGL 扩展机制，可以利用 API 进行功能扩充。

绘制专一性：OpenGL 只提供绘制操作访问，而没有提供建立窗口、接受用户输入等机制，它要求所运行环境中的窗口系统提供这些机制。

网络透明性：OpenGL 允许一个运行在工作站上的进程在本机或通过网络在远程工作站上显示图形。利用这种透明性，能够均衡共同承担图形应用任务的各工作站的负荷，也能使得没有图形功能的服务器使用图形工具。

本文选择在 Win32 应用程序环境中直接建立 OpenGL 应用程序框架进行可视化系统的开发，将三维悬臂梁的等几何拓扑优化结果（即密度值）以图像的方式显示，为研究者对于悬臂梁的拓扑优化提供了帮助，也为等几何拓扑优化的直观表达提供了图形化的理解，具有一定的研究意义。

1.4 本文的主要研究内容

第二章，介绍了基于变密度法的等几何拓扑优化方法，包括 B 样条与 NURBS 理论以及拓扑优化理论，为三维悬臂梁的拓扑优化提供理论基础；对三维悬臂梁在 matlab 中进行数学建模，确定载荷与边界点，转换现有开源代码并使其适应等几何拓扑优化的实际需要，最终计算得到各个单元对应的设计变量（即密度值），为可视化提供数据支撑。

第三章，阐述了可视化研究中所需用到的计算机图形学基础理论，为可视化系统的研究提供理论支撑；介绍了在窗口应用程序中搭建 OpenGL 图形渲染环境的方法。

第四章，使用 C++语言和 OpenGL 理论，针对本文研究所采用的等几何模型表达方法，完成重要变量与结构体的设计，以此为基础对控制点的移动与几何形变进行了探究，并比较了有限元与等几何拓扑优化在可视化方面的差异。

第五章，完成了三维悬臂梁等几何拓扑优化可视化系统的开发。主要包括系统界面设计、数据的产生与读取、可视化功能的实现等工作。

1.5 本章小结

本章首先对等几何拓扑优化与可视化技术研究的研究背景进行了介绍，简要概述了三维可视化图形工具 OpenGL 的特点，由此确立本文的主要研究内容。

第二章 基于变密度法的等几何拓扑优化方法

2.1 引言

本章首先对作为等几何构型基础的 B 样条和 NURBS 的基础理论进行了介绍，随后介绍了基于变密度法的等几何拓扑优化方法，并基于拓扑优化理论给出了 matlab 核心代码，通过代码中描述的方法得到了结构柔度值，为结构设计变量（即密度值）的计算奠定了基础，并为后文的可视化提供了数据支撑。

2.2 B 样条与 NURBS 理论

本节简要概述了 B 样条曲线和 NURBS 的构造。

2.2.1 Bernstein 多项式和 Bézier 曲线

一个阶数为 p 的 Bézier 曲线由 p+1 个 Bernstein 多项式基函数的线性组合表示而成。在这里，我们将基函数表示为 $\mathbf{B}(\xi)=\{B_{i,p}(\xi)\}_{i=1}^{p+1}$，同时将对应的控制点表示为 $\mathbf{P}=\{\mathbf{P}_i\}_{i=1}^{p+1}$，对于任意位于[0,1]内的参数坐标 ξ，对应的 p 阶 Bernstein 多项式可由以下递推式获得：

$$B_{i,p}(\xi)=\begin{cases}(1-\xi)B_{i,p-1}+\xi B_{i-1,p-1}(\xi) & i=1,2,...,p+1\\ 0 & i<1 \text{ 或 } i>p+1\end{cases}$$

其中 $B_{i,0}(\xi)=1$。阶次 $p=1,2,3$ 的 Bernstein 基函数图 2 如所示

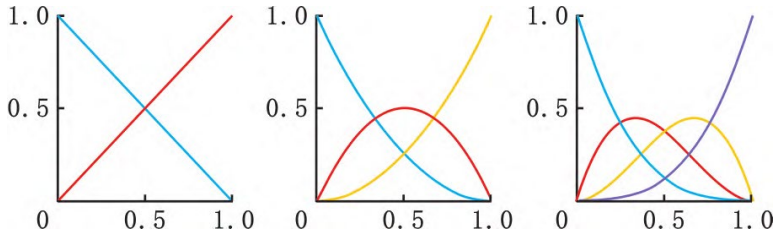


图 1 定义在[0,1]的 Bernstein 基函数

Bézier 曲线的表达式为

$$C_{Bezier}(\xi)=\sum_{i=1}^{p+1}\mathbf{P}_i B_{i,p}(\xi)=\mathbf{P}^{\mathrm{T}}\mathbf{B}(\xi)\qquad \xi\in[0,1]$$

2.2.2 节点向量与 B 样条曲线

节点向量由参数空间中的一组非递减的坐标组成。给定一组节点向量 $\Xi=\{\xi_1,\xi_2,...,\xi_{n+p+1}\}$，其中 p、n 分别表示 B 样条基函数的多项式阶数和控制点个数。对于给定的节点向量，B 样条的基函数由 Cox-de Boor 递推公式定义：

$$N_{j,p}(\xi)=\frac{\xi-\xi_j}{\xi_{j+p}-\xi_j}N_{j,p-1}(\xi)+\frac{\xi_{j+p+1}-\xi}{\xi_{j+p+1}-\xi_{j+1}}N_{j+1,p-1}(\xi)$$

$$N_{j,0}(\xi)=\begin{cases}1 & \xi_j\le\xi\le\xi_{j+1}\\ 0 & \text{其他}\end{cases}$$

通过组合 B 样条基函数与控制点可生成 B 样条曲线，B 样条曲线的表达式如下：

$$C_B(\xi)=\sum_{j=1}^{n}\mathbf{P}_jN_{j,p}(\xi)=\mathbf{P}^{\mathrm{T}}\mathbf{N}(\xi)$$

2.2.3 节点插入

在不改变曲线的几何或参数属性的情况下，可以向节点向量中插入节点。给定一组节点向量$\Xi=\{\xi_1,\xi_2,...,\xi_{n+p+1}\}$，插入新节点$\overline{\xi}\in[\xi_k,\xi_{k+1}]$，得到新的节点向量$\Xi=\{\xi_1,\xi_2,...,\xi_k,\overline{\xi},\xi_{k+1},...,\xi_{n+p+1}\}$。设$m=n+1$为插入节点之后得到的控制点数量，新控制点$\{\overline{\mathbf{P}}_i\}_{i=1}^{m}$可由插入前的控制点$\{\overline{\mathbf{P}}_i\}_{i=1}^{n}$得到，表达式如下：

$$\overline{\mathbf{P}}_i=\begin{cases}\mathbf{P}_1 & i=1,\\ \alpha_i\mathbf{P}_i+(1-\alpha_i)\mathbf{P}_{i-1} & 1<i<m,\\ \mathbf{P}_n & i=m,\end{cases}$$

其中

$$\alpha_A=\begin{cases}1 & 1\le A\le k-p,\\ \dfrac{\overline{\xi}-\xi_A}{\xi_{A+p}-\xi_A} & k-p+1\le A\le k,\\ 0 & A\ge k+1\end{cases}$$

2.2.4 NURBS

NURBS 可由一组节点向量$\Xi=\{\xi_1,\xi_2,...,\xi_{n+p+1}\}$、一组有理基函数$\mathbf{R}=\{R_{i,p}\}_{i=1}^{n}$和一组控制点$\mathbf{P}=\{\mathbf{P}_i\}_{i=1}^{n}$定义，表达式如下：

$$T(\xi)=\sum_{i=1}^{n}\mathbf{P}_iR_{i,p}(\xi)$$

NURBS 基函数定义如下：

$$R_{A,p}(\xi)=\frac{w_i N_{i,p}(\xi)}{W(\xi)}$$

其中权函数定义为

$$W(\xi)=\sum_{j=1}^{n} w_j N_{j,p}(\xi)$$

w_j 为第 j 个基函数对应的权重。

为了便于计算，$\mathbb{R}^n$ 空间中的有理曲线可由 $\mathbb{R}^{n+1}$ 空间中的多项式曲线投影而来，对应的高维空间称为投影空间。因此，给定 $\mathbb{R}^n$ 中定义的 NURBS 曲线，$\mathbb{R}^{n+1}$ 中对应的 B 样条曲线为

$$T(\xi)=\sum_{i=1}^{n} \tilde{\mathbf{P}}_i N_{i,p}(\xi)$$

在投影坐标系中可以将 B 样条的算法同样地直接应用于 NURBS。只要在投影坐标系中为 B 样条计算出新的控制变量，简单地除以权重就可以得到 NURBS 对应的控制变量。

2.3 变密度法的等几何拓扑优化理论

变密度法是以单元的相对密度作为设计变量，人为假定相对密度和材料弹性模量之间的一种方法。变密度法程序实现简单、计算效率高。M P Bendsøe 等人采用固体各向同性惩罚微结构模型（solid isotropic microstructures with penalization, SIMP），建立了单元弹性模量与密度之间的关系。SIMP 插值模型如下（式中变量均为无量纲形式）：

$$E(\rho_i)=\rho_i^p E_0,\qquad \rho_i\in[0,1]$$

式中 p 为惩罚系数，E_0 为实体材料的弹性模量。

该方法通过引入惩罚系数，对中间密度值进行惩罚，使中间密度向两端分布，连续变量能够很好地逼近 0/1 离散变量，从而得到清晰的结构。对于常见的以柔度最小化为目标的变密度法拓扑优化问题，其数学模型如下式所示：

$$\begin{cases} \text{find}: \boldsymbol{\rho} = \{\rho_1, \rho_2, ..., \rho_N\} \\ \min: C(\boldsymbol{\rho}) = \mathbf{U}^T \mathbf{K} \mathbf{U} = \sum_{e=1}^{N} \rho_e^p u_e^T k_0 u_e \\ \text{subject to}: \begin{cases} \mathbf{K}\mathbf{U} = \mathbf{F} \\ \dfrac{V(\boldsymbol{\rho})}{V_0} = f \\ \mathbf{0} < \boldsymbol{\rho}_{\min} \le \boldsymbol{\rho} \le \mathbf{1} \end{cases} \end{cases}$$

式中 N 为单元总数目，$\boldsymbol{\rho}$ 为设计变量向量（即离散后每个单元的密度值），p 为用来将单元密度限制在 0 到 1 内的惩罚系数，C 为结构柔度值，$\mathbf{F}$ 为载荷向量，$\mathbf{U}$ 为位移向量，$\mathbf{K}$ 为总体刚度矩阵，$V(\boldsymbol{\rho})$ 为材料体积，V_0 为设计域体积，f 为结构体积约束，$\boldsymbol{\rho}_{\min}$ 为最小密度向量。SIMP 方法旨在根据 V_0，在具有预期结构柔度 C 的设计域中计算出 $\boldsymbol{\rho}$ 的合理布局。

在计算得到结构整体柔度值后，即可在最优化的思想下对结构的设计变量（即密度）进行更新，得到单次迭代的最优解。

2.4 本章小结

本章介绍了等几何拓扑优化的理论基础，包括 B 样条和 NURBS 的核心理论以及基于变密度法的等几何拓扑优化理论，为后续进行三维悬臂梁的等几何拓扑优化提供了理论支撑；在 matlab 软件中根据等几何拓扑优化的基本结构，建立了三维悬臂梁的边界条件与载荷分布，并利用上面的公式进行计算，得到了结构柔度值；在每一次的迭代中通过该值即可针对设计变量（即密度值）进行更新，得到最终结果。

第三章 三维情形下的等几何拓扑优化可视化研究

3.1 引言

在现有的等几何拓扑优化代码中，可视化部分仍使用了与有限元拓扑优化可视化相同的方法，即直接指定控制点为顶点，生成相应的多边形区域。在这样的思想指导下，每个等几何单元的顶点与控制点重合：这不符合 NURBS 曲面的定义，即单元中每个点的位置需要通过控制点与节点向量共同计算，最后插值而成，故等几何单元的顶点与大部分控制点并非重合。

基于该问题，本文讨论利用NURBS基函数来进行等几何单元的绘制的方法，使得可视化结果能够忠实还原拓扑优化结果的几何信息，并通过计算着色器、内部冗余单元消隐算法等优化方法提高了等几何拓扑优化可视化的效率，并实现实时交互移动控制点改变图形几何形态，提升了在大规模拓扑优化可视化时的可用性。

3.2 等几何模型表达

3.2.1 等几何单元数据结构设计

本文讨论的二阶等几何单元由六块 NURBS 面片组成，并与 3×3×3=27 个控制点相关联；每一个等几何单元中包含一份密度值信息。根据前文 2.2 小节中关于 B 样条和 NURBS 理论部分的介绍，等几何单元体的六块 NURBS 面片上每一个顶点的位置信息均可由 27 个控制点的位置坐标和相应节点向量所计算出来的基函数相乘得到。单元结构体如下所示：

IsoElement
int ControlPtsIndex[3][3][3];
float Weights[3][3][3];
float Rho;
float ElementIndex[3];

等几何单元结构体 IsoElement 包含了四个主要的变量，即与等几何单元相关的 27 个控制点的索引、27 个控制点所对应的权重、单元密度值以及该单元体在 x/y/z 方向上的位置（便于确定该单元体对应节点向量上的哪一部分）。

3.2.2 可视化算法实现

1、拓扑优化结果绘制方法

首先基于 Cox-de Boor 递推公式和 NURBS 对应的权重公式求出三个维度上各个节点向量对应的 NURBS 基函数值。随后根据各个等几何单元控制点的索引即可定位到各单元所对应的 27 个控制点，并将这些控制点对应乘以相应的 NURBS 基函数值，即可得到等几何单元几何位置顶点。

这里以 X、Y、Z 方向单元数均为 3 的图形为例，三个方向上的节点向量均为 $\{0,0,0,\frac{1}{3},\frac{2}{3},1,1,1\}$，若 X 方向取节点向量为 $0\sim\frac{1}{3}$ 的部分，Y、Z 方向均取 $\frac{2}{3}\sim1$，即可计算出如图 2 立方体对应的基函数：

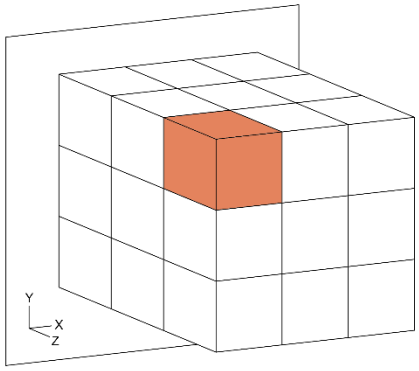


图 2　3×3×3 网格的设计域

为了便于观察，仅显示 X、Y 两个维度上的设计域如图 3 所示。从图中可观察到每个单元所对应的控制点、节点向量和 B 样条基函数等信息；其中绿色控制的点标示的是图 4 中橙色标记立方体所对应的 X、Y 方向上的控制点。

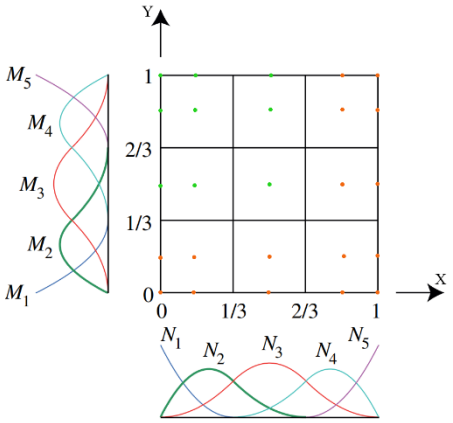


图 3　X、Y 方向上的 3×3 网格的设计域

除了需要计算出每一个节点所对应的实际坐标，在进行顶点的计算并导入 OpenGL 进行绘制的过程中，还需要对节点进行细化，即在 NURBS 的节点向量中插入节点，然后再将其绘制出来，使用 OpenGL 绘制出来的图形更加趋近于实

际的曲线曲面。

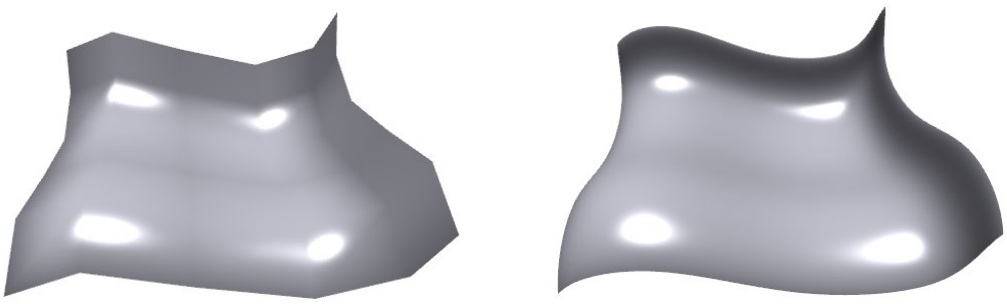


图 4　细分程度低和细分程度高的 NURBS 曲面

对等几何单元体的绘制结束之后，基于 2.3 节中变密度法的等几何拓扑优化理论，可根据索引将等几何拓扑优化计算得到的密度值矩阵分别对应填入各个单元体中。在使用 OpenGL 进行绘制时，采用颜色区分不同的密度值，即密度值高的地方颜色深，密度值低的地方颜色浅，转化为公式可写为

$$color = 1 - density$$

其中 *color* 表示区间为[0,1]的值，对应 OpenGL 中片元着色器的输出结果。

在进行实际的代码编写时，为了提升性能，笔者使用了 OpenGL 4.3 的新功能，即计算着色器，对等几何单元体的 NURBS 面片进行计算加速。计算着色器(Compute Shader)是一个特殊类型的着色器，其独立于 OpenGL 的图像渲染管道之外，但是可以对 GPU 资源（存放在显存中）进行读取和写入操作。本质上来说，计算着色器允许使用者访问 GPU 来实现数据并行算法，而不需要进行任何的实际绘制，它被设计用于充分利用图像处理器 GPU 的大规模并发计算能力，因此它们的作用并不仅仅限于进行图像渲染。像这样的非图形应用使用 GPU 的情况，被称之为 GPGPU（General Purpose GPU）编程。利用计算着色器带来的并发特性，能够同时对多个等几何单元进行计算，带来效率上的显著提升。

具体算法描述如下（在计算着色器中）：

（1）确定单元所包含的六个面对应的基函数；

（2）循环遍历单元体上各个面的顶点，并根据单元体数据结构中的 ElementIndex 属性和该顶点的顶点索引，确定该顶点所对应的节点向量，按照 $T(\xi)=\sum_{i=1}^{n}\mathbf{P}_i R_{i,p}(\xi)$ 先算出基函数的值，随后与控制点坐标相乘求和即可得到所计算的顶点的实际位置。

（3）从等几何拓扑优化计算得到的密度值矩阵中，根据 ElementIndex 信息取出该单元所对应的密度值，并通过 $color = 1 - density$ 的映射关系，赋予顶点相应的颜色值。

（4）计算该等几何单元体六个面上所有三角面片的索引，为填入 OpenGL 索引缓冲区并进行正确的可视化做准备。

如图 5 所示，为 30×20×5 的等几何网格拓扑优化并根据如上算法进行可视化后得到的结果。该结果由拓扑优化迭代 100 次得到，并限定密度值显示阈值为 0.5。

图 5　等几何网格可视化结果

本节通过拓扑优化结果可视化算法得到了等几何图形与密度值的正确显示，但仍存在问题：图形内部无法被观察到的等几何单元同样进入了 OpenGL 的可视化流水线进行了显示，这导致了大量不必要的图形资源消耗，并且随着细分次数的增加，这样的消耗也成倍数增加。针对这一问题，本文对等几何图形内部冗余单元提出了一种基于密度值的消隐算法，提升了程序整体的性能。

2、内部冗余单元消隐算法

消隐问题是计算机图形学中的一个十分重要的内容，对内部冗余单元的消隐常常能大幅降低图形资源消耗，提升动态可交互性。但在对通用的消隐算法进行研究之后，笔者发现这些算法只能解决一些相对基础的问题，难以在笔者设计的等几何拓扑优化可视化程序中获得实际的效率提升。本节根据等几何可视化的消隐和图形显示特点，提出一种用于等几何可视化的内部冗余单元消隐算法。

如图 6 所示，为 5×5×5 大小的等几何悬臂梁网格未经内部冗余单元消隐而直接可视化得到的效果，除了外部需要直接参与显示的一层单元以外，内部的 3×3×3 网格对用户不可见，可直接消隐节约性能。本节的算法从等几何拓扑优化得到的密度值矩阵入手，着力于减少这一不必要的可视化资源开销。

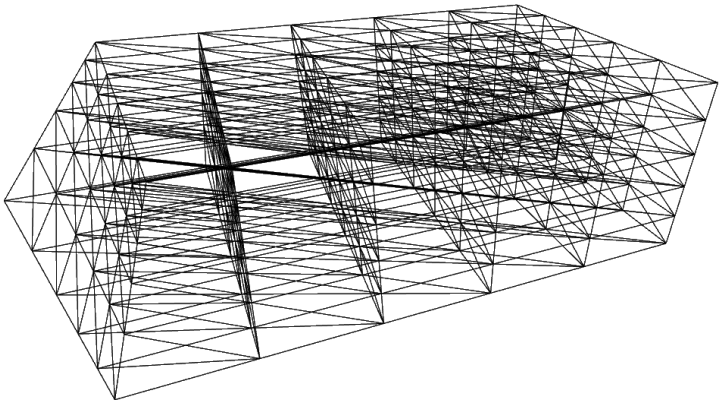


图 6　5×5×5 大小的悬臂梁网格

算法核心思路如下：

维护一个三维阵列 isBound 用于存储每个单元是否为边界，其大小维度与设计域网格相同，若 isBound 中的值为 true，则表明对应单元是边界单元，可视化时需要显示；若 isBound 中的值为 false，则表明对应单元不是边界单元，单元密度值设为–1，在可视化时被密度值阈值筛去，不参与显示。分别从不同方向遍历每一列单元，通过左右指针的方式找到各边界单元，并通过一定条件改变 isBound 中的对应值。

仍以 X、Y、Z 方向单元数均为 3 的图形为例，首先设定一个 3×3×3 大小的三维数组 isBound，并将所有的值均赋为 false；如图 7 所示，分别遍历 X、Y、Z 轴向的 9 列单元（共需遍历 27 次），根据每次遍历得到的结果改变 isBound 数组中的对应值，确定各单元是否为边界。

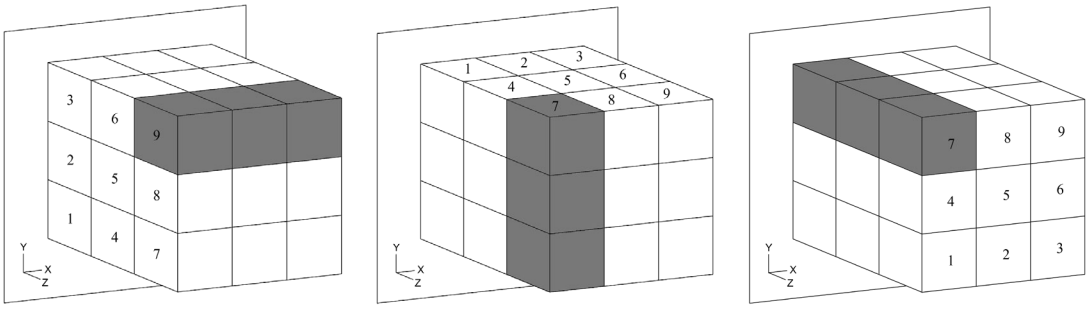


图 7　与 X、Y、Z 轴平行的单元列

在进行每一列等几何单元的遍历时，采用双指针法来确定该列存在的边界单元。在遍历开始前，将左右两个指针均指向该列的第一个单元，若左指针指向的单元密度值小于显示阈值，那么左右指针均向后移一位；若左指针指向的单元密

度值大于显示阈值，那么将右指针向后移动直到右指针所指单元密度值小于阈值，即可确定左右指针当前位置为边界单元，并将当前位置所对应的 isBound 阵列中的值赋为 true，左右指针均移动到右边界的下一个位置；不断往复直至左指针指向该列单元中的最后一个单元。分别在 X、Y、Z 三个方向的每一列单元上均应用改算法，最终即可通过 isBound 数组得到所有边界单元的索引位置。

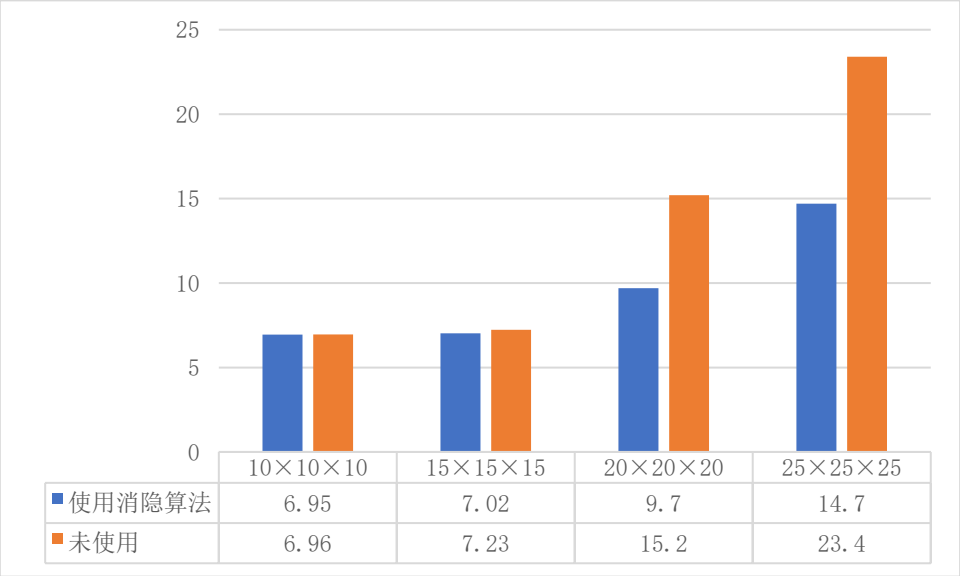
算法伪代码如下：

算法：内部冗余单元消隐

输入：RhoMatrix 密度值阵列，nelx/nely/nelz 各方向上的单元数量
输出：isBound 边界判定阵列

```
1.  isBound:=[nelx][nely][nelz];
2.  for i=0 to (nelx-1) do
3.     for j=0 to (nely-1) do
4.        RhoArray=i、j 索引对应的单元列
5.        isBoundArray=FindBound(RhoArray)
6.        用 isBoundArray 更新 isBound 中的值
7.  对其它两个方向重复上述循环
8.
9.  function FindBound(RhoArray)
10.    isBoundArray:=[RhoArray.size];
11.    for indexLeft=0 to (RhoArray.size-1) do
12.       indexRight=indexLeft;
13.       if RhoArray[indexLeft]<密度值显示阈值  then
14.          indexLeft=indexLeft+1;
15.          continue;
16.       else
17.          while RhoArray[indexRight]≥密度值显示阈值  then
18.             indexRight=indexRight+1;
19.                if indexRight==RhoArray.size then
20.                   break;
21.          isBoundArray[indexLeft]=true;
22.          isBoundArray[indexRight-1]=true;
23.          indexLeft=indexRight;
24.    return isBoundArray;
```

在对算法的性能测试中，设置细分等级为 30，分别对各方向单元数为 10、15、20、25 时使用和未使用消隐算法进行可视化试验，并比较了渲染每帧所需要的毫秒数。每帧所需毫秒数的增加表明在渲染时 GPU 所需时间更多，消耗资源越大。如下表所示，使用消隐算法与未使用时存在着较大的性能差异，且随着单元数量提升，消隐算法带来的性能提升更加显著。



	10×10×10	15×15×15	20×20×20	25×25×25
使用消隐算法	6.95	7.02	9.7	14.7
未使用	6.96	7.23	15.2	23.4

3.2.3 模型的控制点与变形方法

（这里暂定写 OpenGL 的拾取，但是不知道最后代码能不能做得出来）

第四章 基于 OpenGL 等几何拓扑优化可视化系统搭建

4.1 引言

本章基于前文已经累积的理论和工作，自主进行基于 OpenGL 的三维悬臂梁等几何拓扑优化可视化系统的搭建。本系统以 Windows 应用程序接口（即 WinAPI）作为窗体系统框架，配置 OpenGL 环境以进行图形显示及颜色渲染，基本实现了一般使用可视化系统的功能。

4.2 可视化系统概述

在获取等几何拓扑优化后可视化的算法程序框图如图 8 所示。

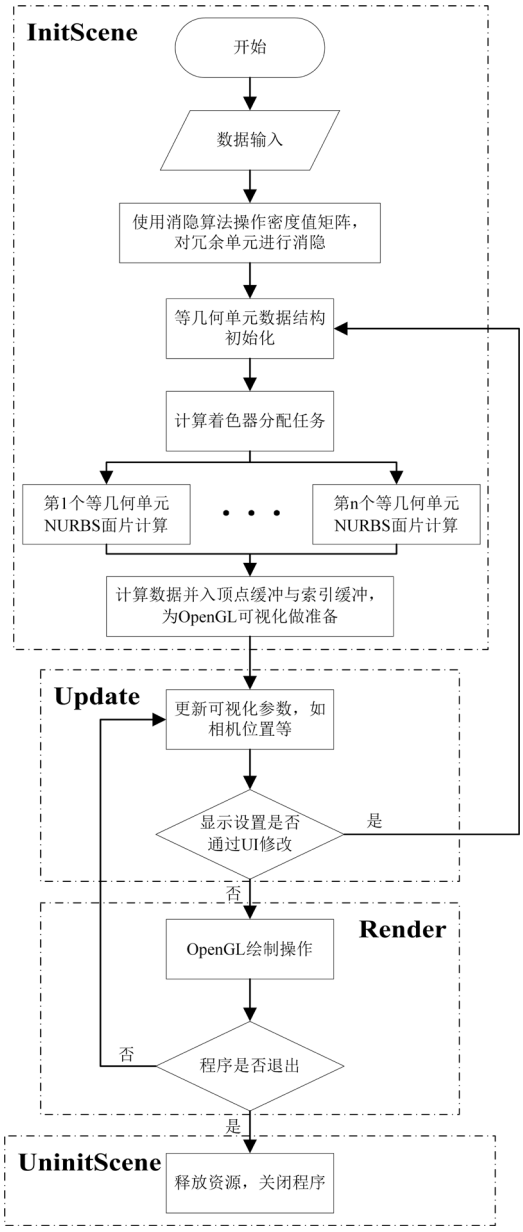


图 8　拓扑优化可视化程序框图

可视化的基本框架由如下四个部分组成：

（1）InitScene：在该函数中实现整个应用程序的初始化，包括窗体初始化、OpenGL 初始化、主要数据初始化等工作，并在该阶段中进行首次 NURBS 细分曲面的计算，将计算着色器得到的数据与相应的 OpenGL 缓冲区绑定，在之后的渲染过程中进行数据在屏幕视口的显示。

（2）Update：在渲染步骤前调用，主要进行摄像机的更新以及显示图形形状、状态、显示模式等属性的更新，并根据实际情况确定是否需要重新进行细分曲面计算。

（3）Render：渲染过程，是利用 OpenGL 进行图形显示所需的关键函数，再整个应用程序正常运行的时间内，都会根据需要不断进行调用，并在调用函数的过程中给出所有关于图形显示的命令。

（4）UninitScene：在图形程序退出时进行调用，断开程序与 OpenGL 的连接，清理所有 C++生成的指针等信息，防止内存泄漏等问题的出现。

4.3 OpenGL 框架搭建

4.3.1 Win32 程序

Windows 应用程序接口（Windows API）是微软 Windows 操作系统中的一套核心应用程序接口。Windows API 这一叫法实际上是多个 Windows 平台上相似接口的统称，这些接口也拥有各自的名字，如 Win32 API。几乎所有的 Windows 应用程序都在与 Windows API 进行交互。

Windows 程序通过类似于自动化中的响应机制不断处理外部信息，而非设置一个无限循环来接收消息。系统以消息的形式将输入传递给窗口过程。消息由系统和应用程序生成。系统在每个输入事件（例如，用户键入时、移动鼠标或单击滚动条等控件）生成消息。系统还会生成消息，以响应应用程序带来的系统更改，例如应用程序更改系统字体资源的池或调整其窗口的大小。应用程序可以生成消息，以指示其自己的窗口执行任务或与其他应用程序中的窗口通信。系统使用一组四个参数将消息发送到窗口过程：一个窗口句柄、一个消息标识符和两个名为消息参数的值。窗口句柄标识消息的预期窗口。系统使用它来确定应接收消息的窗口过程。

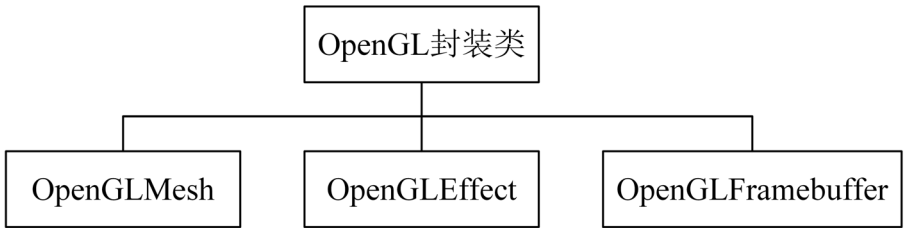
一个基本的窗口过程函数如下所示：

```
LRESULT CALLBACK WndProc(
    HWND hwnd,          // 窗口句柄
    UINT uMsg,          // 消息 ID
    WPARAM wParam,      // 第一个消息参数
    LPARAM lParam)      // 第二个消息参数
{
    switch (uMsg)
    {
        case WM_CREATE:
            // 初始化窗口
            return 0;
        case WM_PAINT:
            // 绘制窗口客户区
            return 0;
        case WM_SIZE:
            // 设置窗口大小和位置
            return 0;
        case WM_DESTROY:
            // 清除窗口数据对象
            return 0;
        //
        // 处理其它消息
        //
        default:
            return DefWindowProc(hwnd, uMsg, wParam, lParam);
    }
    return 0;
}
```

在注册窗口类时，将窗口过程与窗口类相关联。使用有关类的信息填充 WNDCLASS 结构，并将结构地址传递给 RegisterClass 函数。在注册窗口类后，窗口过程即会自动与使用该类创建的每个新窗口相关联。

4.3.2 OpenGL 函数封装

OpenGL 定义了一个跨编程语言、跨平台的编程接口规格的专业的图形程序接口。它用于三维、二维图形，是一个功能强大，调用方便的底层图形库。由于 OpenGL 是图形的底层图形库，故本文运用面向对象编程的思想，对 OpenGL 中的 API 进行封装，便于等几何拓扑优化可视化系统的搭建。下面介绍与可视化密切相关的几个 OpenGL 封装类：



（1）OpenGLMesh

OpenGLMesh 类中包含了参与可视化的网格所对应的顶点缓冲区、索引缓冲区等信息，直接参与 OpenGL 流水线，是各种不同模型文件解析之后得到的抽象信息。在创建场景的过程中通过解析模型将对应信息传入 OpenGLMesh 对象后，即可在渲染过程中通过 OpenGLMesh 类中的绘制函数，将包含的信息传入 OpenGL 接口中，实现屏幕上的显示。

（2）OpenGLEffect

OpenGLEffect 是对 OpenGL 着色器相关函数的封装，并引入渲染管线概念，仿照 DirectX 12 中的固定管线方式，包含了管线中所对应 glsl 代码中所包含的变量信息和 uniform 属性的索引/位置值，让 C++端对 glsl 中对应变量的更新得以简化，也让对整个管线的操作变得简洁、清晰。

OpenGLEffect 对象利用 GLCreateEffectFromFile 函数进行初始化，函数介绍如下：

```
bool GLCreateEffectFromFile(
    const char* vsfile,          //顶点着色器
    const char* gsfile,          //几何着色器
```

const char* psfile,　　//片元着色器

OpenGLEffect** effect)　　//传入的 OpenGLEffect 对象指针

（3）OpenGLFramebuffer

OpenGLFramebuffer 是对 OpenGL 中帧缓冲相关函数的封装。帧缓冲是一些二维数组和 OpenGL 所使用的存储区的集合：颜色缓存、深度缓存、模板缓存等。一般情况下，帧缓存完全由 window 系统生成和管理，由 OpenGL 使用。该封装类的作用在于更加方便地进行帧缓冲的创建，并更好地管理和渲染帧缓冲的附件（Attachment）。

4.3.3 Win32 构建 OpenGL 上下文

由于 OpenGL 在创建上下文之前是不存在的，所以 OpenGL 上下文的创建不受 OpenGL 规范的约束；相反，它由特定于平台的 API 进行管理。结合以上对 Windows 应用程序以及 OpenGL 的具体介绍和详细说明，尝试在一个 Win32 应用程序中直接建立一个 OpenGL 运行的上下文环境，实现 Windows 应用程序对 OpenGL 的管理。

（1）设置 OpenGL 运行环境

把程序在编译时所需要的头部文件 gl.h、glu.h 集中放置在 OpenGL/include 目录下，同样地，在程序代码中进行预定义：

```
#pragma comment(lib, "OpenGL32.lib")
#pragma comment(lib, "GLU32.lib")
```

这两个链接库文件分别与 gl.h 和 glu.h 一一对应，是 OpenGL 的静态链接库的一部分，可以在 Windows SDK（微软公司出品的一个软件开发包，向在微软的 Windows 操作系统和.NET 框架上开发软件和网站的程序员提供头文件、库文件、示例代码、开发文档和开发工具）找到。将这两个链接库文件集中放置在 OpenGL/lib 文件夹中，便于程序编译时进行与之链接。

同时在代码中进行预定义：

```
#pragma comment(lib, "gdiplus.lib")
```

这是一个 C++ 源代码中的预处理指令，它用于告诉编译器链接到 GDI+库中的特定函数。GDI+是 Windows 系统中内置的图形处理库，它提供了一组高层次的图形处理 API，支持位图、图形、文本、动画等多种图形元素的绘制。

（2）创建 Windows 窗口

首先进行窗口类 WNDCLASS 的注册。WNDCLASS 是一个由系统支持的结构，用来储存某一类窗口的信息,如消息处理函数，图标、背景等。也就是说，结构 WNDCLASS 包含一个窗口类的全部信息，也是 Windows 编程中使用的基本数据结构之一，应用程序通过定义一个窗口类确定窗口的属性。

然后调用 CreateWindow 函数，将某个 WNDCLASS 定义的窗体变成实例。该函数创建一个重叠式窗口、弹出式窗口或子窗口。它指定窗口类、窗口标题、窗口风格，以及窗口的初始位置及大小（可选）。函数也指定该窗口的父窗口或所属窗口，以及窗口的菜单。

CreateWindow 函数返回参数为窗口句柄 hwnd，并可向 GetDC 函数传入该窗口句柄。Windows 中的每个窗口都有一个与之相关的设备上下文（Device Context, DC），而 GetDC 函数就是用来检索指定窗口或整个屏幕的工作区的设备上下文的句柄。可以在后续 GDI 函数中使用返回的句柄 hdc 在设备上下文中绘制。

（3）设置设备的像素格式

如前所述，每个窗口都有一个与之相关的设备上下文，而该对象可以存储一种称为像素格式的结构。该结构定义了 OpenGL 绘图方式，如设置在窗口中绘图、颜色模式和深度缓冲区大小等。

创建像素格式的方法是填写描述所需功能的结构 PIXELFORMATDESCRIPTOR。然后，将该结构赋予 ChoosePixelFormat 函数，该函数将返回一个数字，该数字表示在支持的像素格式列表中可以找到的最接近的匹配项。然后将此数字设置为设备上下文的像素格式。上述操作的具体实现如下：

```
PIXELFORMATDESCRIPTOR pfd = {
  sizeof(PIXELFORMATDESCRIPTOR),   // 大小
  1,   // 版本
  PFD_DRAW_TO_WINDOW|   // 窗口绘图
  PFD_SUPPORT_OPENGL|   // 获得 OpenGL 支持
  PFD_DOUBLEBUFFER,   // 设置双缓冲模式
  PFD_TYPE_RGBA,   // RGBA 模式
```

```
    32, // 颜色位数
    0, 0, 0, 0, 0, 0,   // 不用于选择模式
    0,   // 不用于选择模式
    0, 0, 0, 0, 0, 0,   // 不用于选择模式
    24, 8, 0,   // 分别指定深度缓冲区位数、模板缓冲区位数、辅助缓冲区数量
    0, 0, 0, 0, 0   //在此不使用
};
// 选择最接近的像素格式匹配项
int pixelformat = ChoosePixelFormat(hdc, &pfd);
if (pixelformat == 0) return false;
// 进行设备上下文的像素格式设置
SetPixelFormat(hdc, pixelformat, &pfd);
```

（4）创建 OpenGL 渲染上下文

只要在设备上下文中设置了像素格式，就可以进行 OpenGL 渲染上下文的创建。通过调用 wglCreateContext 创建渲染上下文，该函数以设备上下文句柄作为参数，返回渲染上下文的句柄（类型为 HGLRC）。

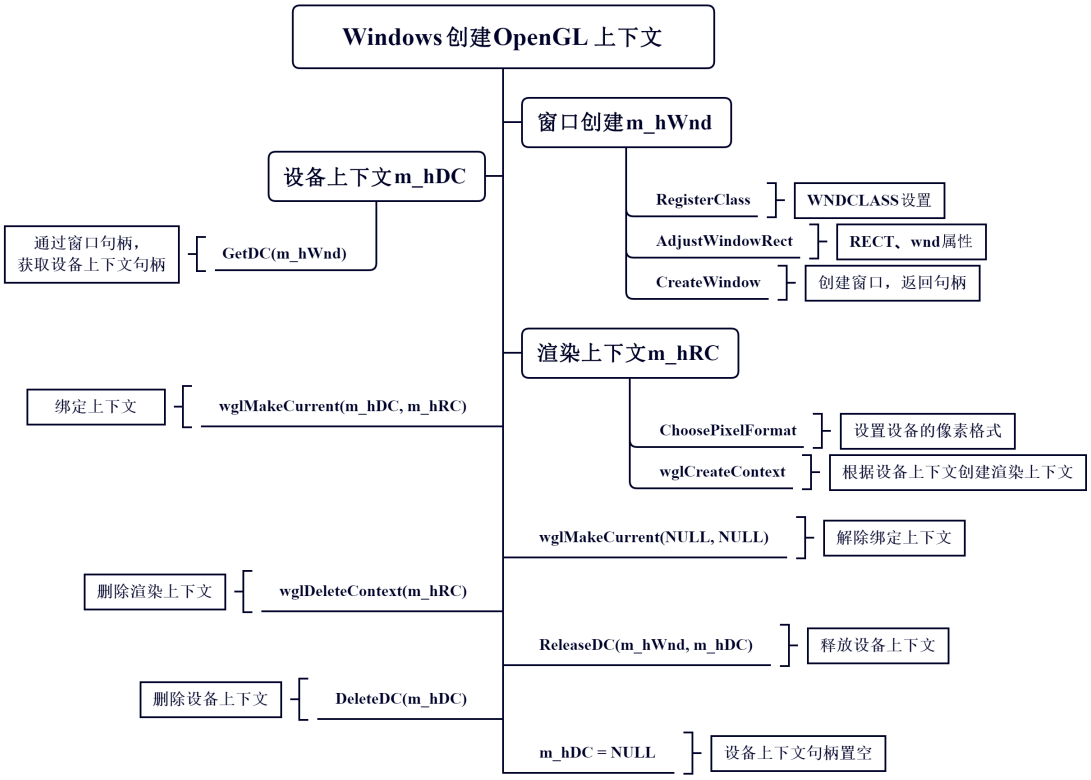
在使用 OpenGL 之前，必须保证切换到当前创建的渲染上下文进行渲染。该操作通过 wglMakeCurrent 函数进行实现，该函数以设备上下文句柄和渲染上下文句柄作为参数，使得程序完成新上下文的替换。此后 OpenGL 函数将引用新渲染上下文中的状态。上述操作的具体实现如下：

```
// 创建渲染上下文
HGLRC hglrc = wglCreateContext(hdc);
if (hglrc == nullptr) return false;
// 更新当前设备上下文的渲染上下文
wglMakeCurrent(hdc, hglrc);
```

（5）删除 OpenGL 渲染上下文

对于非当前使用的渲染上下文，可以调用 wglDeleteContext 进行删除。

在 Windows 程序中创建 OpenGL 上下文流程示意图如下：



4.4 可视化功能模块

4.4.1 数据读取模块

基于第二章中拓扑优化所得到的各项等几何 NURBS 面片参数以及最后计算得到的单元密度矩阵，程序首先在 InitScene 部分对数据进行读取和处理，主要包括数据输入、消隐算法处理密度值矩阵，以及等几何数据结构初始化几个部分。

本文的数据生成和数据读取均使用二进制文件进行，其中二进制文件相较于字符串形式存储的文件更省空间，在使用 C++进行读取时也具有更快的反应速度。C++端分别读取、处理了拓扑优化结果中的控制点索引矩阵、权重矩阵、节点向量和单元密度矩阵这四个主要的数据。以读取浮点形式二进制文件的具体代码为例：

```
// 传入文件名和数组引用，函数退出后即可得到对应文件的所有数据
void read_float(std::string strFile, std::vector<float> &buffer)
{
   std::ifstream infile(strFile.c_str(), std::ifstream::binary);
```

```
    if (!infile.is_open())   // 判断文件是否存在、是否满足读取条件
    {
        printf("Read File:%s Error ... \n", strFile.c_str());
        return;
    }
    infile.seekg(0, std::ifstream::end);
    long size = infile.tellg();
    infile.seekg(0);
    // 输出文件的名称和大小
    printf("The file: [%s] has: %ld(byte) ..... \n", strFile.c_str(), size);
    float temp;   // 临时变量，用于存储读取出来的单个数据
    while (infile.read((char *)&temp, sizeof(float)))
    {
        int readedBytes = infile.gcount();
        buffer.push_back(temp);   // 将读取的临时变量填入数组中
    }
}
```

在结束了所有数据的读取之后，即可基于 3.3.2 部分进行针对于单元密度矩阵的冗余网格消隐操作。为了防止在网格消隐操作后无法对原有的未消隐前的结果进行显示，所以在原有的密度值矩阵基础上复制一份进行消隐操作。程序中设置了是否打开网格消隐操作的开关按键，在运行时可根据使用者的需要切换显示模式，达到更好的观察效果。程序会 Update 阶段根据不同情况向等几何数据结构填入不同的数值，以实现 Render 阶段的正确渲染。

4.4.2 曲面细分模块

基于第三章中的三维情形下的等几何拓扑优化可视化研究部分，本节更加具体地介绍拓扑优化可视化中的曲面细分的实现方法。曲面细分模块主要分为两部分：C++端的数据准备、着色器分配操作和 glsl 端的计算操作。

（1）OpenGL 绑定缓冲对象与数据

在 OpenGL 上下文中，定义了一些绑定点，可以将一个 Uniform 缓冲链接至

绑定点。在创建 Uniform 缓冲之后，将该缓冲绑定到其中一个绑定点上，并将着色器中的 Uniform 块绑定到相同的绑定点，把它们连接到一起。本文用于绑定缓冲对象的函数主要是 glBindBufferBase，该函数需要传入一个目标、一个绑定点索引和一个 Uniform 缓冲对象作为它的参数，并将缓冲对象与指定索引的 Uniform 块关联起来。以传入单元密度值数据为例：

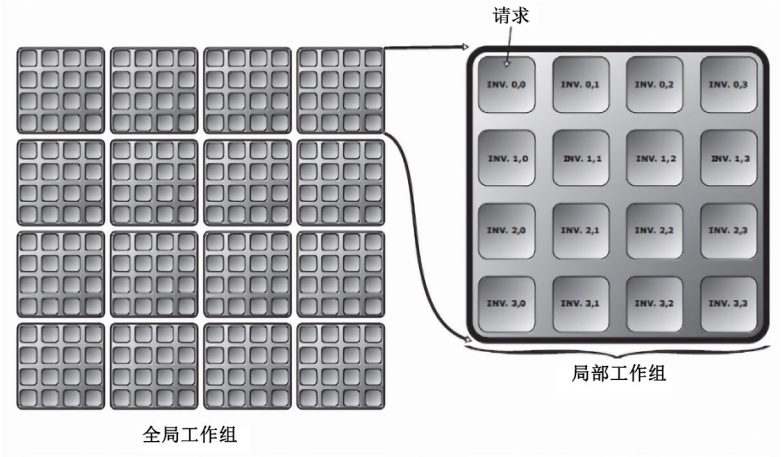
```
uint32_t rhoBuffer
glGenBuffers(1, &rhoBuffer);  // 生成缓冲区对象的名称
glBindBuffer(GL_SHADER_STORAGE_BUFFER, rhoBuffer);
glBufferData(GL_SHADER_STORAGE_BUFFER,  nelU*nelV*nelW*sizeof(float), surfacerho, GL_STATIC_READ);  // 将单元密度值数据读入缓冲区对象
// 将 rhoBuffer 绑定到链接点 4 上
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 4, rhoBuffer);
```

同理，对于控制点、权重以及需计算面片的顶点缓冲和索引缓冲也可采取类似的的方式进行绑定。

（2）分配着色器与着色器计算

在 OpenGL 中，计算着色器的任务以组为单位进行执行，称为工作组（work group）。拥有邻居的工作组被称为本地工作组，这些组可以组成更大的组，称为全局工作组，而其通常作为执行命令的一个单位。

计算着色器会被全局工作组中每一个本地工作组中的每一个单元调用一次，工作组的每一个单元称为工作项，每一次调用称为一次执行（invocation）。执行的单元之间可以通过变量和显存进行通信，且可执行同步操作保持一致性。



在 C++端，通过使用函数 glDispatchCompute()把工作组发送到计算管线上，

函数对应的三个参数为对应方向上工作组的数量。如：

```
glDispatchCompute(nelU, nelV, nelW);  // 工作组的数量对应各方向单元数量
```

在 glsl 端，使用 local_size_x, local_size_y, local_size_z 来进行本地工作组的大小的指定，默认均为 1。如：

```
layout(local_size_x=8,local_size_y=8)in;  // 声明本地工作组大小为 8×8×1
```

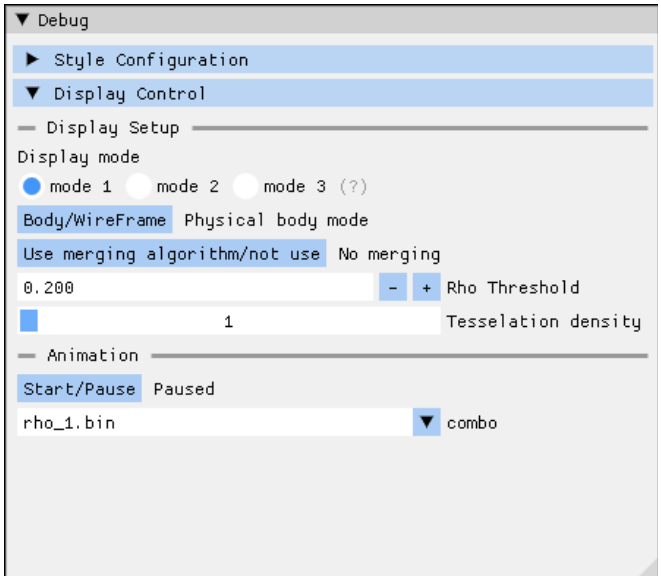
4.4.3 UI 界面模块

基于前文模块中对于运行时参数设置的需求和为了更加方便对软件功能的测试，本文所介绍的等几何拓扑优化可视化程序基于 ImGui 进行了 UI 界面。ImGui 又称为 Dear ImGui，它是与平台无关的 C++轻量级跨平台图形界面库，没有任何第三方依赖，可以将 ImGui 的源码直接加到项目中使用。

以调节显示模式为例，程序设置了三种主要的显示模式，分别是显示等几何图形、只显示控制点以及两者全部都显示的模式，分别对应 mode 1、mode 2 和 mode 3。ImGui 空间的 RadioButton 函数分别传入按键名称、修改的参数指针和修改值，并在程序运行时对 displayMode 值进行修改，达到 UI 界面控制程序运行的效果。调节显示模式的具体 ImGui 相关代码如下：

```
int displayMode = 0;
ImGui::Text("Display mode");
{
   ImGui::RadioButton("mode 1", &displayMode, 0);
   ImGui::SameLine();
   ImGui::RadioButton("mode 2", &displayMode, 1);
   ImGui::SameLine();
   ImGui::RadioButton("mode 3", &displayMode, 2);
   ImGui::SameLine();
   HelpMarker("displayMode = 0 : Display only physical body;\n"
              "displayMode = 1 : Display only control points;\n"
              "displayMode = 2 : Display both");
}
```

等几何拓扑优化可视化程序中所设计的 ImGui 界面如下：



（这里要 P 图 把界面上面打上序号 然后按照①②③④⑤这样一个个把功能介绍一下）

4.5 可视化效果展示