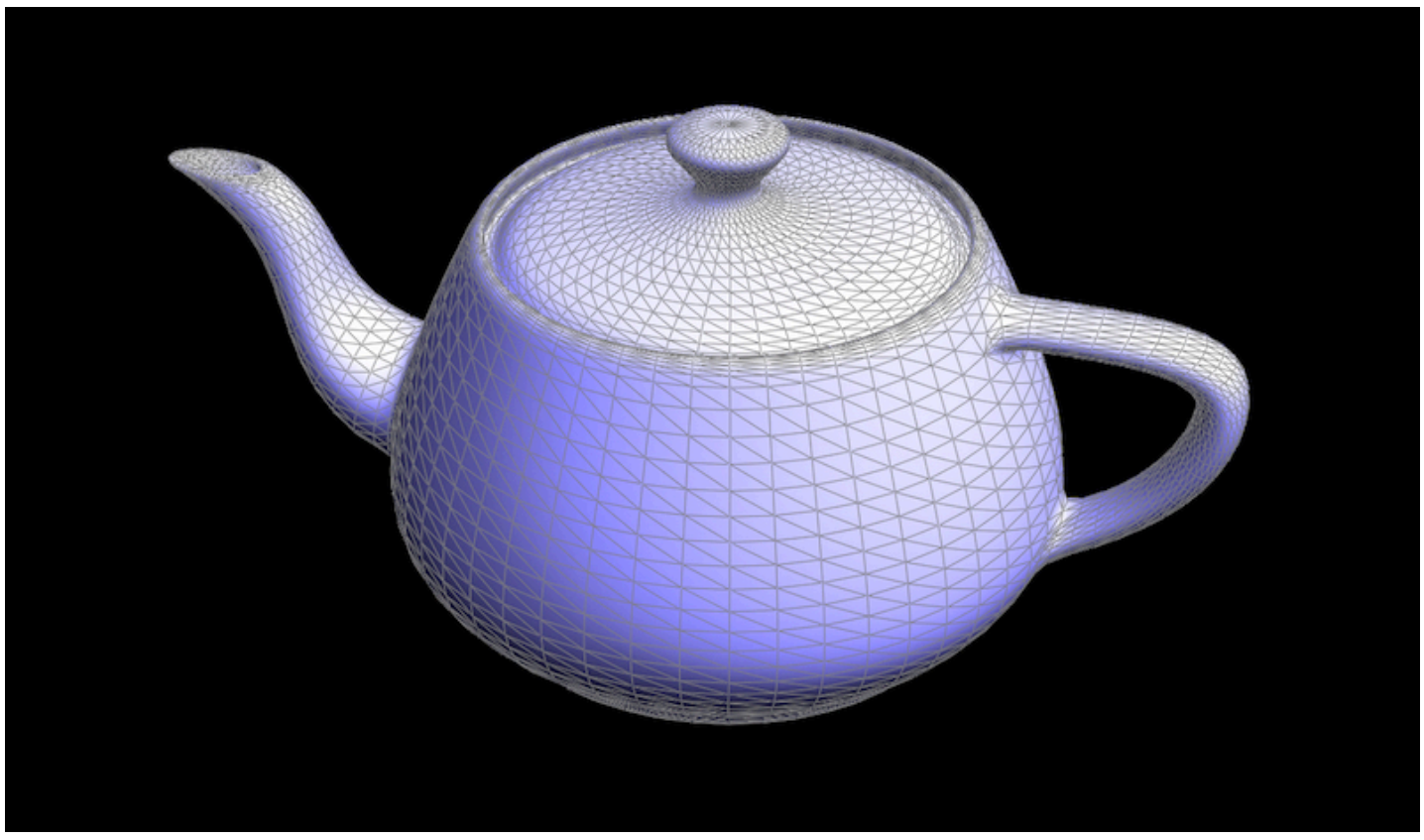
计算机图形学作业2



在本次图形学作业中，你将探索课程中的几何建模主题。你将使用 de Casteljau 算法构建贝塞尔曲线和曲面、实现半边数据结构表示的三角形网格，并实现基于Loop Subdivision的网格细分/上采样方法。你也可以选择使用免费的三维建模软件（例如 Blender）设计自己的三维物体，超额完成作业任务。

请大家自行完成编程，作业中遇到问题，可与其他同学进行技术交流，但请勿直接抄袭代码。

源代码的开发和编译方法见另一文件（`计算机图形学作业编译说明.pdf`）。

在实现代码的同时，请同时完成作业报告。我们给出了报告模板`作业报告.docx`，请按照附加提示完成写作。

作业结构

作业2需要实现6个算法，部分算法较简单，仅需几行代码，是其他算法的支撑。我们将任务分成两个任务模块。

模块一：贝塞尔曲线和曲面

- 算法1：利用 1D de Casteljau 实现贝塞尔曲线
- 算法2：基于 1D de Casteljau 实现贝塞尔曲面

模块二：基于半边数据结构的三角网格上采样

- 算法3：加权面积法求顶点法线
- 算法4：边翻转（Edge Flip）
- 算法5：边分割（Edge Split）
- 算法6：网格上采样（Loop Subdivision）

如何运行可执行文件

本作业编译完的可执行文件名为`meshedit`，运行命令如下：

```
./meshedit <PATH_TO_FILE>
```

请注意，`meshedit`后面的参数为输入文件的路径，此作业输入的文件包含两类，一类是曲线，另一类是曲面，它们的文件格式是不同的，如下表所示：

模块/算法	预期文件格式
算法1	贝塞尔曲线 (.bzc)
算法2	贝塞尔曲面（.bez）
算法3-6	COLLADA 网格文件 (.dae)

举个例子，对于算法1来说，执行以下命令将加载名为curve1的贝塞尔曲线：

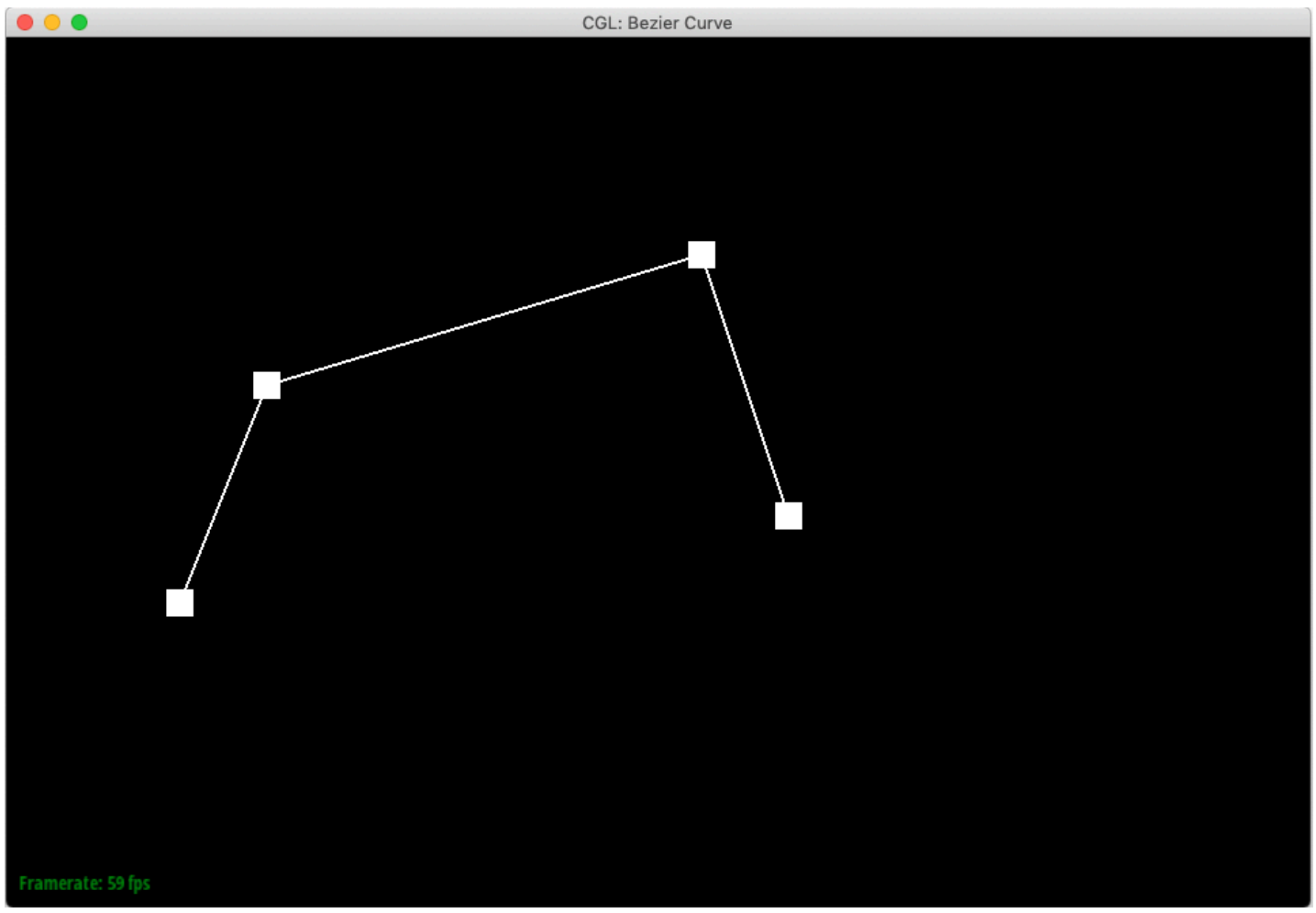
```
./meshedit ../bzc/curve1.bzc
```

使用图形用户界面 (GUI)

接下来，我们详细介绍如何使用`meshedit`提供的 GUI。以下主要介绍鼠标和键盘事件对应的程序功能。你可暂时略过此部分，待当你完成作业后再准备使用 GUI 时，再更详细地阅读它们。

算法1 - 贝塞尔曲线

当你运行`meshedit`载入贝塞尔曲线（.bzc）时，你将在屏幕上看到曲线的控制顶点，如下图所示。



对于此部分，按键对应的功能说明如下。

按键	响应
E	执行一次 `BezierCurve::evaluateStep(...)`，并循环遍历各个层次
C	切换是否显示完整的贝塞尔曲线

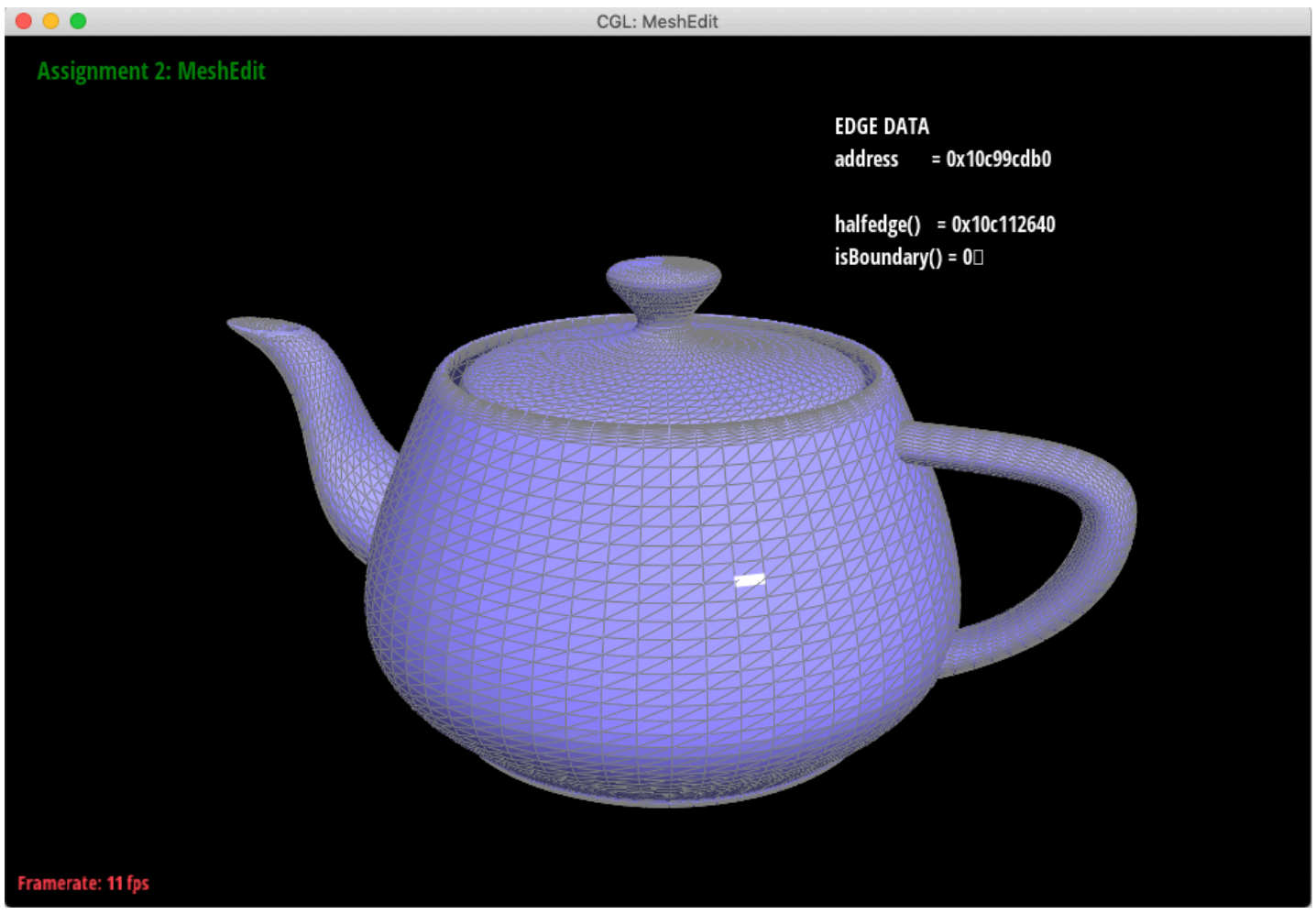
要验证你的实现是否正确，可以反复按 E 来循环遍历你编写的 de Casteljau 算法，以解析的每个级别。你可以按 C 切换显示贝塞尔曲线和控制点，以检查它是否根据控制点正确生成。

你还可以使用鼠标来：

- **单击并拖动鼠标**来移动控制点，查看贝塞尔曲线以及所有中间控制点如何相应变化。
- **滚动鼠标**逐渐生成贝塞尔曲线上的点，查看中间控制点是如何移动的。此操作本质上是改变了曲线变量 t，其取值范围介于 0.0 和 1.0 之间。

算法2-6

从算法2（贝塞尔曲面）开始，运行 `meshedit`，你将在屏幕上看到三角网格，如下所示。



当鼠标停留在屏幕上时，你会注意到鼠标下的网格元素（半边、顶点、边和面）以紫色或白色突出显示，如上图所示的白色边。当你单击其中一个元素时，其将被选中，然后 GUI 将显示有关该元素及其相关数据的一些信息。

以下是此部分的按键完整说明。

按键	功能
Q	切换显示顶点法线（算法3）
F	翻转选中的边（算法4）
S	分割选定的边（算法5）

按键	功能
`L`	对当前网格进行上采样（算法6）
`N`	选择当前半边的下一条半边（next half-edge）
`T`	选择当前半边的孪生半边（twin half-edge）
`W`	切换显示线框结构（线框结构指的是不包含多边形网格的内部）
`I`	切换显示信息叠加
`SPACE`	将相机重置为默认位置
`0`-`9`	在GLSL着色器之间切换
`R`	重新编译着色器

你还可以使用鼠标来：

- **单击并拖动顶点**来改变其位置。
- **单击并拖动背景**或**右键单击并拖动任意位置**以旋转相机。
- **滚动**来调节相机焦距（远近）。

你将分别实现 算法3、算法4、算法5和算法6中的面积加权求法向方法（`Q`）、局部边翻转（`F`）、局部边分割（`S`）和曲面细分（`L`）。这四个功能对应的按键命令在你完成各自部分之前不会执行任何操作。

代码结构

在开始编码之前，这里有一些关于初始代码结构的基本信息。

对于贝塞尔曲线和曲面（模块一），你将填写 `BezierCurve` 和 `BezierPatch` 类的成员函数，定义在 `bezierCurve.h` 和中 `bezierPatch.h` 。

对于三角形网格（模块二），你将填写 `Vertex` 、 `HalfedgeMesh` 和 `MeshResampler` 类的成员函数，定义在 `halfEdgeMesh.h` 。

我们已经在 `student_code.cpp` 中为需要编码的所有函数添加了虚拟定义。你将在此文件中实现此作业的所有部分！

模块一：贝塞尔曲线和曲面

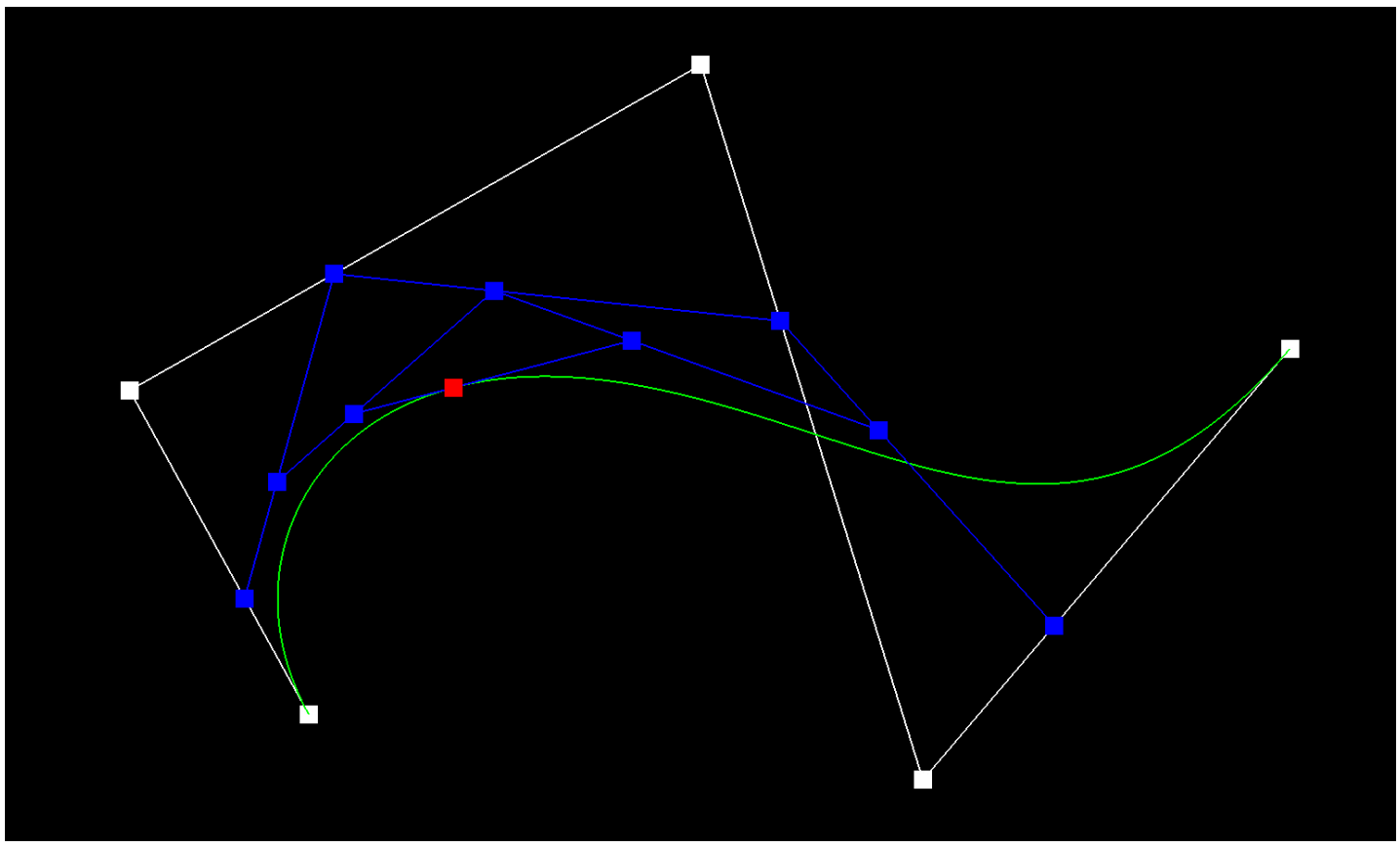
在计算机图形学中，贝塞尔曲线和曲面经常用于建模平滑且可无限缩放的曲线和曲面，例如在Adobe Illustrator（曲线建模软件）和Blender（曲面建模软件）中。

次数为 n 的贝塞尔曲线包含 n+1 个控制点。它是基于单个参数 t 的参数曲线，取值介于0和1之间。

类似地，(n,m) 次贝塞尔曲面有 (n+1 x m+1) 个控制点。它是有两个变量 (u,v) 的参数曲面，取值范围均在0和1之间。

在算法1中，你将使用 de Casteljau 算法对任何给定的控制点和参数集，计算贝塞尔曲线和曲面。例如下图的贝塞尔曲线，在图形中，白色方块是给定的控制点，蓝色方

块是 de Casteljau 算法在给定参数 t 计算的不同层次的中间控制点，红色方块是计算得到的贝塞尔曲线上的点。当 t 取尽0到1的值，则得到绿色轨迹。



算法1：利用 1D de Casteljau 实现贝塞尔曲线

在算法1中你将实现贝塞尔曲线。首先，查看`bezierCurve.h`并检查类中定义的变量。具体来说，你将主要使用以下内容：

- `std::vector<Vector2D> controlPoints`： `std::vector`定义贝塞尔曲线的原始控制点，由输入的贝塞尔曲线文件（`.bzc`）进行初始化。
- `float t`：贝塞尔曲线的参数，范围在0到1之间。

de Casteljau 算法的递归步骤：

给定 n 个控制点 $p_1, \ldots, p_n$ 及参数 t， 使用线性插值计算 $n-1$ 个中间控制点，$p'_1, \ldots, p'_{n-1}$，其中

$$p'_i = lerp(p_i, p_{i+1}, t) = (1-t)p_i + tp_{i+1}$$

通过递归地应用这个步骤，我们收敛到达一个最终的单点，而这个点实际上位于给定参数的贝塞尔曲线上。

你需要在`BezierCurve::evaluateStep(...)`中实现此递归步骤，见文件`student_code.cpp`。函数输入类型为`std::vector`的2D点，并输出类型为`std::vector`的中间控制点。 注意t是类`BezierCurve`的成员变量，你可以在函数内访问它。

编程细节提示

`std::vector`类似于 Java 的`ArrayList`类。你应该使用`push_back(...)`方法将元素添加到`std::vector`。这个类似`ArrayList`中`append(...)`方法。你可以查看此页面以获取有关的更多信息`push_back(...)`。

代码实现检验

要检查你的实现是否正确，你可以使用`meshedit`打开贝塞尔曲线文件（`.bzc`），并在屏幕上查看生成的贝塞尔曲线。按键控制请参见此处算法1 - 贝塞尔曲线。

例如，你可以运行以下命令：

```
./meshedit ../bzc/curve1.bzc
```

其中`bzc/curve1.bzc`是三次贝塞尔曲线。`bzc/curve2.bzc`是4阶贝塞尔曲线。你可以自由创建自己的`.bzc`文件并探索其他阶次的贝塞尔曲线。

算法2：基于 1D de Casteljau 实现贝塞尔曲面

在算法2中，你将把贝塞尔曲线的实现代码改编为贝塞尔曲面代码。首先，查看`bezierPatch.h`并检查类中定义的成员变量。具体来说，你将主要使用以下内容：

- `std::vector<std::vector<Vector3D>> controlPoints`： `std::vector`存储了网格的 n x n 个输入控制点。`controlPoints`的第一维大小为n， 其每个元素亦是`std::vector`类型，即`controlPoints[i]`包含有n个元素。

算法的输入为 $n \times n$ 个控制点， P_{ij}，其中 i 和 j 分别为行和列下标，以及输入两个参数 u 和 v.

首先，对于每 **行** 的 n 个控制点，$P_{i0}, \ldots, P_{i(n-1)}$，其定义了一条贝塞尔曲线，曲线参数为$u$。与算法1类似，我们使用递归算法计算出$P_i$ 。然后，所有行的P_i 构成了一组控制点，对这组顶点再次调用递归算法计算贝塞尔曲线，既得贝塞尔曲面。第二次计算使用的参数为v，得到的点为P。 on this Bezier curve at the parameter v。点P既位于贝塞尔曲面参数为u 和 v的位置。

你需要实现的函数如下，都定义在`student_code.cpp`中:

- `BezierPatch::evaluateStep(...)`：与`BezierCurve::evaluateStep(...)`非常相似，此递归算法的输入为一组3D坐标，类型为`std::vector`，另一输入为参数t，输出为3D坐标（多个递归层次，最后一层输出一个坐标，而中间层次的输出为多个中间控制点），输出类型为`std::vector`，取不同的参数t将计算曲线上不同的点。
- `BezierPatch::evaluate1D(...)`：此函数的输入是类型为`std::vector`的3D坐标和参数t，输出为曲线上的点，注意与前一函数的区别，此函数直接输出最终结果，不输出中间控制点。具体实现此函数时，你将调用`BezierPatch::evaluateStep(...)`。
- `BezierPatch::evaluate(...)`：此函数输入两个参数u 和 v，然后输出位于此参数下的贝塞尔曲面上的点。$n \times n$ `controlPoints`定义在类`BezierPatch`中，可直接访问。

代码实现检验

要检查你的实现是否正确，你可以运行`meshedit`，参数为贝塞尔曲面文件(.bez)路径，你将可以在屏幕上查看生成的贝塞尔曲面。

例如，你可以运行以下命令，你应该会看到一个茶壶：

```
./meshedit ../bez/teapot.bez
```

模块二：基于半边数据结构的三角网格上采样

提示：

- 在第二个算法模块中，你将广泛使用半边数据结构以及`HalfedgeMesh`。
- 在你深入研究本节之前，确保你已理解了半边数据结构的定义，否则请先阅读课件。
- 另外，UCB的图形学课程提供了详细的实践指南，可参考入门`HalfedgeMesh`。
- 最后，代码文档也给予了补充信息，见`halfedgeMesh.h`。

在第一个模块中，你已经实现了贝塞尔曲面，这是一种由控制点定义的二维参数曲面。除此之外，我们也可以使用三角形网格来表示曲面。虽然贝塞尔曲面比三角形网格更适合表示平滑曲面，并且需要的内存更少，但是贝塞尔曲面绘制起来更麻烦。大多数情况下，贝塞尔曲面会先转换为三角形网格，然后再在屏幕上显示。

因此，三角形网格大多数时候是计算机图形学中表示 3D 几何模型的最佳方式。存储三角形网格的一种方法将其存储为顶点列表和索引顶点的三角形列表，但是此方法对于查询和删减操作代价较高，在课程中，我们也介绍了关联矩阵表示，顶点与边的关联矩阵和边与三角形面的关联矩阵，这些矩阵可以以链表形式实现，所以查询和删减的复杂度都是O(1)。但是此方法需要存储较多冗余信息。半边数据结构明确的存储了网格元素（即顶点、边和面）之间的连接信息，同时具有操作容易和内存占用低等优点。在本模块中，你将使用半边数据结构来实现常用的几何操作。

算法3：加权面积法求顶点法线

此算法中，已知三角形的法向，你将计算三角形网格的顶点法向，具体方法为对顶点相邻的多个三角形的法向进行加权平均，而权重为三角形面积。在我们讲Shading时知道每个像素的明暗程度与物体表面的法向相关，顶点法线被用于Phong Shading，它比Flat Shading提供更好的着色效果，后者仅使用三角形法向，较为粗糙。。

首先，关于顶点类`Vertex`，其定义在文件`halfedgeMesh.h`，可先阅读头文件中的注释，了解顶点类的基本构造和用途。简单来说，`Vertex`对象封装了一个网格顶点，并具有以下成员变量（下述列表并未完全列举所有变量）:)：

- `Vector3D position`：该顶点的3D坐标。
- `HalfedgeIter& halfedge`：顶点包含的半边的引用。
- `HalfedgeCIter halfedge`：与上面一样的半边，只是该变量是`const`（即常量）并且不能被修改。

要计算给定顶点的面积加权法线，你将使用半边数据结构遍历与该顶点相关的面（三角形），相关指的是三角形包含了该顶点。对于每个这样的面，需要计算其面积，把面积当权重（需归一化），求三角形法线的加权平均数。

具体实现时，你需要在`Vertex::normal()`。函数中实现此部分，同样只需在`student_code.cpp`完成。此函数的输入是空参数，输出顶点法线。由于`normal()`是`Vertex`的成员函数，你可以在函数中访问类的成员变量，例如`Vector3D position`。至于如何遍历顶点相关的所有三角形，请参阅课件。这个半边数据结构入门教程则更贴近代码实现。为了获得每个面的法线，你将需要知道面的三个顶点，这三个点可构成两个向量，向量的叉积即得到垂直向量，也就是三角形的法线，如果对向量运算和运算的几何/物体含义不熟悉，可能需要再温习下向量代数知识（vector algebra）。

编程细节提示

你正在实现一个`const`（即常量）成员函数，这要求此函数内的任何代码都不能修改任何成员变量。这意味着你应该使用`HalfedgeCIter`而不是`HalfedgeIter`。

关于`Halfedge *`，`HalfedgeIter`及`HalfedgeCIter`之间的差异，请参阅迭代器与指针`HalfedgeCIter`简文。

你应该会使用到CGL库进行向量运算，CGL的API和实例可参考CGL向量操作文档。

代码实现检验

为了检查你的实现是否正确，你可以运行`meshedit`，载入COLLADA网格文件(`.dae`)。加载模型后，按`Q`键切换显示法线，即计算`Vertex::normal()`并使用计算得到的法线，此时模型的明暗应该变得更平滑（smooth），而不再仅是均匀（flat）。我们经常讲函数/信号的平滑性（smoothness）、平坦性(flatness)、稳定性（stableness）等，是对其几何外观的一种直觉描述，此类描述也可以用更形式化的数学量来表示——一般是与函数的若干阶微分相关的一个量。另外，按键控制的完整说明见前面表格。

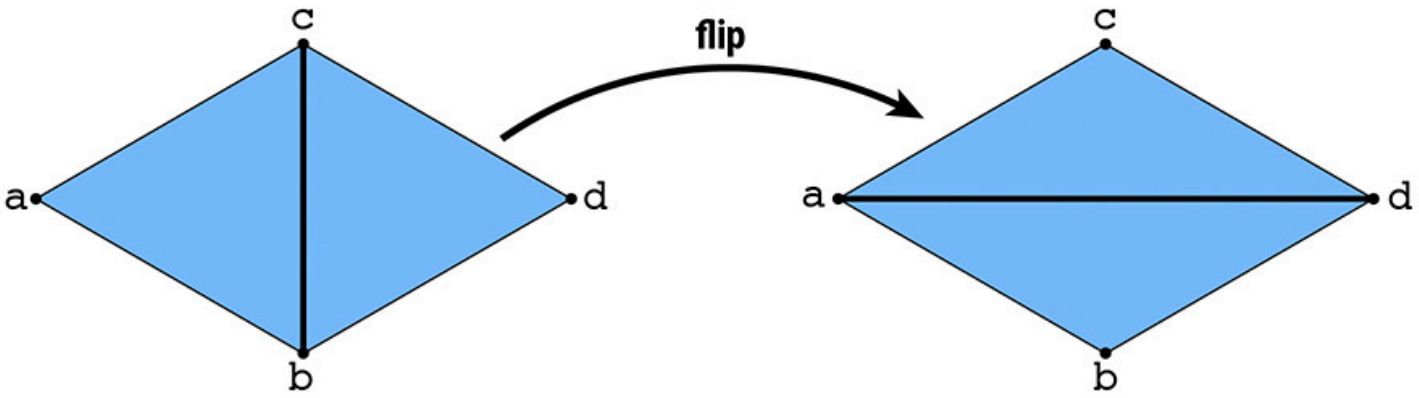
例如，你可以运行以下命令：

```
./meshedit ../dae/teapot.dae
```

按下`Q`后，茶壶的明暗应该像下图一样平滑。如果你实现效果的明暗与图像不同，那么你的`Vertex::normal()`实现可能不正确。一个常见的错误是计算出的法线指向模型的内部，而不是计算机图形学中大家默认的指向外部。若法向朝向内部的话，光线与顶点法线的夹角大于90度(小于180度)，因而余弦小于0，使用blinn phong反射模型计算漫反射和镜面反射时，光源到达物体表面的能量变为0（max(0,...)），绘制出来是全暗的效果。修复此问题可简单把法向方向取反。

算法4：边翻转（Edge Flip)

此算法将对边进行翻转操作实现局部的重网格化任务。给定一个三角形（a,b,c）和（c,b,d)，在其共享的边bc进行翻转操作，将原来一对三角形转换成新的两个三角形（a,b,d）和（c,a,d)，如下图所示：



你需要在`HalfedgeMesh::flipEdge(...)`中实现此算法。此函数输入类型为`EdgeIter`的边，例如边bc，然后输出边ad，同样为`EdgeIter`类型。

为了正确执行任何重新网格化操作，你需要格外小心，以确保在修改后的网格中，所有网格元素的所有指针仍然指向正确的元素。网格元素是半边、顶点、边或面。下面是每个网格元素指向内容的总结。我们建议你参考以下步骤，以确保在重网格化操作后所有元素的所有指针仍然有效：

1. 绘制一个简单的网格，如上面的两个三角形（a,b,c）和（c,b,d），并手写出此网格中的所有元素（即半边、顶点、边和面）的列表。
2. 在重网格化操作后的绘制网格，并再次写下修改后的网格中的所有元素的列表。
3. 如果重新网格化操作在修改后的网格中添加了新元素，则创建这些新元素。边翻转**不会**添加新元素，但下一个算法的边分割**会添加**新元素。
4. 对于修改后的网格中的每个元素，将其所有指针设置为修改后的网格中的正确元素，**即使**指向的元素没有实际改变：
    - 对于每个顶点、边和面，设置其`halfedge`指针。
    - 对于每个半边，将其`next`、`twin`、`vertex`、`edge`和`face`指针设置为正确的元素。你可以使用`Halfedge::setNeighbors(...)`函数一次设设置半边的所有指针。
    - 我们建议设置修改后的网格中所有元素的所有指针，而不仅仅是已更改的元素，因为很容易遗漏指针并导致难以调试的错误。一旦确定你的实现有效，你可以根据需要删除不必要的指针分配。

你的`HalfedgeMesh::flipEdge(...)`实现应该执行以下操作：

- 我们输入的网格是Manifold的，其可能是有边界的，也就是open-boundary manifold，边界上的边的度为1。所以切勿翻转网格边界上的边，因为此边仅与1个三角形相依。如果边的任一相邻面位于边界上，则该函数应立即返回。判断每个网格否位于边界上，可使用`isBoundary()`函数，如果元素位于边界环上，则该函数返回 true。
- 边翻转操作的复杂度是O(1)，计算和操作量是恒定的，不随网格的大小而变化。
- 不要添加或删除任何网格元素。边翻转前后的**元素**数量应完全相同，你只需重新分配指针。

编程细节提示

给定任何网格元素，你可以使用提供的`check_for(...)`调试函数检查网格中哪些其他元素指向它。例如，你可以使用此函数确认元素是否被正确数量的其他元素指向。请参考`HalfedgeMesh`入门指南的“调试辅助”部分以获取更多信息。

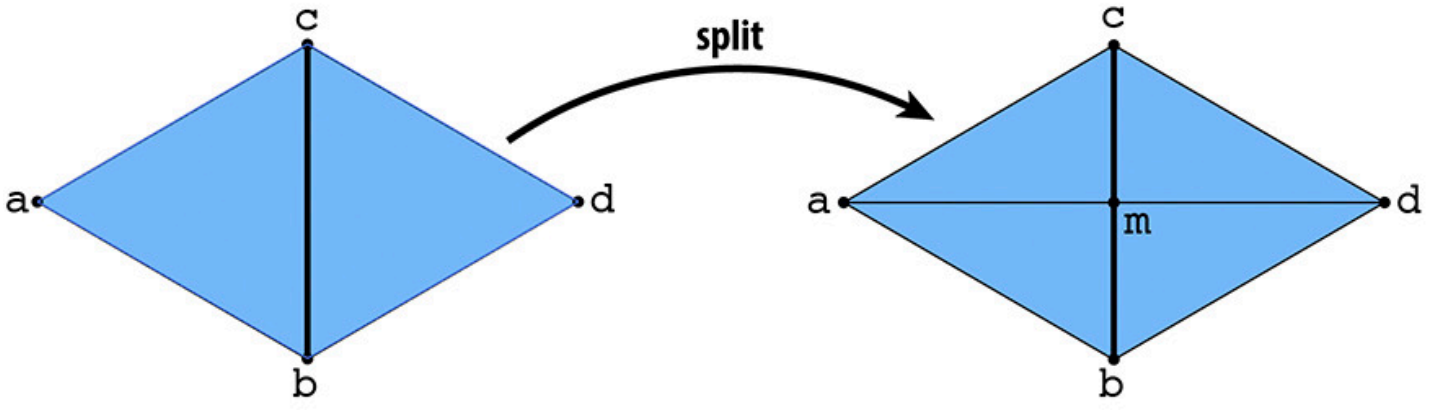
代码实现检验

加载`.dae`文件后，你将看到对应的三维网格。你可以单击选择一条边，然后按`F`键将其翻转。键盘与功能对照见控件列表。有时，当你只执行一次边翻转操作时，你的

实现似乎没问题。但建议你多执行几次，多换几条边，以确保你的代码确实按预期工作，在算法6中，你会使用到边翻转函数。

算法5：边分割（Edge Split）

此算法实现另一种局部重新网格化操作，既对边进行分割。给定一对三角形（a,b,c)和（c,b,d)，在它们的共享边 bc 上进行分割操作，可在边的中点插入一个新顶点 m，并将 m 与对角顶点 a 和 d 相连 ，得到如下图的四个三角形：



你需要在`HalfedgeMesh::splitEdge(...)`函数中实现此算法。此函数将需要分割的边（如上面的边 bc ）作为输入，类型为`EdgeIter`，并将新插入的顶点（例如上面的顶点 m ）作为输出，类型为`VertexIter`。

由于边分割会添加新的网格元素，包括新的顶点、两个新的三角形、三个新的边等，因此你需要跟踪更多的指针，并且会发现边分割比边翻转更难实现。我们鼓励你再次遵循算法4中的建议，以确保所有网格元素的所有指针都指向修改后的网格中的正确元素。

实现`HalfedgeMesh::splitEdge(...)`应该执行以下操作：

- 暂时先不考虑边界边，如果边的任一相邻面位于边界上，则该函数可以立即返回。请注意，与前一算法不同的是，分割边界上的边是合理的，而翻转边界上的边是无意义的。如果你完成了基本要求，可以考虑实现边界边分割功能。
- 新顶点的位置为输入边的中点。如前算法3所述，类。`Vertex`有一成员变量`Vector3D position`。
- 分割单个边仅需要执行恒定量的工作量，其时空成本不与网格大小成比例。

代码实现检验

加载`.dae`文件后，屏幕将绘制三角网格，你可以单击选择一条边，然后按`S`键将其分割。请参阅查看按键说明。要验证你的实现是否正确，你可以翻转一些已分割的边，然后分割一些已翻转的边。你还可以在相近和相距较远的网格区域中多次交替翻转和分割边，并检查网格是否正确更改。

算法6：网格上采样（Loop Subdivision）

有时，我们可能希望将粗糙的多边形网格转换为更高分辨率的网格，以便更好地显示、更精确地模拟等。这种转换需要一种上采样算法，能够很好地对原始网格数据的进行插值或近似。

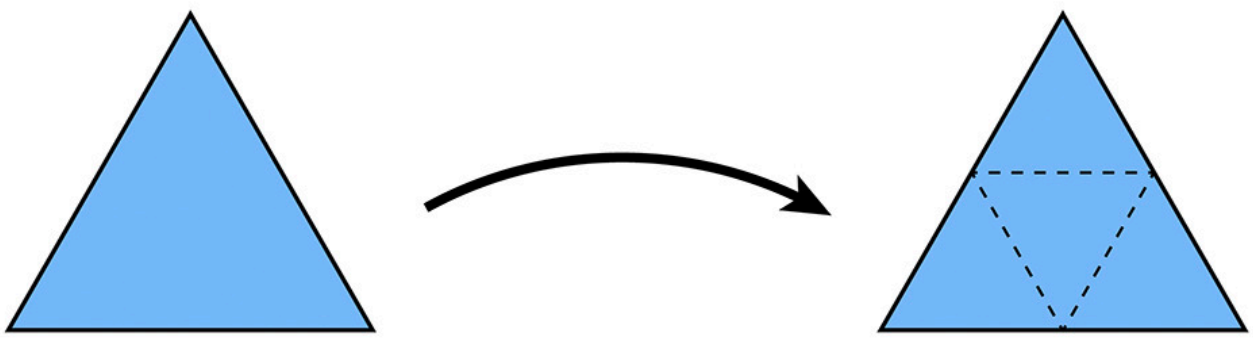
我们之前接触到简单的二维图像（如纹理图像）上采样的技术（如双线性滤波、双三次滤波等），这类技术不能简单地拓展到三维网格的上采样方法上。主要原因是网格的顶点通常位于不规则的位置，而非图像那样每个像素位于规则排列的格子上。

此算法你将实现一种网格上采样方法，称为**Loop Subdivision**。简而言之，loop subdivision通过以下方式对网格进行上采样：（1）网格的每个三角形细分为四个较小三角形和（2）根据某种加权方案更新细分网格的顶点。

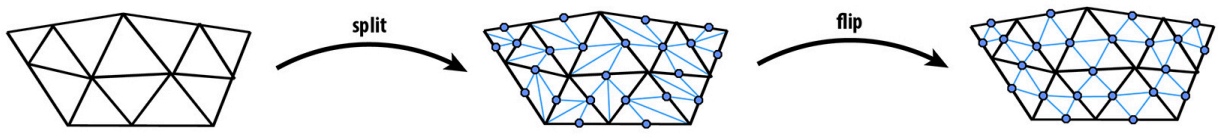
你需要在`MeshResampler::upsample(...)`中实现loop subdivision的两个步骤。此函数没有输出，输入为待细分的网格`HalfedgeMesh`，输入采用引用形式，直接对输入数据进行修改。

执行一次Loop subdivision包括以上两个步骤。多次执行loop subdivision可渐渐地细分曲面，得到与原始网格近似的平滑网格。具体而言，两个步骤的细节如下：

1. 通过连接三条边的中点将网格中的每个三角形细分为四个，如下图所示。这称为**4-1细分**。



要对整个网格执行 4-1 细分，可以使用已实现的边翻转和边分割操作。具体来说，您可以执行以下操作：

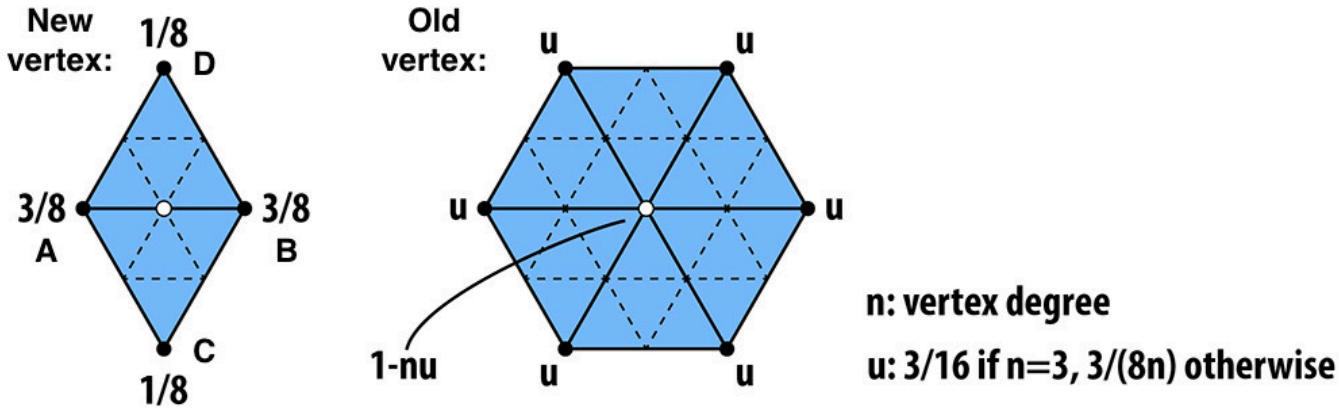


(1) 分割网格的每个现有边；(2) 将连接旧顶点和新顶点的任意新边翻转。

提示：在 (1) 之后，每个原始边现在将由 2 条边表示。你不应该**翻转**这些边，**只**应翻转连接旧顶点和新顶点的蓝色边。不可以**翻转**任何黑色的新边。

2. 将顶点位置更新为相邻顶点位置的加权平均值。每个新顶点位置都可以根据原始顶点位置计算得出，如下图所示。注意区分两类顶点，一种是新加顶点，另一种是原始顶点。

下图显示了我们用于（1）计算新添加顶点的位置或（2）更新现有顶点位置的权重。新顶点和旧顶点在各自的网格中用白色圆圈表示。



（1）对于位于一对三角形（A,C,B）和（A,B,D）交界边 AB 上的新顶点，其更新方法如下：

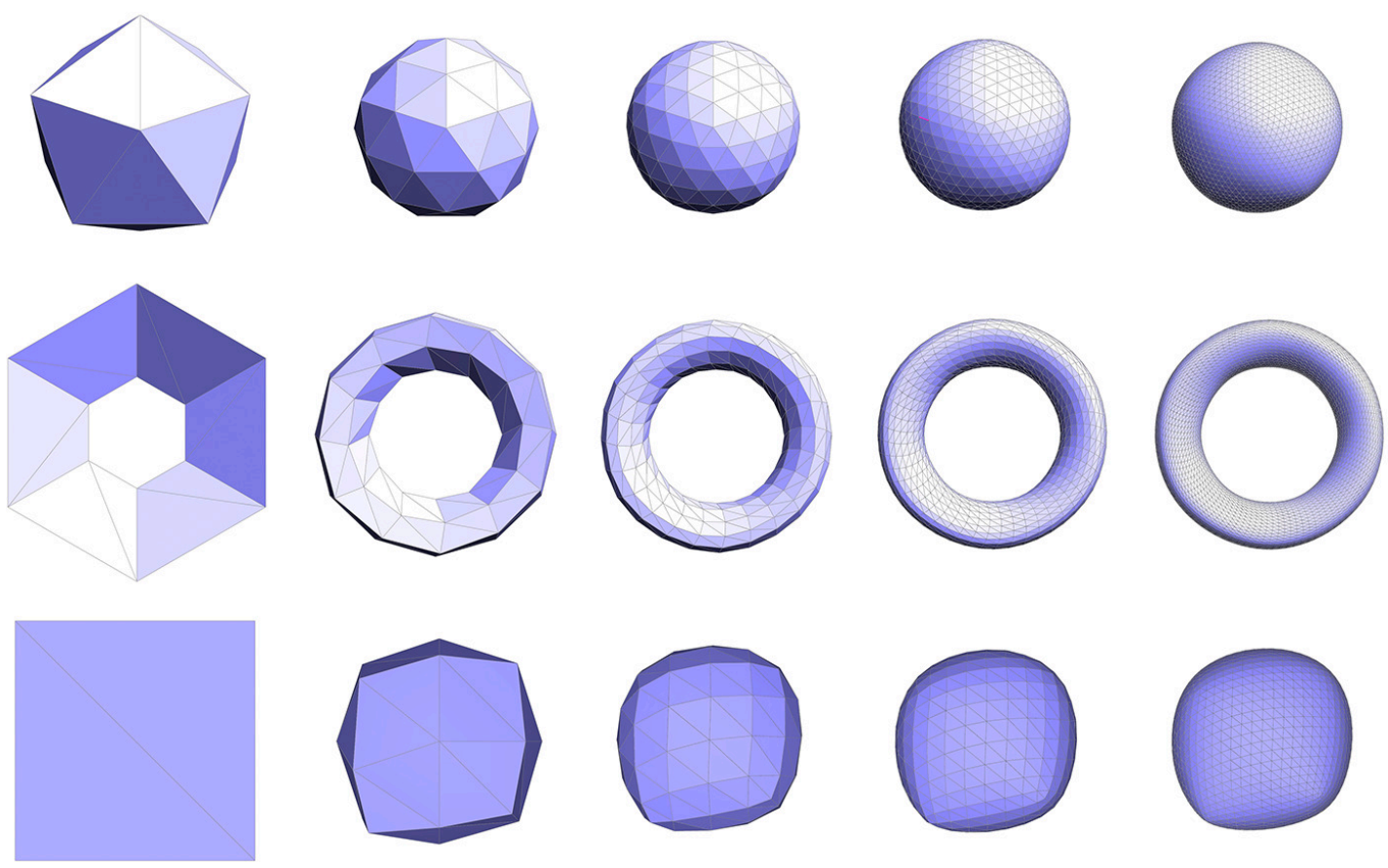
```
3/8 * (A + B) + 1/8 * (C + D)
```

（2）原始顶点的位置更新为

```
(1 - n * u) * original_position + u * original_neighbor_position_sum
```

其中`n`是顶点度数，即与顶点关联的边数，`u`是如图所示的顶点，`original_position`是该顶点的**原始**位置，而`original_neighbor_position_sum`是原始顶点的相邻顶点的位置的总和。

下面的例子显示了正确的Loop细分结果：

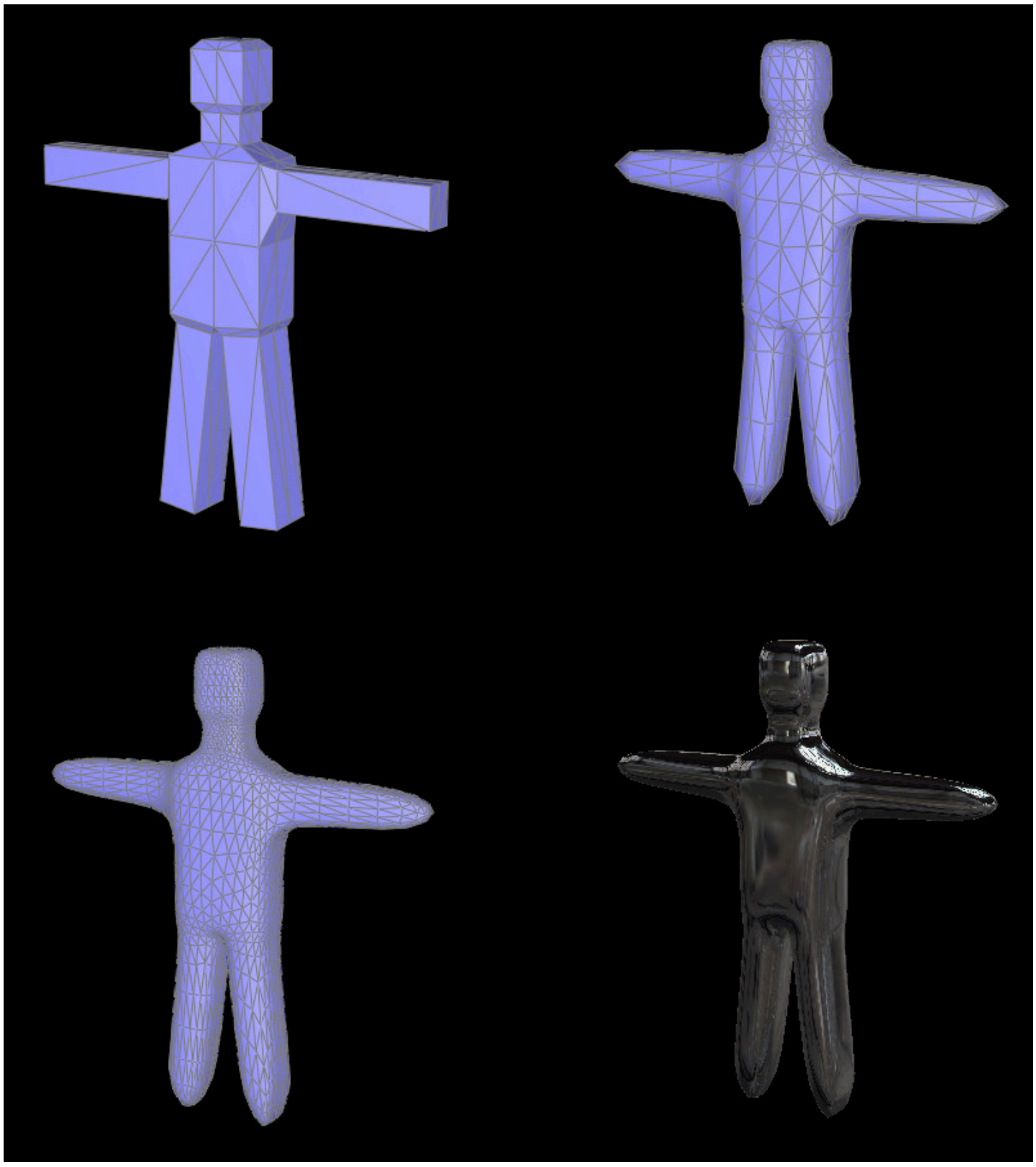


扩展学习： Loop细分有很多替代方案，针对三角形的细分方案包括改进的Butterfly方案和$\sqrt{3}$细分；四边形网格最流行的细分方案之一是Catmull-Clark。还有一些特殊的细分方案，例如用于处理包含高顶点度数的网格，如polar细分。

拓展实践

你可以使用免费程序Blender设计自己的网格模型，把其保存为COLLADA网格文件(`.dae`)，作为本作业算法的输入。作为入门学习，我们建议设计带有一个头、两条腿和两支手的人形网格。以下链接提供了使用视频教程，指导你完成制作简单的人形网格的基础知识。

设计好的网格保存为`.dae`文件后，你应该将其加载到`meshedit`程序中，并执行网络细分算法得到平滑网格。下图显示了简单人形网格和细分后效果，以及应用了环境贴图进行着色的效果。



项目调试

为了确保边分割、翻转和细分得到的网格数据结构是准确的，请参考以下调试代码（可直接复制出来，被遮挡代码将可见）。

```
#define ASSERT(a) if (!a) {printf("%s:%i:Assertion failed: %s \n Termina
namespace Debug
{
	static bool is_closed(std::vector<HalfedgeIter>& a_halfedges) {
		ASSERT(a_halfedges.size() > 1);
		for (int i = 0; i < a_halfedges.size() - 1; i++) {
			auto it = a_halfedges[i];
			if (it->next() != a_halfedges[i + 1]) {
				return false;
			}
		}
		if (a_halfedges[a_halfedges.size() - 1]->next() != a_ha
			return false;
		}
		return true;
	}
// make sure that all the half edges form a closed surface
#define CHECK_CLOSED(...) ASSERT(Debug::is_closed(std::vector<HalfedgeI
	static bool is_face_of(FaceIter a_face, std::vector<HalfedgeIte
		for (auto i : a_halfedges) { // check ptr from edge to
			if (i->face() != a_face) {
				return false;
```

```
                        }
                }
                auto it = a_face->halfedge(); // check ptr from face to
                do {
                        if (std::find(a_halfedges.begin(), a_halfedges.
                                return false;
                        }
                        it = it->next();
                } while (it != a_face->halfedge());
                return true;
        }
// make sure that the face consist of the half edges
#define CHECK_FACE(face, ...) ASSERT(Debug::is_face_of(face, std::vecto
        static bool is_vertex_of(VertexIter a_vertex, std::vector<Halfe
                ASSERT(a_halfedges.size() > 0)
                for (auto i : a_halfedges) { // check ptr from edge to
                        if (i->vertex() != a_vertex) {
                                return false;
                        }
                }
                auto it = a_vertex->halfedge();
                do {
                        auto f = std::find(a_halfedges.begin(), a_halfe
                        if (f != a_halfedges.end()) {
                                a_halfedges.erase(f); // remove found e
                        }
                        it = it->twin()->next();
                } while (it != a_vertex->halfedge());
                return a_halfedges.size() == 0;
        }
// make sure that the vertex is connected to the half edges
#define CHECK_VERTEX(vertex, ...) ASSERT(Debug::is_vertex_of(vertex, st
}
```

使用调试断言的示例：

```
VertexIter HalfedgeMesh::splitEdge(EdgeIter e0)
{
        HalfedgeIter h0, h1, h2, h3, h4, h5, h6, h7, h8, h9;
        VertexIter v0, v1, v2, v3;
        EdgeIter e1, e2, e3, e4;
        FaceIter f0, f1;
        ...
        ...
        // sanity check
        // check that half edges are closed
        CHECK_CLOSED(h3, hb, h5)
        CHECK_CLOSED(h4, hy, ha)
        // check that faces are made of the given half edges
        CHECK_FACE(f1, h3, hb, h5)
        CHECK_FACE(f0, hz, h0, h1)
        // check that vertices are connected to the given half edges
        CHECK_VERTEX(v0, h7, hx, h4)
        CHECK_VERTEX(vx, h0, hb, ha, hc)
}
```