Lecture 06 - searching

Philip Caplan

CSCI 0422 - Geometric Modeling (Spring 2022)
Learning objectives

By the end of this lecture you will be able to:

- practice with some flux conventions,
- calculate element-to-element adjacency information,
- extract the unique list of edges in a mesh,
- practice with linear algebra in flux,
- search for a triangle in a mesh with some property.
Getting started...

Switch Host & Client today!

```
$ git pull
$ make update
$ cmake .
$ make template_class06_searching
```

Compiling and running the exercise:

```
$ make class06_searching
```

Compiling and running the solution (after class):

```
$ make class06_searching_sol
```
Now that we’ve created some meshes, here are some conventions.

- \( \text{nb}() \): returns the total \textbf{Number} of elements (i.e. the number of vertices or the number of triangles).

- \( \text{count}(k) \): returns the number of entries stored for element \( k \). For data which is stored in a rectangular manner, such as vertex coordinates or triangle indices, this will always be the stride (the dimension for vertices, and 3 for triangles).

- \( \text{operator}(k,j) \): allows you to read/write the \( j \)-th entry in element \( k \). For example, for vertices stored in vertices, then vertices(20,2) will access the \( z \)-coordinate of the 21st vertex.

- \( \text{operator}[]() \): allows you to read/write the data stored at the memory location for element \( k \). In other words, this function returns a pointer to first entry of element \( k \).

we will practice with these conventions today!
Now that we’ve created some meshes, here are some conventions.

- \texttt{nb()}: returns the total \textbf{N}umber of elements (i.e. the number of vertices or the number of triangles).
- \texttt{count(k)}: returns the number of entries stored for element \textit{k}. For data which is stored in a rectangular manner, such as vertex coordinates or triangle indices, this will always be the stride (the dimension for vertices, and 3 for triangles).

**we will practice with these conventions today!**
Now that we’ve created some meshes, here are some conventions.

- `nb()`: returns the total **Number** of elements (i.e. the number of vertices or the number of triangles).
- `count(k)`: returns the number of entries stored for element `k`. For data which is stored in a rectangular manner, such as vertex coordinates or triangle indices, this will always be the stride (the dimension for vertices, and 3 for triangles).
- `operator()(k,j)`: allows you to read/write the `j`-th entry in element `k`. For example, for vertices stored in `vertices`, then `vertices(20,2)` will access the z-coordinate of the 21st vertex.

we will practice with these conventions today!
Now that we’ve created some meshes, here are some conventions.

- \texttt{nb()}: returns the total \texttt{Number} of elements (i.e. the number of vertices or the number of triangles).

- \texttt{count(k)}: returns the number of entries stored for element \texttt{k}. For data which is stored in a rectangular manner, such as vertex coordinates or triangle indices, this will always be the stride (the dimension for vertices, and 3 for triangles).

- \texttt{operator()(k,j)}: allows you to read/write the \texttt{j}-th entry in element \texttt{k}. For example, for vertices stored in \texttt{vertices}, then \texttt{vertices(20,2)} will access the \texttt{z}-coordinate of the 21st vertex.

- \texttt{operator[](k)}: allows you to read/write the data stored at the memory location for element \texttt{k}. In other words, this function returns a pointer to first entry of element \texttt{k}.

\textit{we will practice with these conventions today!}
Commonly used element shapes: in src/flux-base/src/element.[h,cpp]

```cpp
/**
 * \brief Triangle element:
 * *
 * 2
 * |      
 * | \    
 * face/ | \  face/edge 1
 * edge  | \ 
 * 2     | t0  
 * 0 ---- 1
 * face/edge 0
 */
struct Triangle {
    static const int dimension = 2;
    static const int nb_vertices = 3;
    static const int nb_edges = 3;
    static const int nb_faces = 3;
    static const int nb_triangles = 1;
    static int edges[6]; // this equals {0,1,1,2,2,0};
    static int triangles[3];
    static int faces[6]; // this also equals {0,1,1,2,2,0};
    typedef Line face_type;
};
```

**note:** we can also use i and (i+1)%3 to retrieve the vertex indices of the i-th edge (in a Triangle).
Extracting the edges and triangle-triangle adjacencies.

what are your ideas on how to do this?

hint: see starter code in exercises/class06/searching.cpp and src/flux-base/src/element.[h,cpp]
Searching for a triangle in a mesh with some property.

Problem: Find an element (a triangle) that has a vertex within a distance $R$ to some point $q$. 
Other search structures.

k-d tree
Other search structures.

- k-d tree
- quadtree
Other search structures.

- k-d tree
- quadtree
- octree
Your TODO list . . .

- examine src/flux-base/src/adjacency.[h,cpp],
- add unit tests for src/flux-base/src/adjacency.[h,cpp],
- see the documentation! (see the "docs" button at the top of course website),
- review pointers: our discussion of half-edges will involve a lot of pointers,
- work on Project 1!
Don’t forget to commit and push your changes!

Host: (assuming you are in top-level flux directory)

```bash
$ git add exercises/class06
$ git commit -a -m "added exercises from lecture 6"
$ git push
```

If you are in a build directory, the first command would be: $ git add ..//exercises/class06

Client:

```bash
$ git pull
```
An example of testing **Adjacency:**

```cpp
UT_TEST_CASE( two_triangle_test ) {
    /*
    * 1
    * 3 ------- 2
    * |  t1   /
    * |     /  
    * 2 | 0 / 2 | 1
    * |     /  
    * |     t0 |
    * 0 ------- 1
    * 0
    */
    Mesh<Triangle> mesh(2);
    double x0[2] = {0,0}; mesh.vertices().add(x0);
    double x1[2] = {1,0}; mesh.vertices().add(x1);
    double x2[2] = {1,1}; mesh.vertices().add(x2);
    double x3[2] = {0,1}; mesh.vertices().add(x3);

    int t0[3] = {0,1,2}; mesh.add(t0);
    int t1[3] = {0,2,3}; mesh.add(t1);

    Adjacency<Triangle> adj(mesh);
    UT_ASSERT_EQUALS( adj.nb() , 2 );
    UT_ASSERT_EQUALS( adj.nb_neighbors() , 3 );

    UT_ASSERT_EQUALS( adj(0,0) , -1 );
    UT_ASSERT_EQUALS( adj(0,1) , -1 );
    UT_ASSERT_EQUALS( adj(0,2) ,  1 );

    UT_ASSERT_EQUALS( adj(1,0) ,  0 );
    UT_ASSERT_EQUALS( adj(1,1) , -1 );
    UT_ASSERT_EQUALS( adj(1,2) , -1 );

    UT_ASSERT_EQUALS( adj.indexof(0,1) , 2 );
    UT_ASSERT_EQUALS( adj.indexof(1,0) , 0 );
}
UT_TEST_CASE_END( two_triangle_test )
```
Another test case?

```c
UT_TEST_CASE( triangle_test ) {
    
    typedef Triangle type;

    int n = 10;
    Grid<type> grid( {n,n} );
    Adjacency<type> adj(grid);

    UT_ASSERT_EQUALS( adj.nb() , grid.nb() );
    UT_ASSERT_EQUALS( adj.nb_neighbors() , 3 );

    int nb_boundary = 0;
    for (int k = 0; k < grid.nb(); k++) {
        for (int j = 0; j < 3; j++) {
            int n = adj(k,j);
            if (n < 0) {
                nb_boundary ++;
                continue;
            }
            int j1 = adj.indexof(k,n);
            UT_ASSERT( j1 >= 0 );
            UT_ASSERT_EQUALS( adj(k,j1) , n );
        }
        UT_ASSERT_EQUALS( adj.indexof( 0 , grid.nb() ) , -1 );
        UT_ASSERT_EQUALS( nb_boundary , /* what should this be ? */ );
    }

    UT_TEST_CASE_END( triangle_test );
```