TP5 - Working with meshes

Visualizing more complex 3D objects

Computer Graphics, 2024-2025
2nd year, Multimedia track

Session 5 and 6

Version: v2025.1.0-rc5
(develop @ df05f2c 2025-04-01)

Contents

[git] • Branch: develop @ df05f2c • Release: v2025.1.0-rc5 (2025-04-01)
Head tags: v2025.1.0-rc5

1 Objective

In this last TP we will see one more aspect of the OpenGL modeling by building a simple viewer of 3D meshes defined in a standard format, the OBJ format. This will allow us to introduce a more efficient way to draw the geometric primitives OpenGL and to implement a subdivision algorithm.

2 Efficient drawing in OpenGL: the vertex arrays

So far we have drawn 3D objects calling the function `glVertexf()` for each single vertex: in the case of the icosahedron, *e.g.*, we drew each triangle using 3 calls to `glVertexf()`. In general, each time we call a function we also introduce a small overhead in the computation, hence when dealing with meshes with many triangles, a huge overhead is introduced that affect the performances of the program.

OpenGL has another way to deal with large meshes, the vertex arrays. These routines allow you to specify a lot of vertex-related data with just a few arrays and to access that data with equally few function calls: *e.g.* all the vertices of the mesh can be collected into a single array and draw with a single instruction. To use the vertex arrays there are 3 main steps to follow:

1. Activate and enable the arrays, each storing a different type of data: vertex coordinates, RGBA colors, color indices, surface normals, vertex indices, texture coordinates. OpenGL provides the function `glEnableClientState()` which activates the type of array (vertices, normals, *etc.*):

```
// Specifies the array to enable. Symbolic constants GL_VERTEX_ARRAY, GL_COLOR_ARRAY,
//GL_INDEX_ARRAY, GL_NORMAL_ARRAY, GL_TEXTURE_COORD_ARRAY, and GL_EDGE_FLAG_ARRAY
//are acceptable parameters.
void glEnableClientState(GLenum array)
```

   For example, when using lighting, you may want to define a surface normal for every vertex and thus activate both the surface normal and vertex coordinate arrays:

```
glEnableClientState(GL_NORMAL_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);
```

2. Fill the array or arrays with the data by passing the addresses of (*i.e.*, the pointers to) their memory locations. In order to pass the point to the array containing the data, OpenGL provides several functions, one for each type of data. In particular, for vertices and normals the following functions are available:

```
// pointer is the memory address of the first coordinate of the first vertex in the array.
// type specifies the data type (GL_SHORT, GL_INT, GL_FLOAT, or GL_DOUBLE)
// size is the number of coordinates per vertex, which must be 2, 3, or 4.
// stride is the byte offset between consecutive vertexes (0 means that the vertices are
// defined one after another)
void glVertexPointer(GLint size, GLenum type, GLsizei stride, const GLvoid *pointer);

```


[git] • Branch: develop @ df05f2c • Release: v2025.1.0-rc5 (2025-04-01)
Head tags: v2025.1.0-rc5

```
// pointer is the memory address of the first coordinate of the first normal in the array.
// type specifies the data type (GL_SHORT, GL_INT, GL_FLOAT, or GL_DOUBLE)
// stride is the byte offset between consecutive normals (0 means that the normals are
// defined one after another)
void glNormalPointer(GLenum type, GLsizei stride, const GLvoid *pointer);
```

3. Draw geometry primitives with the data. OpenGL provides the function `glDrawElements()` (the doc) that draws all the previously defined arrays in a single shot using the indices specified as input: the indices are an array containing the indices of the vertices in the order they have to be drawn according to the chosen type of geometric primitive to render (`mode`):

```
/**
 * Specifies multiple geometric primitives in a single call
 * @param mode The kind of primitives to render (GL_POINTS, GL_LINES, GL_TRIANGLES etc)
 * @param count The total number of elements (in terms of vertices) to be rendered.
 * @param type The type of the values in indices. We'll use GL_UNSIGNED_INT.
 * @param indices The pointer to the location where the indices are stored.
 */
void glDrawElements(GLenum mode, GLsizei count, GLenum type, const GLvoid * indices);
```

4. Deactivate the arrays previously enabled by using the function `glDisableClientState()`:

```
// The opposite of glEnableClientState()
void glDisableClientState(GLenum array);
```

3 The OBJ format

The OBJ format is a geometry definition file format first developed by Wavefront Technologies for its Advanced Visualizer animation package. The file format is open and has been adopted by other 3D graphics application vendors. It represents the 3D geometry of the model in terms of vertices, faces, normals and, optionally textures and colors. In this TP we will use its simplest version in which there are no texture definitions and the normal to each face must be computed at run-time. Here is a sample of an `.obj` file:

```
# List of Vertices, with (x,y,z) coordinates
v 0.123 0.234 0.345
v ...

# Face Definitions as a list of indices of vertex (indices starts from 1!!)
f 1 2 3
f ...
  ...
```

It is composed by a list of vertices starting with the letter `v` and followed by the 3 coordinates x, y, z of the vertex. Then there is a list of faces, each face starting with the letter `f` and

followed by 3 indices of the vertices that compose the triangle. The indices start from 1 and they refer to the above list of vertices. That's all you need to know about the OBJ format for this TP. If you are interest on the full format you can refer to the on-line documentation here (wikipedia) and here.

4 The OBJ viewer

The goal of this TP is to build a simple viewer of OBJ files: the program should take as input an OBJ file, parse it and display it in an OpenGL window (*c.f.* Figure 1).

You can find several source files in the archive that has been given to you:

- `main.cpp` is the main file with the usual OpenGL pipeline. You don't have to modify anything here!.
- `core.{cpp,hpp}` contain the declaration and the implementation of some basic types and structures that (hopefully) would help with your task. You don't have to modify anything here!.
- `MeshModel.{cpp,hpp}` contain the declaration of the class `MeshModel` that contains the representation of the 3D model. Nothing to do here.
- `geomery.{cpp,hpp}` contain functions (that you have to implement) to compute the normal of a face and the angle at a vertex.
- `objReader.{cpp,hpp}` contain functions (that you have to implement) load the obj file and create the `MeshModel` structure.
- `rendering.{cpp,hpp}` contain all the functions (that you have to implement) to render the model in wire-frame, flat and smooth mode.
- `loop.{cpp,hpp}` contain the functions (that you have to implement) for applying the Loop subdivision algorithm to the model.

The class `MeshModel` reads the input OBJ file and fills up the arrays containing the vertices, their normals, and the faces as they are parsed in the OBJ file. In particular the class contains 3 main arrays:

- `_vertices` is a `vector` of `point3d` containing the vertices;
- `_normals` is a `vector` of `vec3d` containing, for each element of `_vertices`, the relevant normal;
- `_mesh` is a `vector` of `face` in which each entry contains a triplet of vertex indices forming a face of the mesh;

The three data types are defined as below. `point3d` and `vec3d` are two aliases for the same data structure that has 3 members, one for each coordinate `x`, `y`, `z`. The main basic operations are defined for this data structure, included the `norm()` function, the dot and cross product and the normalization procedure (*i.e.* the division by its norm). The `face` data structure is similar to the previous ones but its members are defined as `idxtype` which is just another redefinition of `unsigned int`.

[git] • Branch: develop @ df05f2c • Release: v2025.1.0-rc5 (2025-04-01)
Head tags: v2025.1.0-rc5

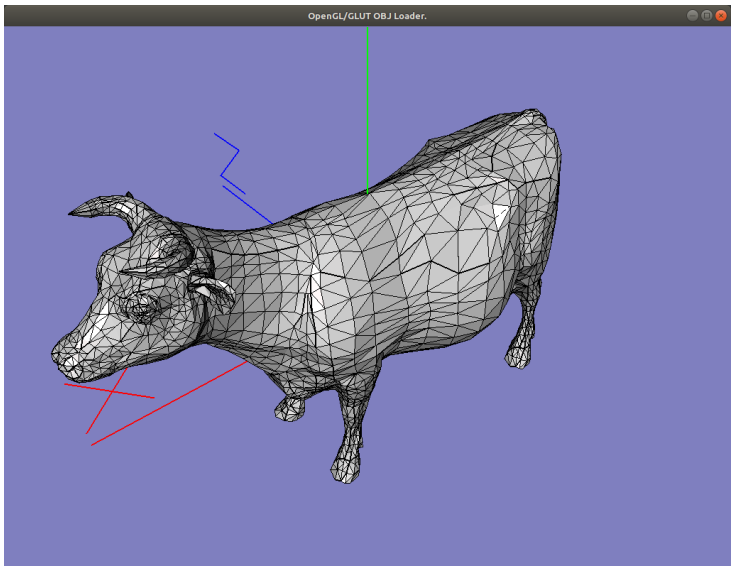


Figure 1: An example of the viewer to implement when the model in `cow-nonormals.obj` is loaded.

With all this in mind it's easy to verify that, *e.g.*, the 2nd vertex of the 10th face of the mesh is `_vertices[ _mesh[9].v2 ]` and its normal is `_normals[ _mesh[9].v2 ]`.

```
struct v3f
{
    float x; //!< the first component
    float y; //!< the second component
    float z; //!< the third component
...
}
using point3d = struct v3f ;
using vec3d = struct v3f;

using idxtype = GLuint;     // a unsigned int

struct face // a face contains the 3 indices of the vertices
{
    idxtype v1; //!< the first index
    idxtype v2; //!< the second index
    idxtype v3; //!< the third index
...
}

// Some examples of operations of point3d and vec3d

point3d pt1, pt2, pt3;
pt1 = pt2 + pt3;           // Addition and subtraction
pt1 -= pt2;                // Addition/subtraction assignment
pt1 = pt2 * pt3;           // Element-wise product/division
pt1 = pt2 * a;             // Multiplication and division with a scalar
pt1 *= a;                  // Multiplication/division assignment
vec3d v1, v2, v3
float value = v1.norm();   // L2 norm
v1.normalize();            // normalize the vector, ie divide by its norm (changing v1)
float value = v1.dot(v1);  // Dot product
```

```
33  vec3d v3 = v1.cross(v2);   // Cross product
```

5 The Mesh viewer

In this first bunch of exercises we will experiment different ways to render the 3D object: we will start with a "warm up" exercise to load the model from the file and render its wire-frame model. Then we will implement a basic flat rendering function of the model and finally we will use the vertex array approach to render the model in order to obtain a smoother surface using the OpenGL shading interpolation. Note that you will implement these functions and you will be able to switch from one rendering mode to another using the key bindings reported in Section B and recalled in each of the next sections.

As a final note, most of the code is already provided, you just need to complete the functions following the proposed trace and fill up the missing instructions after the comments surrounded by `//************************`. For debugging purpose, a macro `PRINTVAR( variable );` is defined in order to print the value of a variable, be it a single variable or a list / vector of elements.

Finally, in order to compile and execute the code check the `BUILD.md` file for the instructions of your platform. In the folder `data/models` you can find some 3D models (obj files) that you can use to test the visualizer. It is strongly suggested to start with very simple models such as `cube.obj` and `icosahedron.obj` that are easier to debug.

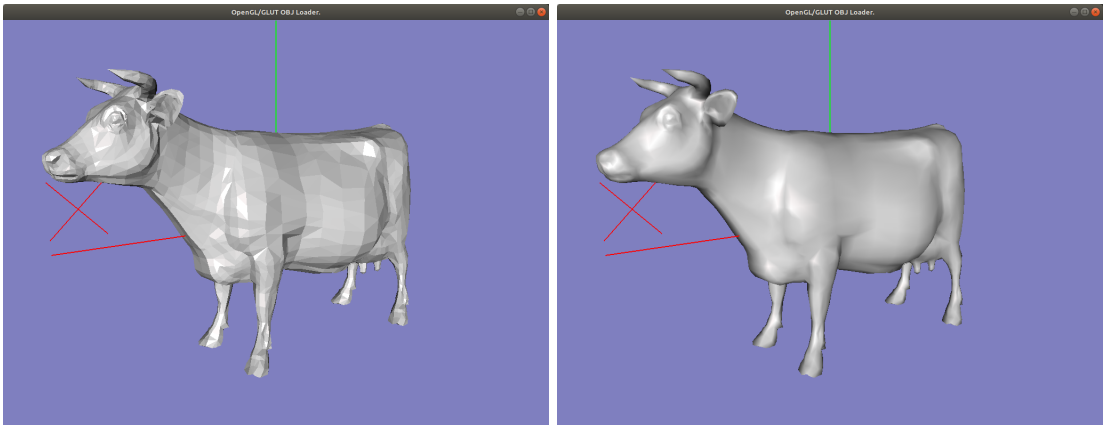
5.1 First basic wire-frame rendering

In this first exercise, we implement the function that reads and store the OBJ data into the data members of the class and a first, basic rendering of the wire-frame of the mesh.

1. complete the `load()` function in `objReader.cpp`: it parses the .obj file, for each line starting with a `'v'` it gets the 3 coordinates of the vertex and it add the new vertex to the list `_vertices`; otherwise, for each line starting with a `'f'` it gets the 3 indices of the vertices composing the face and add the triple index to the list `_mesh`.
   - Complete the two parts of the `load()` function with the relevant statements to add the input into the corresponding vectors. It may seems trivial and boring (and it is!), but it's just to get you warmed up to work with `vector` and C++.
2. complete the `drawWireframe` function `rendering.cpp`: given the vertices and the mesh as input, it draws the contour of each face
   - In order to draw the vertices you can use the usual either `glVertex3f()` function specifying the 3 coordinates or its vector (and faster) version `glVertex3fv(float *v)` which requires as input a vector of 3 `float` elements. In this case you need to pass the address to the vertex casted as a pointer to `float`, *i.e.* `(float*) &vertices[i]`, with `i` replaced by the proper vertex index that you recover from the parameter `mesh`. For a more detailed explanation about the way of passing struct-based vectors as raw pointers check the Appendix D.

Now you should see the model rendered in wire-frame mode. Note that the W allows to enable and disable the display of the wire-frame. You can move around the 3D model with the usual arrow keys and use PgUp , PgDn to zoom in/out.

[git] • Branch: develop @ df05f2c • Release: v2025.1.0-rc5 (2025-04-01)
Head tags: v2025.1.0-rc5



(a) (b)

Figure 2: The same model rendered using a single face normal for each vertex (a) and its smoother version rendered using for each vertex an averaged normal of the surrounding faces (b).

5.2 A first rendering

The next step is to implement the function that renders the mesh using the light to shade each face. In this first version we will render the face by computing the normal to each face as the normal of the plane containing its 3 vertices.

1. Complete the function `computeNormal()` in `geometry.cpp`: the function takes 3 vertices and computes the normal using the cross-product of two edges of the triangle.
2. Complete the function `drawFaces()` in `rendering.cpp`: given the vertices and the mesh as input, for each face it computes the normal to the face and draws it.
   - Again, we can use `glNormal3f()` specifying the 3 coordinates or its vector version `glNormal3fv(float *v)` that takes a vector as input. In this case if `vec3d n` is the normal vector we still need to pass its address casted to `float` *i.e.* `(float*) &n`.

As you can see the model is now rendered as in Figure 2.a, in which each triangular face is shaded using its normal.

5.3 A more efficient rendering

In the next step, we implement a more efficient way to render the model using the vertex array (see Section 2). Moreover, we would like to have more pleasant visualization of the model, in which the surface of the model appears smoother and the single triangular faces are not visible. To do so, OpenGL needs to have the normal for each vertex instead of a single normal for each face (hence common to each of the 3 vertices of the face). How can we compute a single normal to each vertex? For each vertex we can compute its normal as the average normal among all the faces it contributes to: *i.e.* for each vertex, we can sum the normals of all its faces and then re-normalize the result.

[git] • Branch: develop @ df05f2c • Release: v2025.1.0-rc5 (2025-04-01)
Head tags: v2025.1.0-rc5

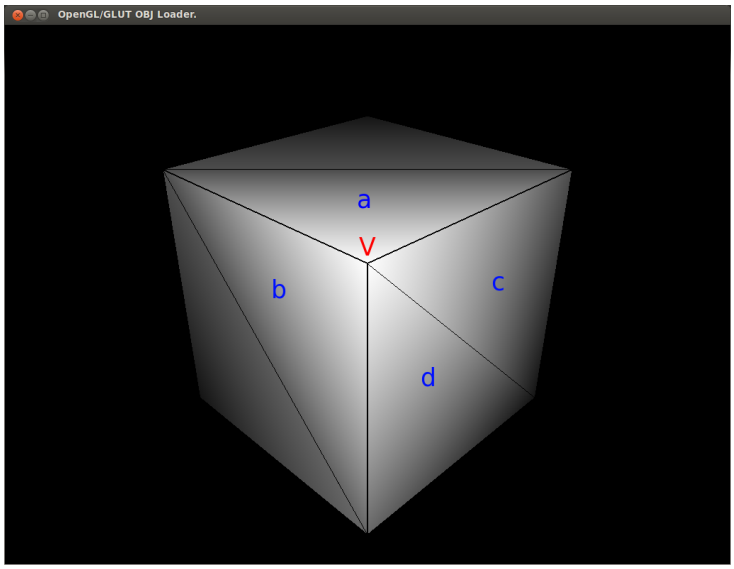


Figure 3: An example of the biased computation of the normals for the vertex V: it receives a contribution from the faces a and b and a double contribution with the same normal from the faces c and d.

1. Complete the function `load()` in `objReader.cpp`: whenever a new vertex is read set its normal to a zero vector. Then, whenever a face is read from the input file, compute the normal to the face and sum it to the normal of each vertex. At the end, when the file has been read, re-normalize all the normals.

2. Complete the function `drawArrayFaces()` in `rendering.cpp`: given the vertices, their normals and the mesh as input, draw the model using the vertex array in OpenGL as explained in Section 2. To pass the struct-based vector as raw pointer check Appendix D.

Once you have correctly implemented the function `drawArrayFaces()` you can turn this rendering on using the key S. This will switch between the `drawArrayFaces()` and `drawFaces()`. Also, the A allows to switch from the `GL_SMOOTH` to `GL_FLAT` rendering. Now you should be able to see a smoother rendering of the model when `GL_SMOOTH` and `drawArrayFaces()` are enabled.

On the other hand, this implementation is not quite satisfying for models like the cube (try it...). The computed normals are indeed biased for some vertices because they may receive a double contribution from the two triangular faces of the same cube face and only one from the the triangular face of the other cube face (see Figure 3). To solve this issue we can weight the contribution of each normal with the angle formed at the vertex.

1. Modify the way the normals are summed in function `load()`: for each vertex V of the face, the normal of the face is weighted by the extent of the angle formed by the edges at the vertex V.

   - To this end you can use the function `angleAtVertex()`:

```
1 /**
2  * Computes the angle at vertex baseV formed by the edges connecting it with the
3  * vertices v1 and v2 respectively, ie the baseV-v1 and baseV-v2 edges
```

```
4    *
5    * @brief Computes the angle at vertex
6    * @param[in] baseV the vertex at which to compute the angle
7    * @param[in] v1 the other vertex of the first edge baseV-v1
8    * @param[in] v2 the other vertex of the second edge baseV-v2
9    * @return the angle in radiants
10   */
11  float angleAtVertex(const point3d& baseV, const point3d& v2, const point3d& v3);
```

You can test the vertex rendering with the larger models that you can find in `data/models/stanford`. You should note that, in general, the frame per second (FPS, *i.e.* how many frame per second the system is able to render) is higher when the model is rendered using the vertex array approach. Try to continuously orbit around the object (disable the wireframe rendering) and zoom in/out to see the difference in terms of the FPS as displayed in the bottom-right corner of the window.

[git] ▪ Branch: develop @ df05f2c ▪ Release: v2025.1.0-rc5 (2025-04-01)

Head tags: v2025.1.0-rc5

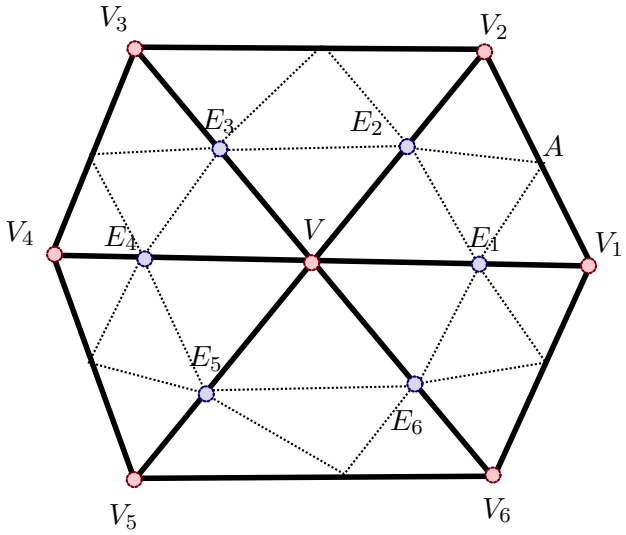


Figure 4: An example of Loop subdivision algorithm applied to a triangular mesh. Each edge of the original mesh generates a new vertex E_i, and new smaller faces are formed connecting the new vertices E_i and the original vertices V_i. Note that the vertices E_i in general DO NOT lie on the edges, here they are placed on the edge just for readability of the figure.

6 Loop's subdivision

The Loop subdivision scheme subdivides a triangular mesh surface by performing several iterations in which each (triangular) face is subdivided into 4 smaller ones. The algorithm starts with a set of points that are the vertices of triangles. Each iteration computes a new set of edge and vertex points that become the vertices of the new, smaller triangles. Specifically, a new edge point is computed for each edge and a new vertex point is computed for each vertex of the triangular mesh. If the original mesh has E edges and V vertices, the new mesh will have $E+V$ vertex points (and a number of edges that depends on the complexity of the original mesh). The new points become the vertices of the new, finer mesh, and more iterations may be applied to refine the mesh as much as needed.

Considering the simple mesh in Figure 4 centered on the vertex V, the rule to generate the new vertices E_i for each of the edge sharing V is:

$$E_i = \frac{3}{8}(V + V_i) + \frac{1}{8}(V_{i+1} + V_{i-1}), \tag{1}$$

whereas, for boundary edges such as V_1-V_2 (see Figure 4), the new vertex A is the midpoint of the two extrema of the edge

$$A = \frac{1}{2}(V + V_i). \tag{2}$$

The original vertex V is updated using the Loop's coefficients:

$$\hat{V} = \frac{5}{8}V + \frac{3}{8}\frac{1}{n}\sum_{i}^{n} V_i. \tag{3}$$

[git] • Branch: develop @ df05f2c • Release: v2025.1.0-rc5 (2025-04-01)
Head tags: v2025.1.0-rc5

6.1 The Algorithm

The algorithm to implement is relatively simple and it requires three main steps: (*i*) generate the new vertices according to (1) and (2), (*ii*) update the original vertices according to (3), and finally (*iii*) recompute the normals for all the vertices as you did in the `load()` function.

The first part (*i*) is maybe the trickiest part. Since we don't have a list of edges, we can look at each face and generate a new vertex for each of its edges. However, this requires that we keep a record of each newly generated vertex and the corresponding edge that generated it, in order to avoid replicating the vertices: *e.g.*, considering the face V-V_1-V_2, we can generate the new vertices E_1, E_2 and A, but then, for the face V-V_2-V_3 we must be able to know that E_2 has already been created. For this reason, we will keep a database in which each entry has a unique key, the edge, and a value, the index of the corresponding generated vertex. For your convenience, the database structure is already implemented in the class `EdgeList` (see the doc in Section A).

Finally, we can consider an edge as a pair of indices, one for each of the two vertices. Again, for your convenience, the corresponding data structure is already defined:

```cpp
/**
 * An edge is defined as a pair of vertex indices
 */
using edge = std::pair<idxtype, idxtype>;

// exemples of usage
edge e(2, 3);           // create an edge e with vertex indices 2 and 3
idxtype v1 = e.first;   // get the first index (v1 == 2 in this case)
idxtype v2 = e.second;  // get the first index (v2 == 3 in this case)
```

6.2 Let's implement it

To make things (hopefully...) easier, the algorithm has been split into two main functions. The function `loopSubdivision()` is the main one containing the 3 steps discussed above, and is called to subdivide a mesh. The function `getNewVertex()` is a helper function that creates new vertices.

1. Complete the helper function `getNewVertex()` in `loop.cpp`: this function is used to create a new vertex at a given edge during the subdivision process. It returns the index of the vertex associated with the given edge. See the complete list of its parameters in the source code.

   - To check if an edge is a boundary edge, you can use the function `isBoundaryEdge()`, which returns the indices of the two opposite vertices of the edge, or only one if the edge is a boundary edge (and only in this case it returns `true`):

```cpp
/**
 * It checks if the edge e is a boundary edge in the list of triangle. It also
 * return the indices of the two opposite vertices of the edge or only one of
 * them if it is a boundary edge
 *
 * @param[in] e the edge to check
 * @param[in] mesh the list of triangles
 * @param[out] oppVert1 the index of the first opposite vertices
 * @param[out] oppVert2 the index of the second opposite vertices (if any)
 * @return true if the edge is a boundary edge
```

```
11  */
12  bool isBoundaryEdge(const edge& e, const vector<face>& mesh, idxtype& oppVert1, idxtype& oppVert2 )
```

2. Complete the function `loopSubdivision()` in `loop.cpp`: given a list of vertices and the corresponding mesh as input, it returns a new list of vertices, their normals, and the new mesh corresponding to the output of one step of Loop's subdivision algorithm.

   - The function implements the three main parts of the algorithm described above. They roughly correspond to 3 main couples of `for` loops:
     - the first `for` loop iterates over all the faces to generate the new vertices and the new faces; it uses the database `newVertices`, an instance of `EdgeList` (see Section A), to keep track of the new vertices and the edges that generated them.
     - The second `for` loop updates the original vertices according to the Loop rule (3); again, since we don't have a neighbourhood relation for the vertices, we can update the vertices incrementally going through each face: each vertex of a face is updated using (3) with the other two vertices of the face. To this purpose, we can use an auxiliary vertex list `tmp` that cumulates the contribution from each face. For example, considering the vertex V in Figure 4:
       - the first face V-V_1-V_2 updates V using V_1 and V_2, the second face V-V_2-V_3 using V_2 and V_3, and so on.
       - Be careful though, as this way each V_i is contributing twice, hence a small change in the coefficients of (3) is required...
       - We are not quite finished yet, as we also need to divide the final cumulated sums by the number of faces in which the vertex has been seen (this corresponds to the $\frac{1}{n}$ in (3)). For this, we can keep another list of integer `occurrences` that is updated every time a vertex is updated[1].
     - The third and final `for` loop updates the normals associated to each vertex of the new mesh, in a similar way as you already did for the `load()` function.

Once you are done implementing the algorithm you can switch between the original model and its subdivided version using the key [H]. You can also use the keys [1] ... [4] to apply $n = 1 \div 4$ iteration of the Loop algorithm. It may be wise not to apply more than 2 subdivision iterations to large meshes (it may take a while...).

You can try the program with the different 3D models that you can find in `data` folder. Avoid applying too many subdivision steps for large meshes. You should note something *b*izarre happening when subdividing some of the models, *e.g.* the cow or the teapot. To find out what could be causing this problem you can try to look at the normals of the vertices where the problem occurs (key [N]): try to compare the affected vertices and the normal(s) on the original mesh and on the subdivided model, you should note something strange going on at those vertices... Any clue?

[1] As you can probably note, this is actually a simplification, it will only works with manifold meshes without boundaries. If you want to implement it correctly you have to built a proper data structure to have the neighbourhood relation of vertices...

[git] • Branch: develop @ df05f2c • Release: v2025.1.0-rc5 (2025-04-01)
Head tags: v2025.1.0-rc5

A The EdgeList class

```
/**
 * A helper class containing the indices of the new vertices added with the subdivision
 * coupled with the edge that has generated them. More specifically, it is a list in which
 * each entry has a key (ie an identifier) and a value: the key is the edge that
 * generate the vertex, the value is the index of the new vertex.
 * The key (ie the edge) is unique, ie an edge cannot generate more than one vertex. The
 * edge is a pair of vertex indices, two edges are the same if they contain the
 * same pair of vertices, no matter their order, ie
 * edge(v1, v2) == edge(v2, v1)
 *
 * @see edge
 */
class EdgeList
{
public:

    /**
     * Add the edge and the index of the new vertex generated on it
     * @param[in] e the edge
     * @param[in] idx the index of the new vertex generated on the edge
     */
    void add( const edge& e, const idxtype& idx );

    /**
     * Return true if the edge is in the map
     * @param[in] e the edge to search for
     * @return true if the edge is in the map
     */
    bool contains( const edge& e ) const;

    /**
     * Get the vertex index associated to the edge
     * @param e the edge
     * @return the index
     */
    idxtype getIndex( const edge& e );
...
}
```

B The full key mapping

Key mapping for the viewer:

- W enable/disable wireframe
- A enable/disable `GL_SMOOTH` rendering (*i.e.* shading)
- S enable/disable index rendering
- D enable/disable solid rendering of the mesh (*i.e.* no face color)
- H switch between the original model and its subdivided version

- N show/hide vertex normals
- 1 ... 4 apply 1 ... 4 steps of subdivision
- You can move around the 3D model (once displayed) with the usual arrow keys and use PgUp , PgDn to zoom in/out.

C Range loops in C++

In C++, we often need to iterate over the elements of a container, such as a vector or an array. Traditionally, we would use a loop with an index variable, like this

```cpp
std::vector<int> numbers = {1, 2, 3, 4, 5};
for (size_t i = 0; i < numbers.size(); ++i)
{
    std::cout << numbers[i] << std::endl;
}
```

However, this is rather verbose and not the most readable way to iterate over a container. This approach has some drawbacks as it uses an index variable that is not always necessary and can be error-prone.

In modern C++(*i.e.* starting from C++11), we can use a range-based for loop to iterate over the elements of a container more easily and safely. The basic syntax of a range-based for loop is similar to the one in Java or Python:

```cpp
for (range_declaration : range_expression)
{
    // loop body
}
```

The `range_expression` is the container we want to iterate over. The `range_declaration` is a variable that will hold the value of each element in the container as we iterate over it.

Here's how we can use a range-based for loop to iterate over the elements of a vector (but it is similar for any other type of containers like `array`, `map`, etc):

```cpp
std::vector<int> numbers = {1, 2, 3, 4, 5};
for (int num : numbers)
{
    std::cout << num << std::endl;
}
```

In this example, `num` is a variable that holds a copy of the value of each element in the numbers vector as we iterate over it.

We can also use a reference to the element instead of a copy. This is more efficient, especially for large or complex objects. To do this, we use the `auto&` or `const auto&` syntax:

```cpp
std::vector<int> numbers = {1, 2, 3, 4, 5};
for (auto& num : numbers)
{
    std::cout << num << std::endl;
    // We can modify num here
}
```

In this example, `num` is a reference to each element in the numbers vector. We can modify the element of the vector through this reference.

If we want to ensure that we don't accidentally modify the element, we can use a `const` reference:

[git] • Branch: develop @ df05f2c • Release: v2025.1.0-rc5 (2025-04-01)
Head tags: v2025.1.0-rc5

```cpp
std::vector<int> numbers = {1, 2, 3, 4, 5};
for (const auto& num : numbers)
{
    std::cout << num << std::endl;
    // We cannot modify num here
}
```

In this example, `num` is a const reference to each element in the numbers vector. This means that we cannot modify the element through the const reference.

The `auto` keyword is a placeholder for the type of the variable. It is left to the compiler to infer the type of the variable from the initializer. In the case of a range-based for loop, the compiler will infer the type of the variable from the type of the elements in the container (`int` in this case).

The `&` symbol is used to declare a reference. A reference is an alias for an existing variable. We use a reference to refer to the existing variable directly, without making a copy of it.

D Passing Struct-Based Vectors as Raw Pointers in C++

In C++, when working with a `std::vector<T>`, the elements are stored in a contiguous memory block, similar to common arrays. Likewise, when we declare a `struct` containing multiple fields, those fields are stored contiguously in memory, following the order of declaration. For example, consider the following `Vertex` struct:

```cpp
struct Vertex {
    float x, y, z; // Position coordinates
};
```

In memory, each `Vertex` object consists of three contiguous float values: x, y, and z. Now, suppose we have a vector of such structs:

```cpp
std::vector<Vertex> vertices;
```

How can we pass this vector to an API that requires a raw pointer to float (*i.e.* `float*`)? If we simply pass `&vertices`, it won't work because that only gives the address of the vector object itself, not the underlying array of floats. Instead, we need to obtain a reference to the raw float data. To do so, we can use the C-style cast and pass

```cpp
(float*)&vertices[0]
```

This means:

- `vertices[0]` accesses the first `Vertex` element.
- `&vertices[0]` takes its address.
- Casting it to `float*` tells the compiler to interpret this memory as an array of float values.

Since `Vertex` is laid out contiguously in memory, this cast allows us to treat the vector as an array of `3 * numElements` floats. This works only because the struct's fields are stored contiguously, and all of them are float.

[git] • Branch: develop @ df05f2c • Release: v2025.1.0-rc5 (2025-04-01)

Head tags: v2025.1.0-rc5