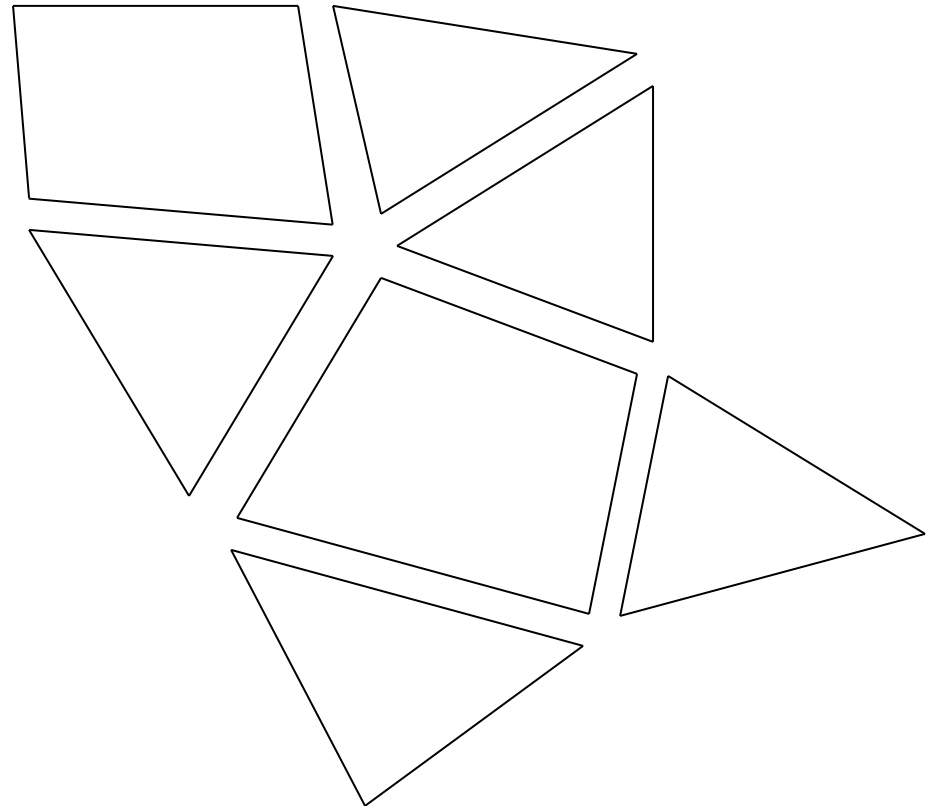


Lecture 13

- Polygonal Mesh Data Structure
- Connectivity Graph (Dual)
- Depth-first and breadth-first search
 - In binary tree
 - In polygonal mesh
- Highlighting High Dihedral-Angle Edges
- Identifying mesh type
 - Using YsShellExt class in a command-line program.

Polygonal Mesh

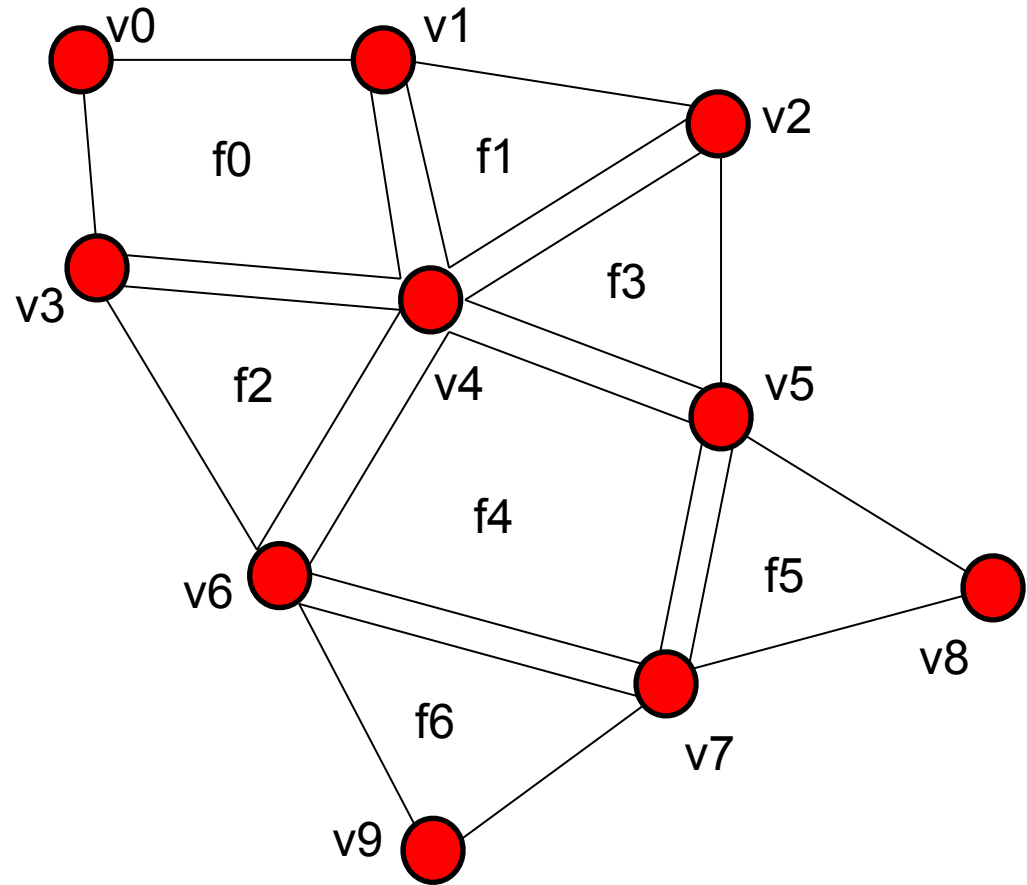
- Minimum information is a set of disconnected polygons (eg. STL data)
- Minimum information is good for visualizing.
- Useless for other purposes.



Better-Than Minimum Information

- Vertices & Connections
- A list of vertices
- A list of polygons, each of which has a chain of vertices

f0 {v1, v0, v3, v4}
f1 {v2, v1, v4}
f2 {v4, v3, v6}
f3 {v2, v4, v5}
f4 {v5, v4, v6, v7}
f5 {v7, v8, v5}
f6 {v9, v7, v6}



More useful information

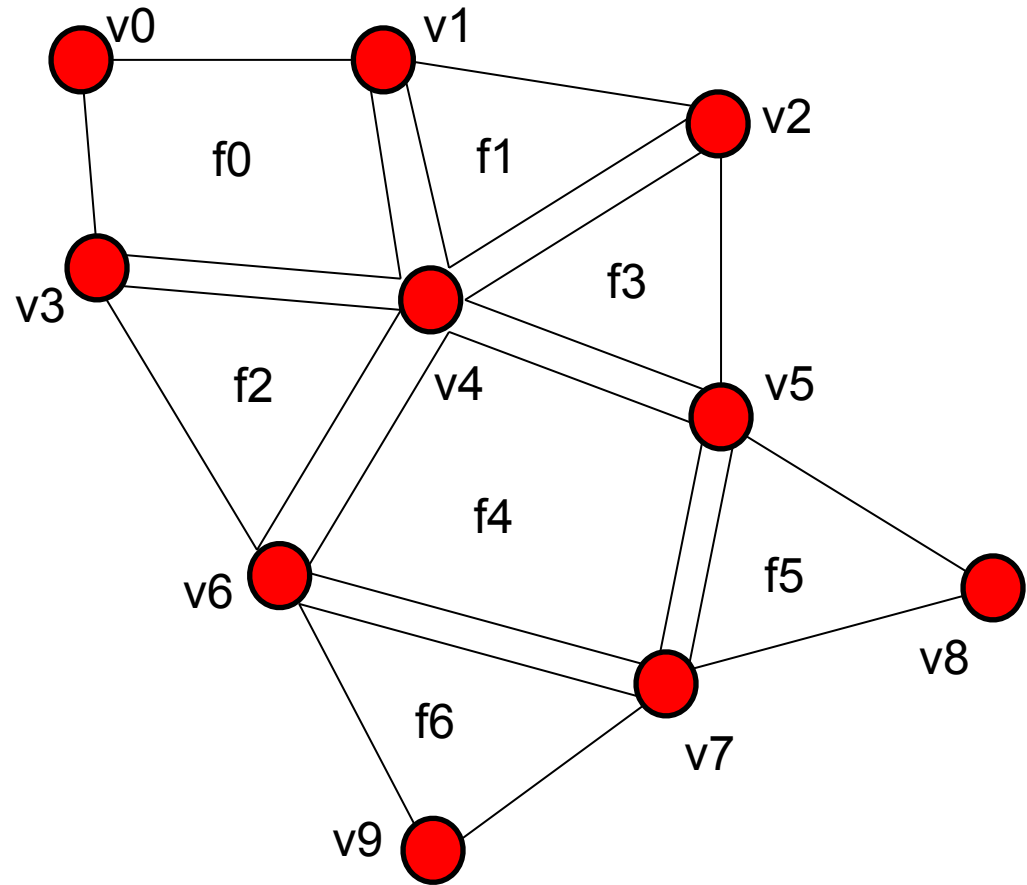
- Polygons are seach-able from a vertex or an edge.
- Can be done by keeping a hash table.

v0 {f0}
v1 {f0, f1}
v2 {f1, f3}
v3 {f0, f2}
v4 {f0, f1, f2, f3, f4}
v5 {f3, f4, f5}

:

e(v0,v1) {f0}
e(v1,v2) {f1}
e(v1,v4) {f0, f1}
e(v3,v4) {f0, f2}
e(v5,v7) {f4, f5}

:



Richer information

- Constraint edges
 - Also called feature edges
 - Chain of vertices
 - Automatic detection is still a hot research topic
- Face groups
 - Also called segments
 - Group of polygons
 - In many cases dual of constraint edges (constraint edges are the boundaries between face groups)
 - Automatic segmentation that works for general geometry does not exist yet.

Closed (manifold), open, and non-manifold polygonal meshes

A polygonal mesh can be classified as:

- Closed mesh (Manifold mesh)
 - Every edge is used by two polygons. (Two-manifold-ness)
 - Necessary condition to represent a solid.
- Open mesh
 - Some edges are used by only one polygon, all other edges are used by two polygons.
 - Good for fabric, sheet-metal, terrain-elevation models.
- Non-manifold mesh
 - Some edges may be used by more than two polygons.
 - Used for an assembly of sheet-metal parts, assembly of multiple volumes.

Implementation

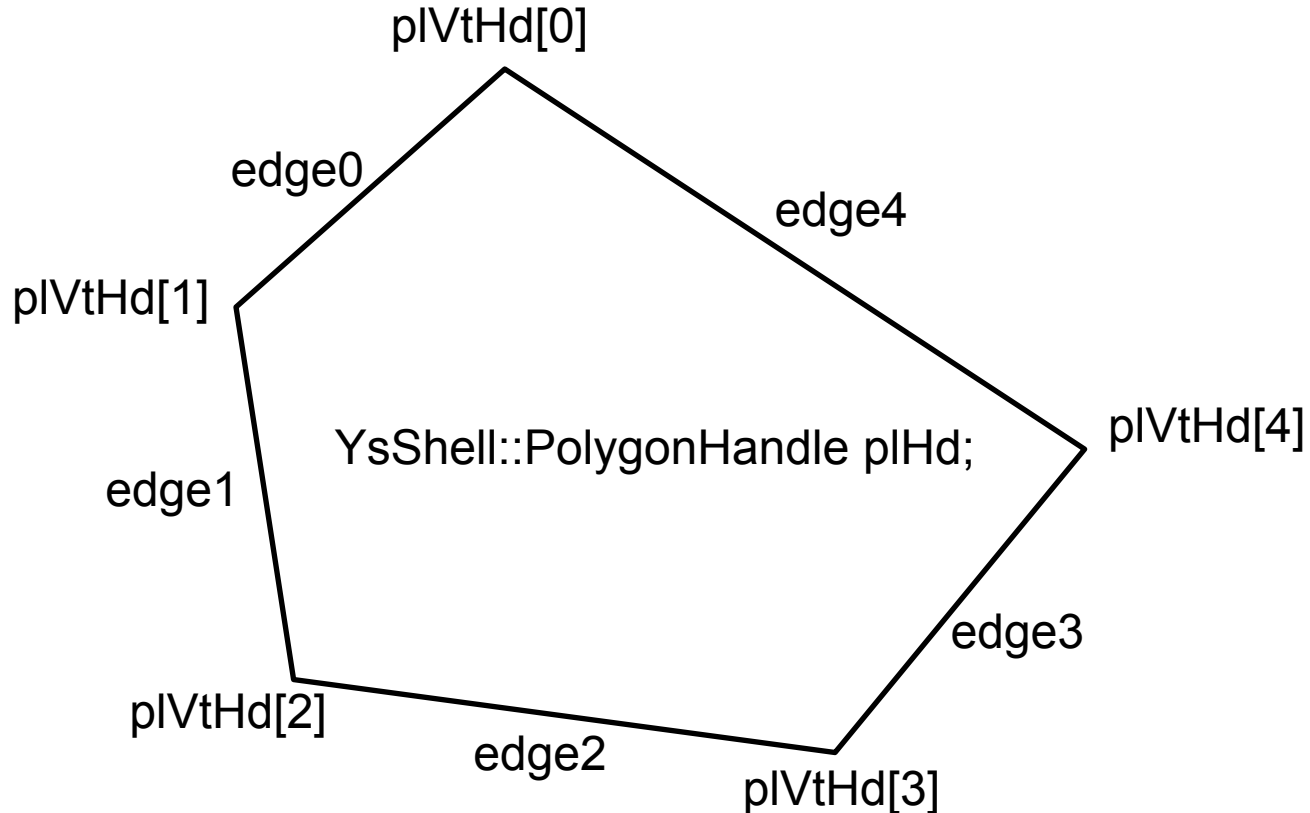
- When a modification is made to the data structure, topology (connection) tables must also be updated.
- If the outside code can directly modify the raw data structure, the table may become out of sync.
- Raw data structure must be hidden from the outside, or only allow access as constant.

Polygonal mesh classes in the public repository

- YsShell
 - Vertices and Polygons.
 - Topology table for vertices and polygons can be attached.
- YsShellExt
 - Inherited from YsShell.
 - Can store constraint edges and face groups, with internal topology table.
 - Volume information in the work.
 - Need to call **EnableSearch()** to use topological information.
- YsShellExtEdit
 - Inherited from YsShellExt
 - Capable of storing linear undo/redo logs.

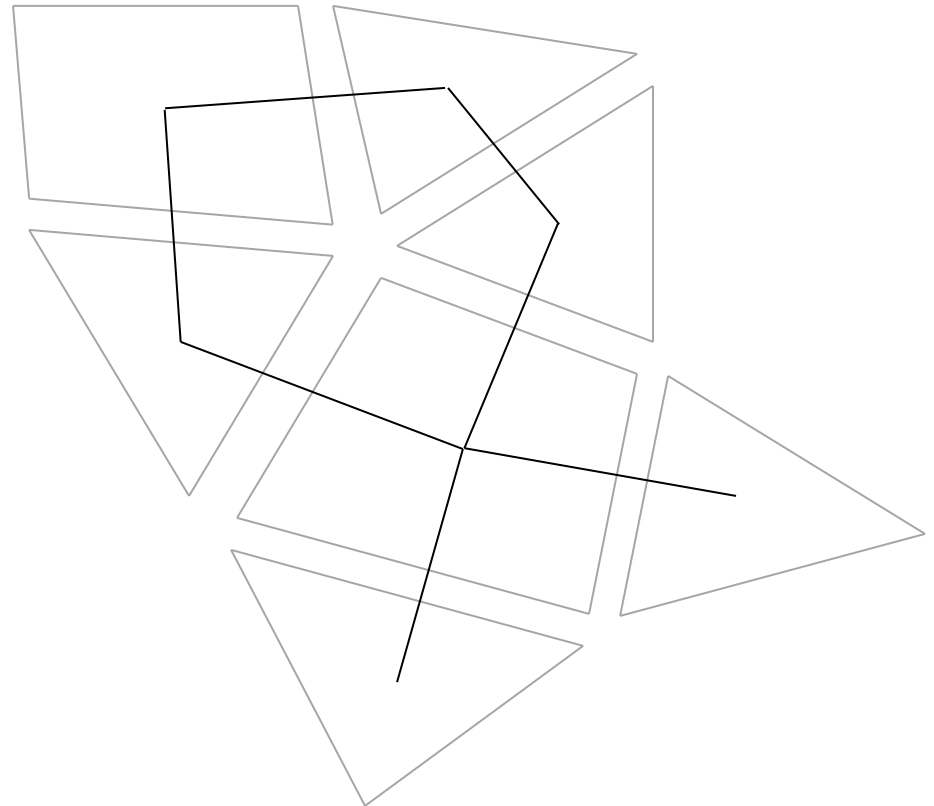
Polygon in YsShell

```
auto pIVtHd=shl.GetPolygonVertex(pIHd);  
(pIVtHd is YsArray <YsShell::VertexHandle>)
```



Drawing a connectivity graph (Dual)

- Connectivity graph of dual: A graph constructed by connecting the centers of the neighboring elements.
- If the triangulation is a Delaunay triangulation, this is called a Voronoi diagram.



- Starting from stl_to_ysshell example.
- D-key : Show dual
- N-key : Back to the normal view

1. Copy stl_to_ysell example and rename the project name to mesh_dual
2. Add the following member variables:

```
enum STATE
{
    STATE_NORMAL,
    STATE_DUAL
};
```

```
STATE state;
```

```
std::vector<float> dualVtx;
```

3. In the constructor, initialize state as:

4. In LoadBinaryStl, add:

```
state=STATE_NORMAL;
```

5. Add MakeDual function.

```
shl.EnableSearch();
```

For each polygon in shl

Get center of the polygon

Get number of edges=number of vertices

For each edge

Find neighbor

If the neighbor exists, and the search key of the neighbor is greater than this polygon. We do not need duplicate line. The search key comparison is for avoiding duplicate.

Get center of the neighboring polygon

YsVec3::xf(), yf(), and zf() returns values in float.

```
void FsLazyWindowApplication::MakeDual(void)
{
    dualVtx.clear();
    for(auto plHd : shl.AllPolygon())
    {
        auto cen=shl.GetCenter(plHd);
        auto nEdge=shl.GetPolygonNumVertex(plHd);
        for(decType(nEdge) edIdx=0; edIdx<nEdge; ++edIdx)
        {
            auto neiPIHd=shl.GetNeighborPolygon(plHd,edIdx);
            if(nullptr!=neiPIHd && shl.GetSearchKey(plHd)<shl.GetSearchKey(neiPIHd))
            {
                auto neiCen=shl.GetCenter(neiPIHd);
                dualVtx.push_back(cen.xf());
                dualVtx.push_back(cen.yf());
                dualVtx.push_back(cen.zf());
                dualVtx.push_back(neiCen.xf());
                dualVtx.push_back(neiCen.yf());
                dualVtx.push_back(neiCen.zf());
            }
        }
    }
}
```

6. In Interval function:

```
if(FSKEY_D==key)
{
    MakeDual();
    state=STATE_DUAL;
}
if(FSKEY_N==key)
{
    state=STATE_NORMAL;
}
```

7. In Draw function:

```
if(STATE_NORMAL==state)
{
    GLfloat lightDir[]={0,0,1};

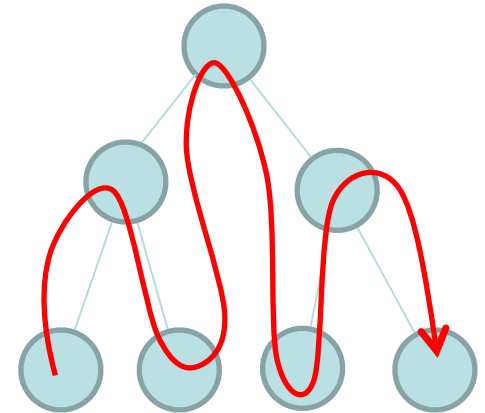
    YsGLSLShaded3DRenderer renderer;
    renderer.SetProjection(projMat);
    renderer.SetModelView(viewMat);
    renderer.SetLightDirectionInCameraCoordinate(0,lightDir);

    renderer.DrawVtxNomCol(GL_TRIANGLES,vtx.size()/3,vtx.data(),nom.data(),col.data());
}
else if(STATE_DUAL==state)
{
    YsGLSLPlain3DRenderer renderer;
    renderer.SetProjection(projMat);
    renderer.SetModelView(viewMat);

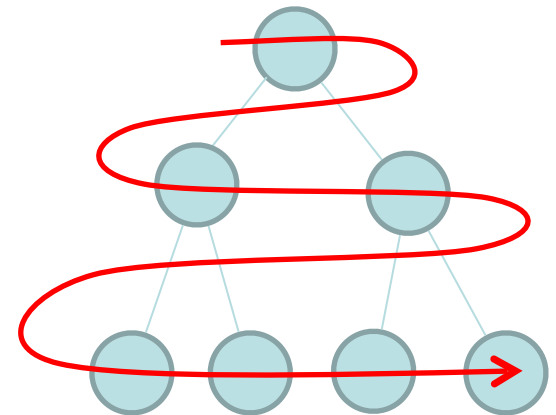
    GLfloat color[4]={0,0,0,1};
    renderer.SetUniformColor(color);
    renderer.DrawVtx(GL_LINES,dualVtx.size()/3,dualVtx.data());
}
```

Depth-first and breadth-first search in binary tree

- Depth-first search
 - Normal traversal of a binary tree.



- Breadth-first search
 - Visit higher-nodes first.



Implementation

- Depth-First search: LIFO (Last-In First-Out) buffer
 - Stack is a LIFO buffer.
 - That's why depth-first search can easily be implemented with a recursion.
- Breadth-First search: FIFO (First-In First-Out) buffer
 - Queue. Can be implemented with a linked-list.
 - Can also be done by a doubling-array.

- Starting from binary_tree_visualizer

```
void BreadthFirstTraversal(const BinaryTree <int,int>::Node *rootNode)
{
    std::vector <const BinaryTree <int,int>::Node *> todo;
    if(nullptr!=rootNode)
    {
        todo.push_back(rootNode);

        long long int firstPtr=0;
        while(firstPtr<todo.size())
        {
            auto node=todo[firstPtr];
            printf("%d ",node->key);


            if(nullptr!=node->left)
            {
                todo.push_back(node->left);
            }
            if(nullptr!=node->right)
            {
                todo.push_back(node->right);
            }

            ++firstPtr;
        }
    }
}
```

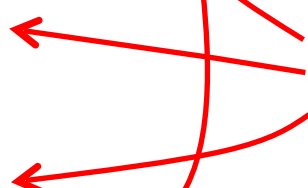
Use std::vector as a queue



Next node to be processed is pointed by firstPtr, which starts from the first in the queue, eventually catches up the length of the queue.

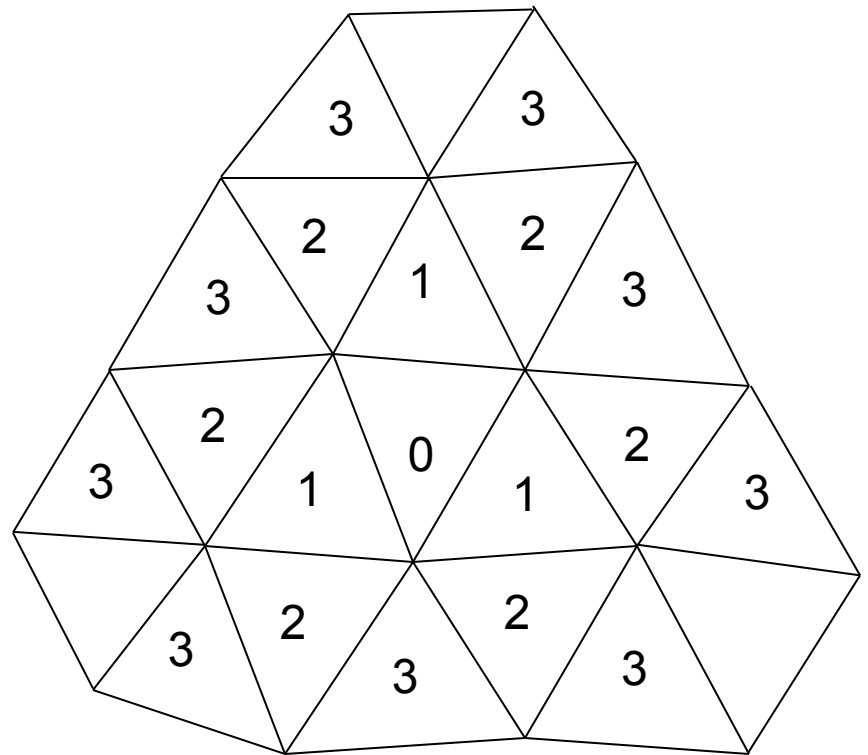


What needs to be processed is added to the back.



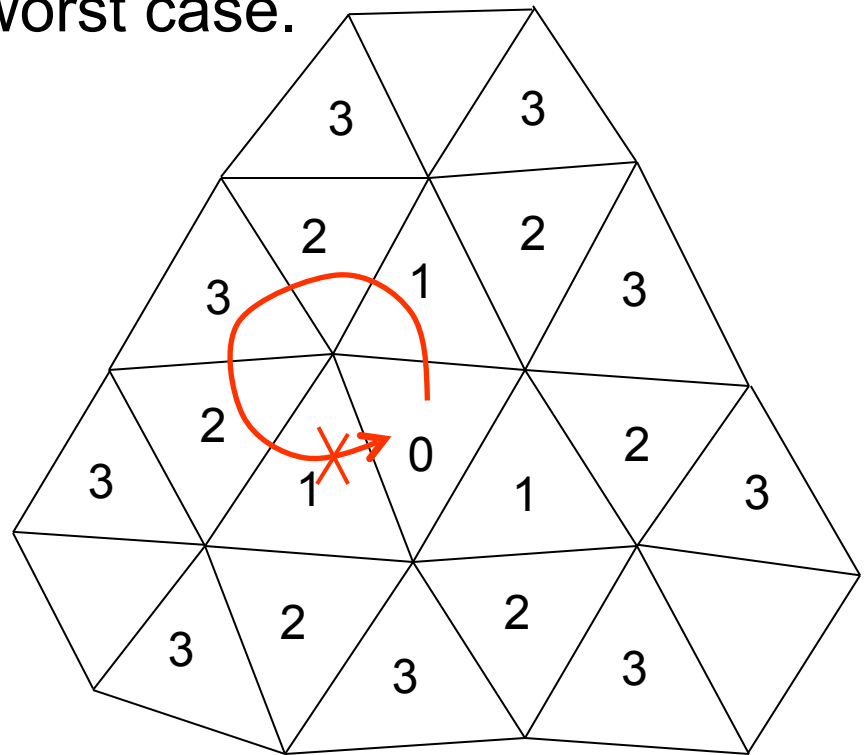
Breadth-First Search in a Graph

- Finding N-neighbors



Breadth-First Search in a Graph

- Need to prevent back fire.
- Unlike a tree, there is a path to come back to a node that has already visited.
- Without keeping track of the already-visited nodes, it falls into an infinite loop in the worst case.



Breadth-First Search in a Graph

- Use a hash table for preventing infinite recursion.
- Make a hash table that connects a node (in this case, a polygon) and the depth.
- When exploring a new node, check if the node is already in the hash table.
 - If it is included, the node has already been visited and does not have to re-visit.
 - If it is not, the node should be visited next.

```

void FsLazyWindowApplication::GetNNNeighbor(
    std::vector<YsShell::PolygonHandle> &neiPIHd,
    std::vector<int> &neiDist,
    const YsShellExt &shl, YsShell::PolygonHandle plHd, int N)
{
    neiPIHd.clear();
    neiDist.clear();
    YsShellPolygonStore visited(shl.Conv());

    neiPIHd.push_back(plHd);
    neiDist.push_back(0);
    visited.Add(plHd);

    for(long long int ptr=0; ptr<neiPIHd.size(); ++ptr)
    {
        if(neiDist[ptr]<N)
        {
            YsShell::PolygonHandle plHd=neiPIHd[ptr];
            auto nEdge=shl.GetPolygonNumVertex(plHd);

            for(int edIdx=0; edIdx<nEdge; ++edIdx)
            {
                auto nei=shl.GetNeighborPolygon(plHd,edIdx);
                if(!nei && YSTRUE!=visited.IsIncluded(nei))
                {
                    neiPIHd.push_back(nei);
                    neiDist.push_back(neiDist[ptr]+1);
                    visited.Add(nei);
                }
            }
        }
    }
}

```

YsShellPolygonStore is a set of polygons.
Using a hash-set as a background data structure.

shl.Conv() returns a conversion object, which allows a conversion from YsShellExt to const YsShell &.

When a polygon is added to an array neiPIHd, topological distance also needs to be added in neiDist, so that neiDist[i] is the distance of neiPIHd[i]

Do not look into a polygon that has already been visited.

A polygon is added to the set called *visited* as soon as it is visited.

Why *YSTRUE*, not *true*? It is a relic from the days that bool-type was not even a built-in type.

What's a conversion object?

- YsShellExt inherits YsShell as protected.
- However, I want to open public access to all public members of YsShell.
- An option is casting as:
 `(const YsShell &)shl;`
which breaks the C++'s protection mechanism.
- A conversion object allows to pass a YsShellExt object to a const YsShell reference without breaking the protection mechanism.

```

class YsShellExt::Converter
{
public:
    YsShellExt *shl;
    operator const YsShell &() const
    {
        return *(const YsShell *)shl;
    }
    operator YsShellExt &() const
    {
        return *shl;
    }
};
class YsShellExt:: ConstConverter
{
public:
    const YsShellExt *shl;
    operator const YsShell &() const
    {
        return *(const YsShell *)shl;
    }
    operator const YsShellExt &() const
    {
        return *(const YsShellExt *)shl;
    }
};

```

```

inline YsShellExt::Converter YsShellExt::Conv(void)
{
    Converter conv;
    conv.shl=this;
    return conv;
}

inline YsShellExt::ConstConverter YsShellExt::Conv(void) const
{
    ConstConverter conv;
    conv.shl=this;
    return conv;
}

```

This way, you can pass shl.Conv() safely as a const YsShell &.

If you by mistake pass shl.Conv() as a YsShell &, the compiler will catch your error.

Safer than force casting to (const YsShell &).

Selecting edge-connected vertices within certain radius from a picked vertex.

- Can be depth-first or breadth-first search.
- Needs same mechanism to stop back fire.

```

std::vector<YsShell::VertexHandle> FsLazyWindowApplication::GetVertexWithinRadius(
    YsShellExt &shl,YsShell::VertexHandle from,const double radius)
{
    std::vector<YsShell::VertexHandle> vtHd;
    const YsVec3 fromPos=shl.GetVertexPosition(from);
    YsShellVertexStore visited(shl.Conv());

    vtHd.push_back(from);
    visited.Add(from);
    for(long long int ptr=0; ptr<vtHd.size(); ++ptr)
    {
        for(auto connVtHd : shl.GetConnectedVertex(vtHd[ptr]))
        {
            if(YSTRUE!=visited.IsIncluded(connVtHd))
            {
                auto distSq=(shl.GetVertexPosition(connVtHd)-fromPos).GetSquareLength();
                if(distSq<radius*radius)
                {
                    vtHd.push_back(connVtHd);
                    visited.Add(connVtHd);
                }
            }
        }
    }

    return vtHd;
}

```

- Highlighting vertices within 2.0 radius of the clicked vertex.

Highlighting High Dihedral-Angle Edges

Dihedral angle

- An angle between two neighboring polygons.
- Give a clue of feature edges.
- Identifying high dihedral-angle edges is important for feature identification.

1. Add:

```
std::vector <float> highDhaEdge;
```

2. Add new state

```
enum STATE  
{  
    STATE_NORMAL,  
    STATE_DUAL,  
    STATE_HIGHDHA  
};
```

3. Add:

```
void FsLazyWindowApplication::MakeHighDha(const double dhaThr)
{
    highDhaEdge.clear();
    for(auto pIHd : shl.AllPolygon())
    {
        auto nEdge=shl.GetPolygonNumVertex(pIHd);
        auto pIVtHd=shl.GetPolygonVertex(pIHd);
        auto nom0=shl.GetNormal(pIHd);
        for(decType(nEdge) edIdx=0; edIdx<nEdge; ++edIdx)
        {
            auto neiPIHd=shl.GetNeighborPolygon(pIHd,edIdx);
            if(nullptr!=neiPIHd && shl.GetSearchKey(pIHd)<shl.GetSearchKey(neiPIHd))
            {
                auto nom1=shl.GetNormal(neiPIHd);
                if(nom0*nom1<cos(dhaThr))
                {
                    YsVec3 edVtPos[2]=
                    {
                        shl.GetVertexPosition(pIVtHd[edIdx]),
                        shl.GetVertexPosition(pIVtHd.GetCyclic(edIdx+1))
                    };
                    highDhaEdge.push_back(edVtPos[0].xf());
                    highDhaEdge.push_back(edVtPos[0].yf());
                    highDhaEdge.push_back(edVtPos[0].zf());
                    highDhaEdge.push_back(edVtPos[1].xf());
                    highDhaEdge.push_back(edVtPos[1].yf());
                    highDhaEdge.push_back(edVtPos[1].zf());
                }
            }
        }
    }
}
```

4. In Interval function,

```
if(FSKEY_H==key)
{
    MakeHighDha(YsPi/4.0); // 45 deg
    state=STATE_HIGHDHA;
}
```

5. In Draw function,

```
else if(STATE_HIGHDHA==state)
{
    YsGLSLPlain3DRenderer renderer; // Again, do not nest the renderer!
    renderer.SetProjection(projMat);
    renderer.SetModelView(viewMat);

    GLfloat color[4]={0,0,0,1};
    renderer.SetUniformColor(color);
    renderer.DrawVtx(GL_LINES,highDhaEdge.size()/3,highDhaEdge.data());
}
```

Identifying a mesh type

- YsShellExt class is independent of OpenGL (only depends on C++11 standard)
- Can be used for building a command-line program.
- Example: Identifying a mesh type.


```

#include <stdio.h>
#include <ysshelltext.h>
int main(int argc,char *argv[])
{
    if(2<=argc)
    {
        YsShellExt shl;
        if(YSOK==shl.LoadStl(argv[1]))
        {
            shl.EnableSearch();

            bool openMesh=false,nonManifold=false;
            for(auto plHd : shl.AllPolygon())
            {
                auto plVtHd=shl.GetPolygonVertex(plHd);
                for(int edIdx=0; edIdx<plVtHd.GetN(); ++edIdx)
                {
                    auto n=shl.GetNumPolygonUsingEdge(plVtHd[edIdx],plVtHd.GetCyclic(edIdx+1));
                    if(n<2)
                    {
                        openMesh=true;
                    }
                    else if(2<n)
                    {
                        nonManifold=true;
                    }
                }
            }

            if(true==openMesh)
            {
                printf("This mesh is an open mesh.\n");
            }
            if(true==nonManifold)
            {
                printf("This mesh is a non-manifold mesh.\n");
            }
        }
    }
    return 0;
}

```