Lecture 14

- House keeping with built-in functions
- Fixing Polygon Orientation
- Color-map
- Greedy-Search Path Finding

House keeping with built-in functions

- Let’s use built-in functions so that the program can support Binary/ASCII STL, SRF, OBJ, and OFF formats.
  - SRF: in-house format for representing a polygonal mesh with face groups, constraint edges, and volumes.
  - OBJ: Wavefront OBJ format. A commonly-used data format for polygonal mesh.
  - OFF: Open Mesh format.

1. Copy from nneighbor example and change TARGET_NAME to mesh_cleaning
2. Add library dependency as:
   ```
   set(LIB_DEPENDENCY fslazywindow ysclass ysport ysgl ysglcpp ysglcpp_gl2 geblkernel)
   ```
3. Delete:
   ```
   #include "binary_stl.h"
   ```
4. Also delete binary_stl.h and binary_stl.cpp from CMakeLists.txt
5. Add:
   ```
   #include <ysport.h>
   #include <ysshellextio.h>
   ```
6. Change LoadBinaryStl function to LoadModel function

```
void FsLazyWindowApplication::LoadModel(const char fn[])
{
    YsString str(fn);
    auto ext=str.GetExtension();

    if(0==ext.STRCMP(".SRF"))
    {
        YsFileIO::File fp(fn,"r");
        if(nullptr!=fp)
        {
            auto inStream=fp.InStream();
            YsShellExtReader reader;
            reader.MergeSrf(shl,inStream);
        }
    }
    else if(0==ext.STRCMP(".OBJ"))
    {
        YsFileIO::File fp(fn,"r");
        if(nullptr!=fp)
        {
            auto inStream=fp.InStream();
            YsShellExtObjReader reader;
            YsShellExtObjReader::ReadOption defaultOption;
            reader.ReadObj(shl,inStream,defaultOption);
        }
    }
    else if(0==ext.STRCMP(".STL"))
    {
        shl.LoadStl(fn);
    }
    else if(0==ext.STRCMP(".OFF"))
    {
        YsFileIO::File fp(fn,"r");
        if(nullptr!=fp)
        {
            auto inStream=fp.InStream();
            YsShellExtOffReader reader;
            reader.ReadOff(shl,inStream);
        }
    }
```

```
    YsShellToVtxNom(vtx,nom,col,shl);
    shl.EnableSearch();
    CacheBoundingBox();
}
```

7. Modify BeforeEverything function.

```
void FsLazyWindowApplication::BeforeEverything(int argc,char *argv[])
{
   if(2<=argc)
   {
      LoadModel(argv[1]);
   }
}
```

Fixing Polygon Orientation

Why do we care about the orientation?

- Hardware culling – GPU can identify the orientation of the polygons in the projected coordinate and drop primitives that are facing away from the camera.
- If the polygons are correctly oriented, the normal vector can be calculated by the right-hand rule.

Broken orientation

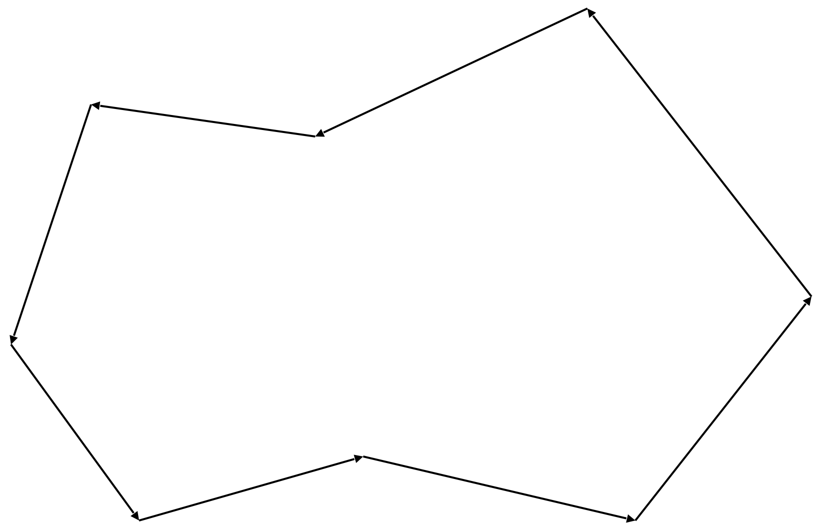
The orientation can be wrong in many ways:

- Assigned normal vectors are correct, but not oriented correctly.
- The orientation is correct, but the assigned normal vector is wrong.
- Both the assigned normal vector and orientation are wrong, but the mesh is closed.

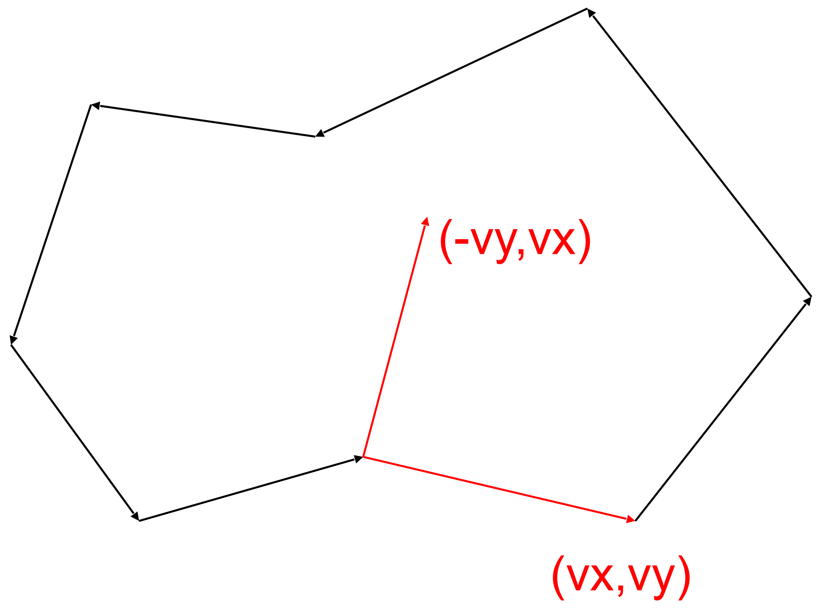
- If the mesh is closed, it is possible to auto-fix the orientations.

In 2D,

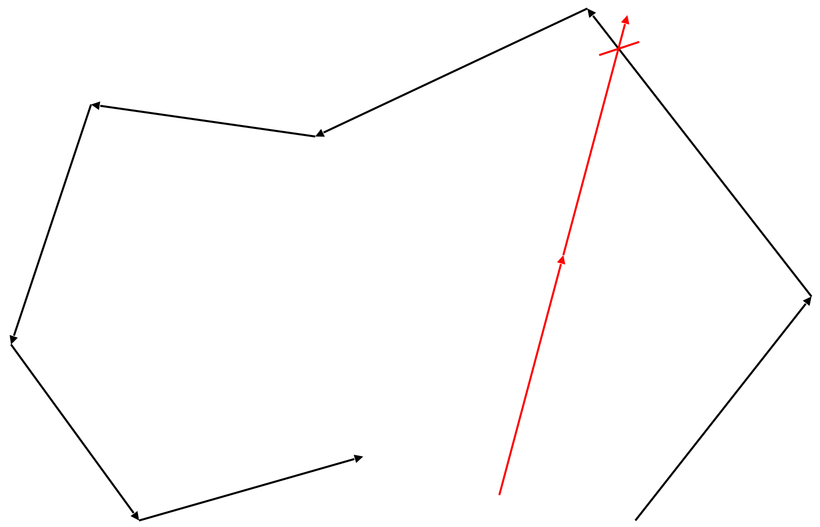
- How can you check if the polygon is oriented clockwise or counter-clockwise?



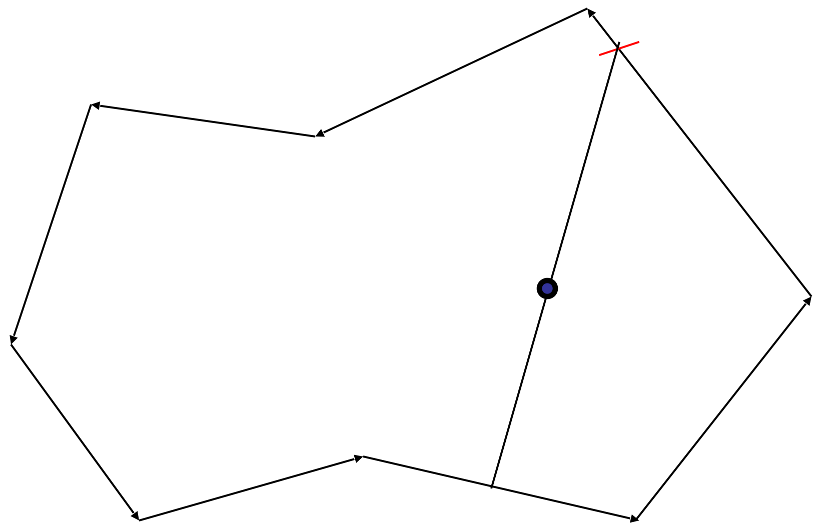
1. Pick one edge, and calculate an edge vector, then rotate it 90 degrees counter-clockwise.



2. Then shoot a ray to the direction of the vector from the center of the edge, and calculate intersections with the edges except the one where the ray is shot from.
3. Pick the one that is (1) in front of and (2) closest to the point where the ray is shot from



4. Calculate the mid-point of the point where the ray is shot from, and the first intersection.
5. If the mid-point is inside the polygon, the polygon is counter-clockwise. If the mid-point is on the boundary, pick another edge and try again.



Possible variations

- In step 3, you can calculate number of intersections and use the parity method. But, if the ray hits a vertex, you need to change the ray direction, or need to do something.

In 3D,

Find one correct orientation with the similar idea.

1. Pick one polygon and calculate a normal vector by the right-hand rule.
2. Shoot a ray from the center of the polygon, to the direction of the normal vector, and calculate intersections with other polygons.
3. If the ray hits nothing, the polygon is correctly oriented.
4. If the ray hits some polygons, pick the mid-point of the center of the polygon and the first intersection.
5. If the mid-point is outside of the volume enclosed by the mesh, the polygon is correctly oriented. If not, the polygon orientation must be flipped. If the mid-point is on the boundary of the volume, pick a different polygon and try again.

After finding one correct orientation,

After finding one correct orientation, propagate the correct orientation to the neighboring polygons.

Finally, calculate correct normal vectors for all polygons.

First half – Finding if a polygon needs to be flipped.

1. Copy project from mesh_cleaning, change target name to mesh_orientation
2. Add a function void FixOrientationFrom(YsShell::PolygonHandle plHd)

Use YSBOOL because it can be YSTRUE, YSFALSE, or YSTFUNKNOWN. The calculation may fail for many reasons.

```
YSBOOL FsLazyWindowApplication::PolygonNeedFlip(YsShell::PolygonHandle plHd) const
{
    YsArray <YsVec3,4> plVtPos;
    shl.GetPolygon(plVtPos,plHd);
    if(3<=plVtPos.GetN())
    {
        auto cen=shl.GetCenter(plHd);
        auto v1=plVtPos[1]-plVtPos[0];
        auto v2=plVtPos[2]-plVtPos[0];
        auto nom=v1^v2;

        if(YSOK==nom.Normalize())
        {
            YsArray <YsVec3> itscPos;
            YsArray <YsShell::PolygonHandle> itscPlHd;

            shl.ShootInfiniteRay(itscPos,itscPlHd,cen,nom);

            YsShell::PolygonHandle nearestItscPlHd=nullptr;
            YsVec3 nearestItscPos;
            double nearestItscDist=0.0;
            for(int i=0; i<itscPlHd.GetN(); ++i)
            {
                if(itscPlHd[i]!=plHd && 0.0<(itscPos[i]-cen)*nom)
                {
                    auto d=(itscPos[i]-cen).GetSquareLength();
                    if(nullptr==nearestItscPlHd || d<nearestItscDist)
                    {
                        nearestItscPlHd=itscPlHd[i];
                        nearestItscPos=itscPos[i];
                        nearestItscDist=d;
                    }
                }
            }
```

Operator ^ is cross product for YsVec3.

Shoot infinite ray function gives intersecting points and polygons of a ray.

Picking the nearest intersection.

```
			bool needFlip=false;
			if(nullptr!=nearestItscPlHd)
			{
				auto mid=(cen+nearestItscPos)/2.0;
				auto side=shl.CheckInsideSolid(mid);
				if(side==YSINSIDE)
				{
					needFlip=true;
				}
				else if(side==YSBOUNDARY)
				{
					// Must try from a different polygon.
					return YSTFUNKNOWN;
				}
			}

			if(true==needFlip)
			{
				printf("Need flip.\n");
				return YSTRUE;
			}
			else
			{
				printf("Do not need flip.\n");
				return YSFALSE;
			}
		}
	}
	return YSTFUNKNOWN;
}
```

If there is no intersection, it means the normal is pointing outside. Therefore, I need to do inside/outside check only if there is an intersection.

If the mid-point is inside of the solid, the normal vector based on the orientation is actually pointing inside. Need flip.

If it is on the boundary, return YSTFUNKNOWN as an error. In this case, the check must start from a different polygon.

Testing the function in LBUTTONDOWN event

```
if(evt==FSMOUSEEVENT_LBUTTONDOWN)
{
    drawEnv.SetWindowSize(wid,hei);
    drawEnv.SetViewportByTwoCorner(0,0,wid,hei);
    drawEnv.TransformScreenCoordTo3DLine(lastClick[0],lastClick[1],mx,my);
    lastClick[1]*=80.0;
    lastClick[1]+=lastClick[0];

    auto plHd=PickedTriangle(mx,my);
    if(nullptr!=plHd)
    {
        PolygonNeedFlip(plHd);
    }
    YsShellToVtxNom(vtx,nom,col,shl);
}
```

If it needs flip, do so and re-assign normal.

```
void FsLazyWindowApplication::Flip(YsShell::PolygonHandle plHd)
{
    auto plVtHd=shl.GetPolygonVertex(plHd);
    plVtHd.Invert();
    shl.SetPolygonVertex(plHd,plVtHd);
}

void FsLazyWindowApplication::RecalculateNormal(YsShell::PolygonHandle plHd)
{
    YsArray <YsVec3,4> plVtPos;
    shl.GetPolygon(plVtPos,plHd);
    if(3<=plVtPos.GetN())
    {
        auto v1=plVtPos[1]-plVtPos[0];
        auto v2=plVtPos[2]-plVtPos[0];
        auto nom=v1^v2;
        if(YSOK==nom.Normalize())
        {
            shl.SetPolygonNormal(plHd,nom);
        }
    }
}
```

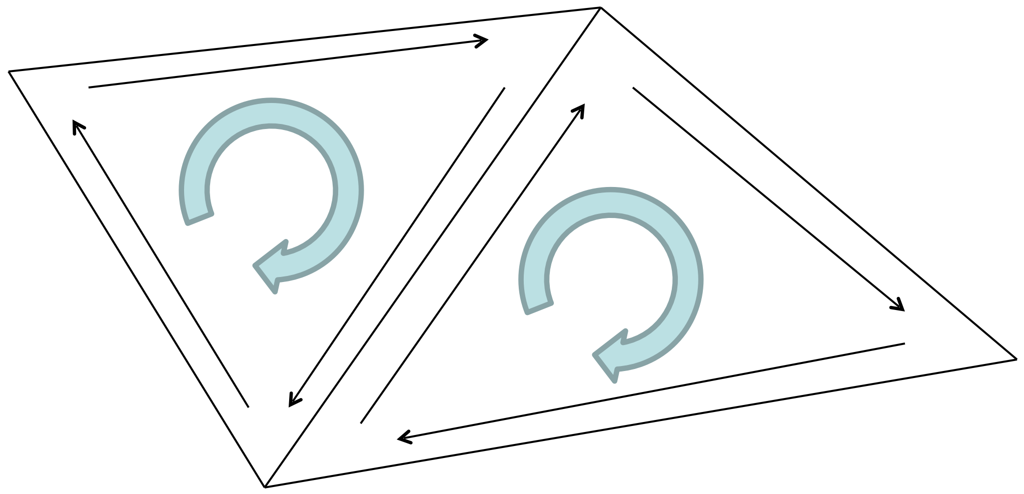
Test in the LBUTTONDOWN event

```
if(evt==FSMOUSEEVENT_LBUTTONDOWN)
{
    drawEnv.SetWindowSize(wid,hei);
    drawEnv.SetViewportByTwoCorner(0,0,wid,hei);
    drawEnv.TransformScreenCoordTo3DLine(lastClick[0],lastClick[1],mx,my);
    lastClick[1]*=80.0;
    lastClick[1]+=lastClick[0];

    auto plHd=PickedTriangle(mx,my);
    if(nullptr!=plHd)
    {
        auto needFlip=PolygonNeedFlip(plHd);
        if(YSTRUE==needFlip)
        {
            Flip(plHd);
            RecalculateNormal(plHd);
        }
        if(YSTRUE==needFlip || YSFALSE==needFlip)
        {
            // Place holder for propagation
        }
    }
    YsShellToVtxNom(vtx,nom,col,shl);
}
```

Propagating the correct orientation to other triangles

- You can click on all triangles to fix orientations.
- But, the life is too short.
- Use the characteristic that each edge needs to be used twice by a polygon in different order.



Basic traversal. Add orientation fixing to this.

```
void FsLazyWindowApplication::PropagateOrientation(YsShell::PolygonHandle fromPlHd)
{
    std::vector <YsShell::PolygonHandle> todo;
    YsShellPolygonStore visited(shl.Conv());

    todo.push_back(fromPlHd);
    visited.Add(fromPlHd);
    while(0<todo.size())
    {
        auto plHd=todo.back();
        todo.pop_back();
        auto nEdge=shl.GetPolygonNumVertex(plHd);
        for(decltype(nEdge) i=0; i<nEdge; ++i)
        {
            auto neiPlHd=shl.GetNeighborPolygon(plHd,i);
            if(nullptr!=neiPlHd && YSTRUE!=visited.IsIncluded(neiPlHd))
            {
                todo.push_back(neiPlHd);
                visited.Add(neiPlHd);
            }
        }
    }
}
```

```
void FsLazyWindowApplication::PropagateOrientation(YsShell::PolygonHandle fromPlHd)
{
    std::vector <YsShell::PolygonHandle> todo;
    YsShellPolygonStore visited(shl.Conv());

    todo.push_back(fromPlHd);
    visited.Add(fromPlHd);
    while(0<todo.size())
    {
        auto plHd=todo.back();
        auto plVtHd=shl.GetPolygonVertex(plHd);
        todo.pop_back();
        auto nEdge=shl.GetPolygonNumVertex(plHd);
        for(decltype(nEdge) i=0; i<nEdge; ++i)
        {
            YsShell::VertexHandle edVtHd[2]=
            {
                plVtHd[i],plVtHd.GetCyclic(i+1)
            };
            auto neiPlHd=shl.GetNeighborPolygon(plHd,i);
            if(nullptr!=neiPlHd && YSTRUE!=visited.IsIncluded(neiPlHd))
            {
                auto neiPlVtHd=shl.GetPolygonVertex(neiPlHd);
                for(int j=0; j<neiPlVtHd.GetN(); ++j)
                {
                    if(edVtHd[0]==neiPlVtHd[j] && edVtHd[1]==neiPlVtHd.GetCyclic(j+1))
                    {
                        Flip(neiPlHd);
                        break;
                    }
                }

                RecalculateNormal(neiPlHd);
                todo.push_back(neiPlHd);
                visited.Add(neiPlHd);
            }
        }
    }
}
```

Need a chain of vertices of the current polygon to see if the neighboring polygon is using the same edge in a different order.

Two vertices of edge i of the current polygon

If an edge of the neighboring polygon is using the same edge in the same order, the neighbor polygon needs to be flipped.

Test from LBUTTONDOWN

```
if(evt==FSMOUSEEVENT_LBUTTONDOWN)
{
    drawEnv.SetWindowSize(wid,hei);
    drawEnv.SetViewportByTwoCorner(0,0,wid,hei);
    drawEnv.TransformScreenCoordTo3DLine(lastClick[0],lastClick[1],mx,my);
    lastClick[1]*=80.0;
    lastClick[1]+=lastClick[0];

    auto plHd=PickedTriangle(mx,my);
    if(nullptr!=plHd)
    {
        auto needFlip=PolygonNeedFlip(plHd);
        if(YSTRUE==needFlip)
        {
            Flip(plHd);
            RecalculateNormal(plHd);
        }
        if(YSTRUE==needFlip || YSFALSE==needFlip)
        {
            PropagateOrientation(plHd);
        }
    }
    YsShellToVtxNom(vtx,nom,col,shl);
}
```

Question

- How would you correct orientations if one model consists of two bodies?
- I let you think.

Color-map problem

- 4-color theorem: You only need four colors to paint all polygons so that no neighboring polygons have the same color.
- It is true for any genus-0 geometry.
- Obvious for a triangular mesh.
- If you don’t care about the minimum number of triangles, it is not a big deal.
- You can easily implement it if you know neighboring polygons.

```
void FsLazyWindowApplication::ColorMap(void)
{
    std::vector <YsColor> palette;
    palette.push_back(YsBlue());
    palette.push_back(YsRed());
    palette.push_back(YsGreen());
    palette.push_back(YsYellow());
    palette.push_back(YsMagenta());
    palette.push_back(YsCyan());
    palette.push_back(YsDarkBlue());
    palette.push_back(YsDarkRed());
    palette.push_back(YsDarkGreen());
    palette.push_back(YsDarkYellow());
    palette.push_back(YsDarkMagenta());
    palette.push_back(YsDarkCyan());


    for(auto plHd : shl.AllPolygon())
    {
        std::vector <bool> used;
        used.resize(palette.size());

        auto nEdge=shl.GetPolygonNumVertex(plHd);
        for(decltype(nEdge) i=0; i<nEdge; ++i)
        {
            auto neiPlHd=shl.GetNeighborPolygon(plHd,i);
            if(nullptr!=neiPlHd)
            {
                auto neiCol=shl.GetColor(neiPlHd);
                for(int j=0; j<palette.size(); ++j)
                {
                    if(neiCol==palette[j])
                    {
                        used[j]=true;
                        break;
                    }
                }
            }
        }

        int unusedIndex=-1;
        for(int k=0; k<palette.size(); ++k)
        {
            if(true!=used[k])
            {
                unusedIndex=k;
                break;
            }
        }
        if(0<=unusedIndex)
        {
            shl.SetPolygonColor(plHd,palette[unusedIndex]);
        }
        else
        {
            // You come up with a new color,
            // use it, and add to palette.
        }
    }
}
```

Greedy-Search Path Finding

- Starting from a vertex, find a neighboring vertex that shortens the distance to the goal the most.
- Repeat it until reaching the goal or no more next vertex is found.

```
std::vector <YsShell::VertexHandle> FsLazyWindowApplication::FindPath(
    YsShell::VertexHandle fromVtHd,YsShell::VertexHandle toVtHd) const
{
    double dist=shl.GetEdgeLength(fromVtHd,toVtHd);

    std::vector <YsShell::VertexHandle> path;
    path.push_back(fromVtHd);

    auto vtHd=fromVtHd;
    while(vtHd!=toVtHd)
    {
        double bestDist=dist;
        YsShell::VertexHandle nextVtHd=nullptr;

        for(auto vtHdCandidate : shl.GetConnectedVertex(vtHd))
        {
            double candidateDist=shl.GetEdgeLength(vtHdCandidate,toVtHd);
            if(candidateDist<bestDist)
            {
                nextVtHd=vtHdCandidate;
                bestDist=candidateDist;
            }
        }

        if(nullptr==nextVtHd)  // Failed.
        {
            path.clear();
            return path;
        }

        path.push_back(nextVtHd);
        vtHd=nextVtHd;
        dist=bestDist;
    }

    return path;
}
```

- This algorithm may fail.
- There are better algorithms.
- A* (A-star) is more popular. But, you need to be able to implement a priority queue.
- Hopefully I can talk about it later in the semester.