3

V

A 3D Visualization Tool Used for Test Automation in the Forza Series

Gustavo Bastos Nunes

3.1 Introduction

Physics engines usually rely on a collision mesh that is hand-crafted by artists. This meshes may have holes, bad normals, or other wrong data that might cause weird behavior at runtime. Testing those wrong behaviors manually has an extremely high cost in regards to manual testing. One small hole or bad normal can cause a character or vehicle to behave in a completely wrong manner, and those bugs are seldom reproduced because it might depend on many variables such as engine time step, character speed, and angle.

Finding issues like open edges in a mesh is not a complex problem in the polygon mesh processing area, and this feature is available in some 3D content creation packages. However, topology-wise for non-closed meshes, there is no difference from a boundary of a mesh and a hole. Therefore, visualizing what is by design and what is a bug requires filtering and semantic analysis of such a given mesh, which is simply impractical at those tools, particularly for multiscale collision meshes. Thus, this yields a myriad of hard-to-find bugs.

This chapter will introduce a 3D visualization tool that automatically analyzes a mesh for bad holes and normal data and gives the manual tester an easy semantic view of what are likely to be bugs and what is by-design data. It will also go through a quick review of the algorithmic implementation of topics in polygon mesh processing such as mesh traversal, half-edge acceleration data structures, detection of holes, open edges, and other issues. This tool was used during the entire production cycle of *Forza Motorsport 5* and *Forza: Horizon 2* by Turn 10 Studios and Playground Games. At previous releases of the *Forza* series, without this tool, the test team used to spend several hundred hours manual-testing the game to find collision mesh issues and finished without a guarantee that there were none, since it was basically a brute-force approach. With this tool, an entire mesh of a track can now be analyzed and all collision bugs can be found in less than 500 milliseconds. Moreover, this provides us the trust that we are shipping the game with collision meshes in a perfect state.

3.2 Collision Mesh Issues

The tool was originally crafted to detect only holes at the collision mesh; later on it was expanded to also detect flipped/skewed normal and malformed triangles. Those are the main issues that causes problems with the physics engine at runtime.

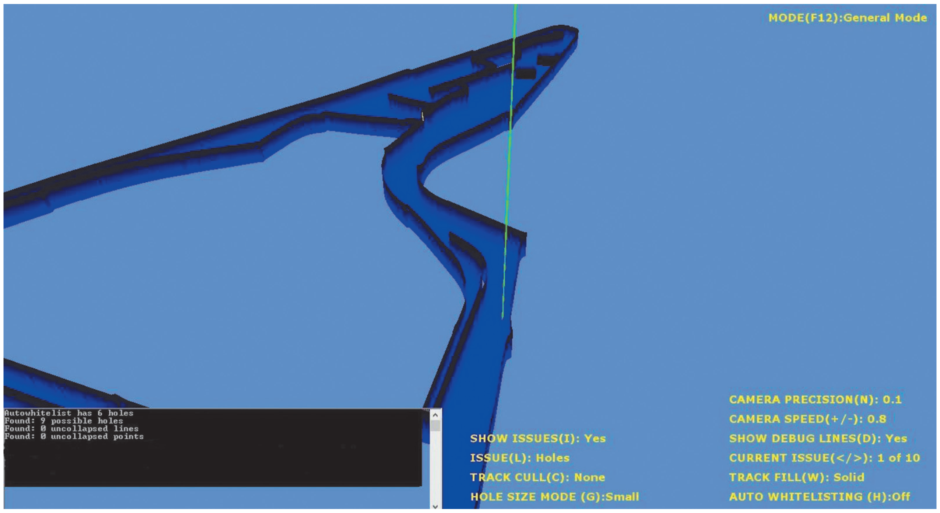
3.2.1 Holes

Holes in the collision mesh was a great problem to us. Big holes were usually not a problem because they end up being caught by the testers and their behaviors are typically very deterministic and clear: e.g., the car goes through a wall that it is not supposed to or it falls through the world. Although it was not fast to detect those issues and sometimes it was costly, they wind up being detected and fixed. The real problems were the small/tiny holes where it would cause the car to behave oddly and in a non-natural way; such a bug would only reproduce with a specific car, in a specific speed, and if hit at a specific angle. Moreover, when the bug was filed, it usually only had a video with the odd behavior happening at that specific part of the track, so the artist that would be responsible to fix it usually would not know what specific triangle was causing that. Figures 3.1 and 3.2 show a tiny hole being detected by the tool.

3.2.2 Wrong Normals

Normals in the collision mesh are responsible for determining the force that is applied to the car at each particular vertex. Thus, if a particular piece of road is flat, the normal at those vertices should be straight up. If the normal was flipped, the user would see the car being dragged into the ground. As with small holes, a single skewed or flipped normal could cause a completely wrong behavior at runtime, and it may also be very hard to detect by only reproducing in very specific scenarios. In this chapter I will call a *flipped normal* any normal where the angle with the Y-up vector is greater than $90°$. A flipped normal is always a wrong normal; however, we can have by-design skewed normals, which is how we simulate the physics effects of the tires hitting the rumble strips. Therefore, it is particular hard to detect when a skewed normal is by design or not.

Artists do not usually author the normals by hand; they are created by the 3D digital content creation tool. The reason why the normals get skewed or flipped



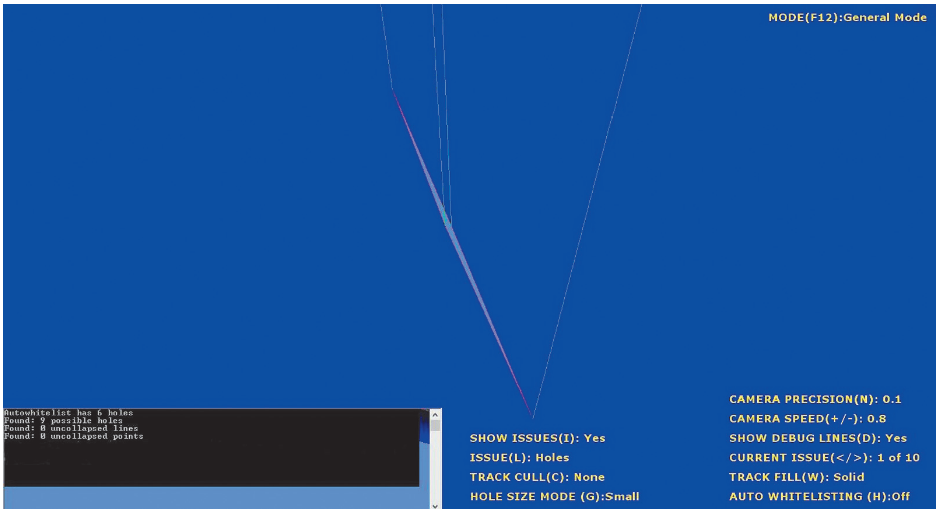
MODE(F12):General Mode

```
Autowhitelist has 6 holes
Found: 9 possible holes
Found: 0 uncollapsed lines
Found: 0 uncollapsed points
```

SHOW ISSUES(I): Yes
ISSUE(L): Holes
TRACK CULL(C): None
HOLE SIZE MODE (G):Small

CAMERA PRECISION(N): 0.1
CAMERA SPEED(+/-): 0.8
SHOW DEBUG LINES(D): Yes
CURRENT ISSUE(</>): 1 of 10
TRACK FILL(W): Solid
AUTO WHITELISTING (H):Off

Figure 3.1. A small hole highlighted by the tool in green.



MODE(F12):General Mode

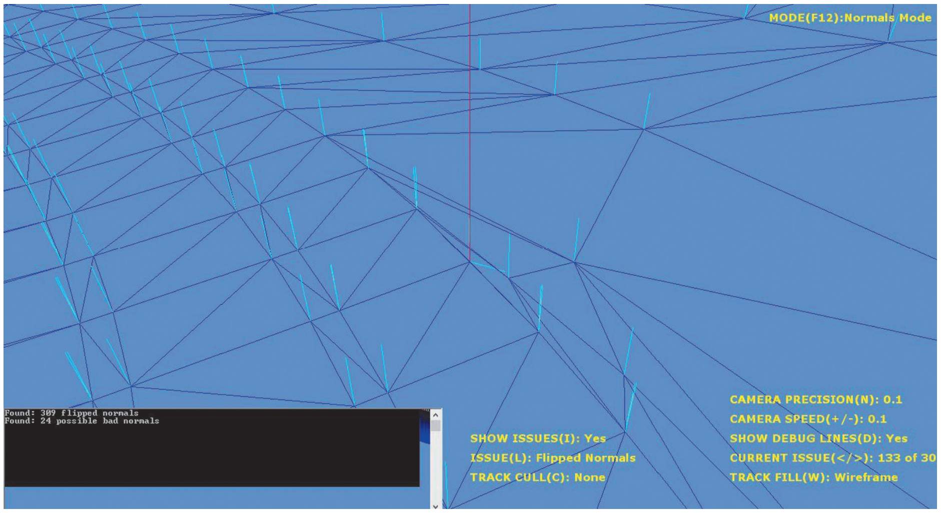
```
Autowhitelist has 6 holes
Found: 9 possible holes
Found: 0 uncollapsed lines
Found: 0 uncollapsed points
```

SHOW ISSUES(I): Yes
ISSUE(L): Holes
TRACK CULL(C): None
HOLE SIZE MODE (G):Small

CAMERA PRECISION(N): 0.1
CAMERA SPEED(+/-): 0.8
SHOW DEBUG LINES(D): Yes
CURRENT ISSUE(</>): 1 of 10
TRACK FILL(W): Solid
AUTO WHITELISTING (H):Off

Figure 3.2. Same hole from Figure 3.1 on a very close-up view.

is because they might weld vertices and create really small triangles. Those small triangles together yield precision issues on the calculation of normals by the 3D DCC tool, and the collision mesh ends up with bad normals. Figure 3.3 shows a flipped normal detected by the tool. Note how every normal is following a good



MODE(F12):Normals Mode

```
Found: 309 flipped normals
Found: 24 possible bad normals
```

SHOW ISSUES(I): Yes
ISSUE(L): Flipped Normals
TRACK CULL(C): None

CAMERA PRECISION(N): 0.1
CAMERA SPEED(+/-): 0.1
SHOW DEBUG LINES(D): Yes
CURRENT ISSUE(</>): 133 of 30
TRACK FILL(W): Wireframe

Figure 3.3. A flipped normal flagged by the tool in red.

pattern of being aligned with the Y-up vector while the flagged normal clearly disturbs this pattern.

3.2.3 Malformed Triangles

Triangles that are malformed, such as triangles that are lines (i.e., two vertices are collinear) or triangles where the three vertices are the same, are also detected by the tool. Usually they are not sources of very bad behaviors like holes or bad normals, but they are definitely wrong data that should not be there to begin with.

3.3 Detecting the Issues

This section will cover the details of building the data structure needed to query and traverse the mesh and how we detect each of the issues described in the previous section.

3.3.1 Building the Data Structure

To be able to detect holes, we need to add the mesh to an easy queryable data structure. We used a half-edge data structure [Mäntylä 88, pp. 161–174; Kettner 99]. Half-edge data structures are easy to implement and are able to represent arbitrary orientable 2-manifold polygonal meshes with no complex edges or vertices.

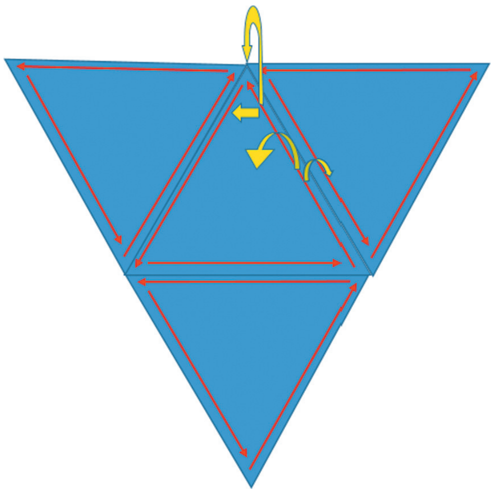


Figure 3.4. Half-edges in red and its references in yellow to the face, vertex, next half-edge, and opposite half-edge.

The data structure is stored in such a way that each triangle face has three half-edges in the same winding order and each of those edges references the next half-edge, the opposite half edge of its neighbor face, and a vertex like it is shown in Figure 3.4. The members of our mesh are detailed in the following pseudocode snippet:

```
class Mesh
{
    List<Face> faces; //List of all faces of this mesh.
}

class HalfEdge
{
    Face face; //Reference to the face this half-edge belongs to.
    HalfEdge next; //Reference to the next half-edge.
    HalfEdge opposite; //Reference to the opposite half-edge.
    Vertex v; //Reference to the tail vertex of this half-edge.
}

class Face
{
    HalfEdge edge; //Reference to one half-edge of this face.
    Vertex v1, v2, v3; //Reference to the three vertices of this face.
}
class Vertex
{
    Vector3 Position; //Position of the vertex.
    Vector3 Normal; //Normal of the vertex.
}
```

By parsing the vertex and index buffer of a mesh and filling into a data structured like the above one, it is really easy to start doing queries on the mesh. For instance, the following snippet finds all neighboring faces of a given face:

```
List<Face> GetNeighbors(Face face)
{
    List<Face> neighbors = new List<Face>();

    if (face.edge.opposite != null)
    {
        neighbors.Add(face.edge.opposite.face);
    }
    if (face.edge.next.opposite != null)
    {
        neighbors.Add(face.edge.next.opposite.face);
    }
    if (face.edge.next.next.opposite != null)
    {
        neighbors.Add(face.edge.next.next.opposite.face);
    }
        return neighbors;
}
```

For more information on half-edge data structures, we suggest the following references to the reader: [McGuire 00, Botsch et al. 10].

3.3.2 Detecting Holes

After storing the mesh in the half-edge data structure, we iterate on the mesh by first looking for holes. To do that, we treat the mesh as if it is an undirected graph where each triangle face is a node and each triangle edge is an edge of the graph. This is illustrated in Figure 3.5.

Next, we conduct a breadth-first search (BFS) looking for any half-edge that does not have an opposite half-edge; this would be an open edge as shown on Figure 3.5. Any open edge is always part of a hole, and we store this open edge on a list of open edges. This process is shown in the following pseudo-code snippet:

```
List<Hole> FindHoles()
{
    //All open edges.
    List<HalfEdgeIdxs> holesEdges = new List<HalfEdgeIdxs>();

    //List of holes to return.
    List<Hole> meshHoleList = new List<Hole>();

    //A set that contains all visited faces.
    Hashset<Face> visitedFaces = new Hashset<Face>();
    //Start by visiting the first face of the mesh.
    Face currFace = meshFacesList[0];
    //A set that contains the non-visited faces.
    HashSet<Face> allFacesHashSet
```

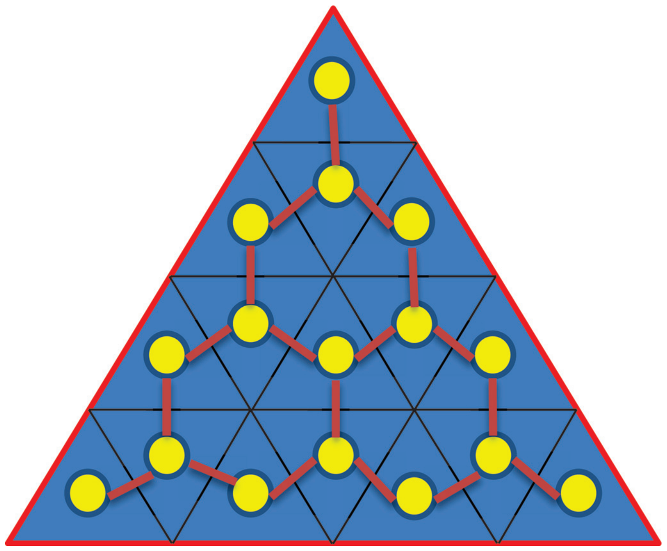


Figure 3.5. Each face is a node, and each triangle edge is an edge of the undirected graph. All the red edges are open edges.

```
            = new HashSet<Face>(meshFacesList);

        //Initialize the BFS queue.
        Queue<Face> bfsQueue = new Queue<Face>();
        bfsQueue.Enqueue(currFace);
        visitedFaces.Add(currFace);
        //Only quit if we have visited all faces
        while (bfsQueue.Count > 0 || visitedFaces.Count
                                     != meshFacesList.Count)
        {
            //If the BFS queue is empty and we are still in the
            //loop, it means that this mesh is a disjoint mesh;
            //we leave this set and go to the next set by
            //re-feeding the queue.
            if (bfsQueue.Count == 0)
            {
                Face face = allFacesHashSet.Next();
                visitedFaces.Add(face);
                bfsQueue.Enqueue(face);
            }
            //Remove from the queue and from the non-visited faces.
            currFace = bfsQueue.Dequeue();
            allFacesHashSet.Remove(currFace);

            //Visit the neighbors of the face.
            List<Face> neighbors = currFace.GetNeighbors();
            foreach (Face neighbor in neighbors)
            {
                 if (!visitedFaces.ContainsKey(neighbor))
                {
                    visitedFaces.Add(neighbor, true);
                    bfsQueue.Enqueue(neighbor);
                }
            }
            //If the number of neighbors of this face is 3,
            //it has no open edges; continue.
            if (neighbors.Count == 3)
            {
                    continue;
            }

            HalfEdge currHalfEdge = currFace.Edge;
            int i = 0;
            //This face has open edges; loop through the edges of
            //the face and add to the open edges list.
            while (i < 3)
            {
                    if (currHalfEdge.Opposite == null)
                    {   //Add the half edge to the hole;
                        //V1 and V2 are the indices of the vertices
                        //on the vertex buffer.
                        HalfEdgeIdxs holeEdge
                             = new HalfEdgeIdxs(currHalfEdge.V1,
                                                currHalfEdge.V2);

                        holesEdges.Add(holeEdge);
                    }

                    currHalfEdge = currHalfEdge.Next;
                    i++;
            }
        }
            //If there are no open edges, return an empty hole list.
            if (holesEdges.Count == 0)
            {
               return meshHoleList;//No holes.
            }
            //Method continues in the next code snippet.
```

The last step is to create a hole based on the open edge information. To do that, we get the first open edge from the list, and let the vertices of this edge be `V1` and `V2`. Next, we assign the open edge to a hole and remove that edge from the open edge list. Also, we mark `V1` as the first vertex of the hole; this is the vertex where the hole has to end when the hole cycle is completed. In other words, there must be an edge on the mesh with vertices `(Vn, V1)` that completes this hole. Now, we loop through all the open edges to find an edge where the first vertex is `V2`; this is the next edge of our hole, and we add it to the hole and remove it from the list of open edges. We continue this search until we find the edge `(Vn, V1)`; this edge completes the hole, and we move on to the next hole. The loop finishes when there are no more edges in the open edges list. The following snippet illustrates this last step:

```
//Get the first open edge and add it to a new hole.
HalfEdgeIdxs currEdge = holesEdges[0];
Hole hole = new Hole();
hole.HoleHalfEdges.Add(currEdge);
//Mark the first vertex of the hole.
```

```
    int firstVertexOfHole = currEdge.V1;
    //Remove the open edge added to the hole from the list of open edges.
    holesEdges.Remove(currEdge);
    while (true)
    {
        //Find the next edge of this hole, where the first vertex is
        //equal to the second one of the current edge.
        HalfEdgeIdxs currEdgeNext = holesEdges.Find(x => x.V1
                                                   == currEdge.V2);
        //Add the found edge to the hole and remove it from the list
        //of open edges.
        hole.HoleHalfEdges.Add(currEdgeNext);
        holesEdges.Remove(currEdgeNext);

        //Test if we found the edge that ends the hole cycle.
        if (currEdgeNext.V2 == firstVertexOfHole)
        {
            meshHoleList.Add(hole);
            //No more open edges; finish loop; all holes found.
            if (holesEdges.Count == 0) break;

            //If there are still open edges, get the next one from
            //the list and start a new hole.
            currEdge = holesEdges[0];
            holesEdges.Remove(currEdge);
            firstVertexOfHole = currEdge.V1;
            hole = new Hole();
            hole.HoleHalfEdges.Add(currEdge.GetName(), currEdge);
        }
        else
        {
            //If we did not find the end of the hole, just go to
            //the next edge.
            currEdge = currEdgeNext;
        }
      }
      //Return the mesh list with all holes.
      return meshHoleList;
    }
```

This algorithm identifies mesh boundaries as a hole, which is explained in the next subsection.

3.3.3 Hole Classification

Topology-wise, for an open mesh, there is no difference between a hole and a boundary of a mesh. This can be easily visualized by making a sector of a circle with a piece of paper and building a cone with this circle, leaving the bottom of the cone open. The bottom of the cone is a hole in the mesh, but it is also the boundary of it. By flattening the cone and making it a circle again, you can again visualize that there is no topology difference between a boundary and a hole; see Figure 3.6.

In our collision meshes there are a great number of boundaries that are by design, and flagging them all as possible issues to be filtered out by the tester would generate too much noise and false positives, making the usage of the tool

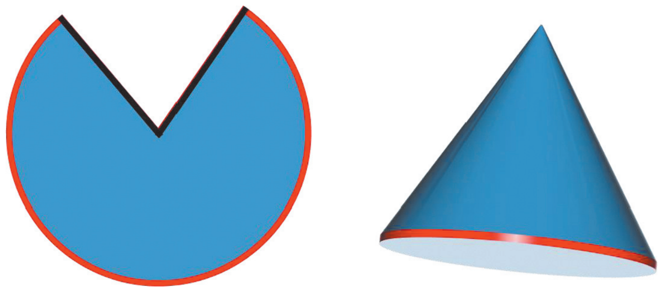


Figure 3.6. The boundary of our flattened cone is highlighted in red (left). The cone is assembled with its hole/boundary highlighted in red (right).

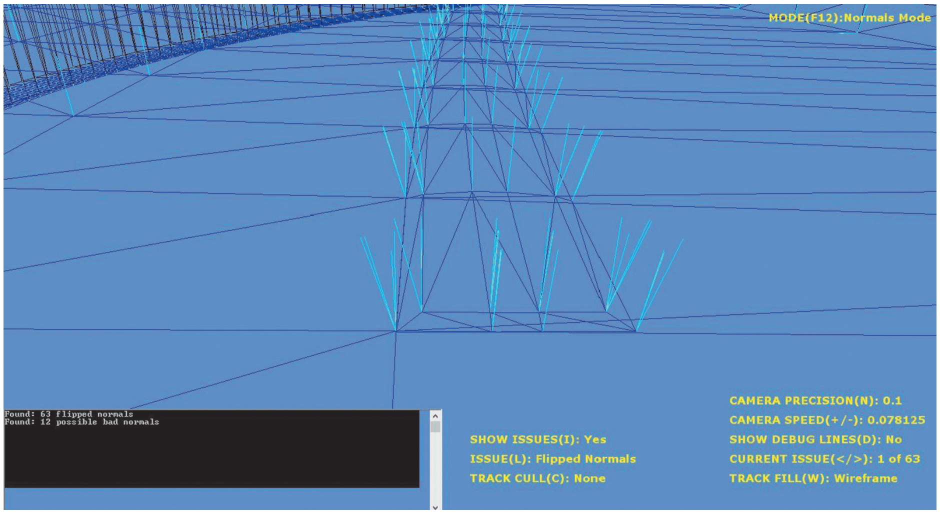
unpractical. To address this issue, we came up with two rules to whitelist holes that are highly likely to be by design. First, our collision meshes have very high vertical walls to prevent cars from falling out of the world, and the vast majority of the by-design boundaries are at the tops of those walls. We usually do not care about issues on the collision mesh that are very high; thus, we whitelist any hole that is found above the upper half of the mesh. The second rule that we use to whitelist is when holes are very big. Our collision mesh contains barriers along the track that have big boundaries, which are all by design; the intent of the second rule is to whitelist those barriers. Whitelisting based on a large hole size has proven to be safe; of course, we could have a giant hole in the track that is indeed a bug, but those are easily and quickly found by playing the build normally. Moreover, the user can also remove the whitelisting and let all holes appear and give a quick inspection by flying above the track.

3.3.4 Detecting Bad Normals

As mentioned in Section 3.2.2, there are two kinds of bad normals: flipped normals and skewed normals. Flipped normals (see Figure 3.3) are straightforward to detect. We loop through all the normals of the collision mesh and mark as flipped any normal that satisfies the following equation:

$$\mathbf{n} \cdot \hat{\mathbf{y}} < \mathbf{0},$$

where $\hat{\mathbf{y}}$ is the unit Y-up vector. Skewed normals are more complicated because we can have those kind of normals by design; see Figure 3.7. However, the ones that are actual bugs come from defects in the crafted mesh, usually very small triangles. The first approach we have tried to identify those is to simply flag triangles with small areas. This did not work well because a normal is influenced by the entire one-ring neighborhood of a vertex and looking locally at only one triangle produced too many incorrect results.



```
Found: 63 flipped normals
Found: 12 possible bad normals
```

MODE(F12):Normals Mode

SHOW ISSUES(I): Yes
ISSUE(L): Flipped Normals
TRACK CULL(C): None

CAMERA PRECISION(N): 0.1
CAMERA SPEED(+/-): 0.078125
SHOW DEBUG LINES(D): No
CURRENT ISSUE(</>): 1 of 63
TRACK FILL(W): Wireframe

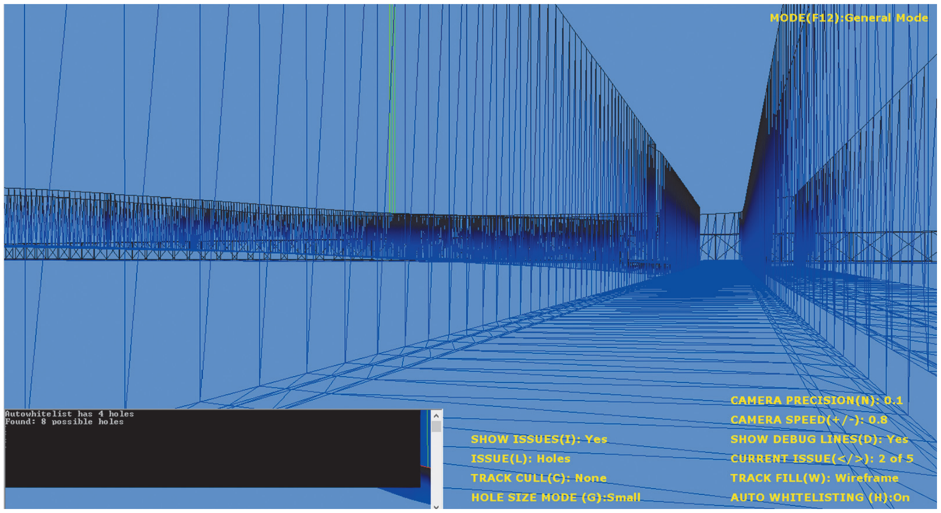
Figure 3.7. By-design skewed normals to simulate the effect of bumpy surfaces on the tires.

Later we arrived on a heuristic that works quite well for our meshes to detect those bad skewed normal. Our normals are exported with a 64-bit precision, and in the tool we recalculate the normal with the following non-weighted formula:

$$\text{normalize}\left(\sum_{i=0}^{k} n_i\right),$$

where k is the number of faces in the one-ring neighborhood of a vertex and n_i is the face normal of each triangle. We calculate this formula with 32-bit precision. After this, we have two set of normals: the original one with 64-bit precision and the calculated one with 32-bit precision. We then compare the two normals of each set; if their directions differ more than a specific threshold, it is highly likely that there are bad skewed normals in that area of the mesh and we flag it. This was done to simulate the behavior of the exporter in order to make a meaningful comparison and catch less false positives.

This method has proven to be a good heuristic; however, it can still cause false positives sometimes. When looking at those issues, we ask the testers to pay attention to the area around the flagged vertex and to see if all normals are following a well-behaved pattern. The threshold for comparing the two sets of normals is predefined for the user, but it can be modified at runtime in case some weird physics behavior is still happening in the area and the tool is not flagging anything. As the threshold gets smaller, there will be more false positives flagged.



```
Autowhitelist has 4 holes
Found: 8 possible holes
```

```
MODE(F12):General Mode
CAMERA PRECISION(N): 0.1
CAMERA SPEED(+/-): 0.8
SHOW ISSUES(I): Yes            SHOW DEBUG LINES(D): Yes
ISSUE(L): Holes                CURRENT ISSUE(</>): 2 of 5
TRACK CULL(C): None            TRACK FILL(W): Wireframe
HOLE SIZE MODE (G):Small       AUTO WHITELISTING (H):On
```

Figure 3.8. For the same small hole from Figure 3.2, the user can easily find it with the help of the green lines.

3.3.5 Detecting Malformed Triangles

Malformed triangles are simple to detect. When traversing the mesh to fill the half-edge data structure, we look at the triangle data and see if any triangles have vertices set in the same coordinates or if the three vertices are collinear. Those issues are flagged by the tool.

3.4 Visualization

Our visualization scheme has proven to be simple and effective. Usually the users of the tool are not very technical, and when designing the tool, we took into consideration that visualizing the issues should be very straightforward. The tool has basically two modes. The *general mode* is used to display holes and malformed triangles, and the *normal mode* is used to display flipped and skewed normals. In each mode, the user selects the type of issue that he wants to visualize (i.e., holes) and all of them are highlighted. The user can then loop through them by using the keyboard arrows, and while he is looping, a set of green lines that goes from each of the highlighted vertices to very high up in the Y-axis appears. Those green lines are extremely useful to actually find where in the mesh the issue appears; see Figure 3.8.

3.5 Navigation

As mentioned in the previous section, the users of the tool are not very technical, and our navigation system should be as easy as possible for a first-time user to learn. For multiscale environments, 3D content creation packages usually use some sort of an arc-ball camera scheme to navigate in the scene. Although artists are usually pretty comfortable with such schemes, regular users may find it hard and nonintuitive at the beginning. The ideal navigation scheme for the testers would be a first-person shooter style, which they would find very familiar. The biggest problem for such a scheme in a multiscale environment is the velocity of the camera; sometimes the user wants it to be very fast to traverse a long distance, and other times one may need it to be very slow to take a closer look at a very small hole. To solve this velocity issue, we tried automated approaches similar to [Trindade et. al 11], where a dynamic cubemap is generated to calculate the distance between the camera and surrounding objects and to automatically adjust the speed based on the distance. This approach worked to some extent, but there were still very bad cases where the camera was going too slow or too fast, which caused frustration to the user.

After testing some possible navigation approaches, we found one that was the best cost benefit in terms of usability and learning curve for our environment. The camera starts at a default speed and the user can increase its speed linearly with subtle moves on the mouse wheel. Yet, quicker moves in the wheel will make it increase exponentially (doubling each time), and a threshold controls the lower and upper speed limit. We also have a shortcut bound to a hotkey for snapping directly to a particular selected issue. Although this is not a scheme used in first-person shooter games, we found that after a few sessions the user can use this scheme quickly and precisely.

3.6 Workflow

The workflow in the studio begins with the artist crafting content source files, then the track will be built with its collision mesh and every other piece of content into binaries that are ready to be read by the game at runtime. After the build finishes, the tracks sits on an escrow folder waiting to be promoted by a test pass; if every test criteria passes, the track is promoted and others in the studio will see it in the build. At the beginning we had a separate export process for the collision mesh from the 3D content creation package to a format that our tool would read. However, this caused too many synchronization-related issues. Sometimes the export process would fail and new collision files would not be created, and testers would do an entire test pass in an old mesh. Moreover, the export script had to always be updated if artists try to use different source files for the collision mesh; if the export process did not get an update, testers would also be testing a wrong collision mesh. To solve this problem, we got rid of the export process and made the tool read the same binary file that is read by the physics engine at runtime.

The tool also has a couple of nice features that improve testers' and artists' workflows when filing and fixing bugs. Whenever an issue is highlighted, the user can press a hotkey to output the coordinates of the issue in the 3D content creation package space. Thus, when fixing the bug, the artist knows the exact coordinates where the hole is. Also, every time the tester presses "print screen" while in the tool, a screenshot will automatically be saved in a user folder with the type and the number of the issue, which makes it easier for the tester to navigate to the tool, take screenshots with the coordinates of every bug, and later file them all.

3.7 Conclusion

This chapter presented a 3D visualization tool for detecting collision mesh issues. This tool was used in production, and we were able to save hundreds of manual testing hours during development by using it. Our goal is not only to provide a solution to this particular problem but also to hopefully inspire the readers to use computer graphics techniques to solve problems in other domains, as it was shown with our testing problem.

3.8 Acknowledgments

Thanks to Zach Hooper for constantly providing feedback in the development of this tool and to Daniel Adent for the support on publishing this. Special thanks to my wife and family for all their help and to my friend F. F. Marmot.

Bibliography

[Botsch et al. 10] Mario Botsch, Leif Kobbelt, Mark Pauly, Pierre Alliez, and Bruno Levy. *Polygon Mesh Processing*. Natick, MA: A K Peters/CRC Press, 2010.

[Kettner 99] Lutz Kettner. "Using Generic Programming for Designing a Data Structure for Polyhedral Surfaces." *Computational Geometry* 13.1 (1999), 65–90.

[Mäntylä 88] Martti Mäntylä. *An Introduction to Solid Modeling*. New York: W. H. Freeman, 1988.

[McGuire 00] Max McGuire. "The Half-Edge Data Structure." http://www.flipcode.com/articles/articlehalfedgepf.shtml, 2000.

[Trindade et al. 11] Daniel R.Trindade and Alberto B. Raposo. "Improving 3D Navigation in Multiscale Environments Using Cubemap-Based Techniques." In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pp. 1215–1221. New York: ACM, 2011.