Polygon Reduction for Collision using Navigation Voxel and QEM

Eitaro Iwabuchi
Tokyo, Japan
iwabuchieitaro@gmail.com

Koji Mikami
Tokyo University of Technology
Tokyo,Japan
mikami@stf.teu.ac.jp

ABSTRACT

This paper presents an efficient approach for polygon reduction for collision. In general, collision models are used to detect intersection with the player model. In game production, an artist can use software like SIMPLYGON (by Microsoft Corporation) with a weight painting future to create a collision model. Weight painting work is time consuming for the artist because there are a lot of environment maps that need to be painted. Also, a lot of teams don't have access to software like SIMPLYOGN and need to create the collision models by hand. This paper proposes an approach to detect an area that a player can walk around in the form of navigation voxels. This means it can detect which areas need more polygons and which areas need less. Then it can use this information to reduce the polygons where needed. This method keeps the artist workflow in mind and it can reduce amount of work the artist is required to do.

CCS CONCEPTS

• **Computer methodologies → Mesh geometry models**

KEYWORDS

Collision, Polygon Reduction, QEM, Real-time

ACM Reference format:
Eitaro Iwabuchi, Koji Mikami. 2018. Polygon Reduction for Collision using Navigation Voxel and QEM. In international Conference on Virtual Reality Continuum and its Applications in Industry (VRCAI '18), December 2-3, 2018, Hachioji, Japan ACM, New York, NY, USA, 4 pages.
https://doi.org/10.1145/3284398.3284428

1 INTRODUCTION

Making the collision models takes a lot of time in game production. Simplifying the collision workflow can reduce the amount of work and time spend for the artist in production.

There is a lot of back and forth when making these collision models. Even though collision models are usually created in the final phase of development, whenever the direction changes happen it can drastically increase the amount of work for the artists. Collision models are always similar to the visible models, but the way these models are used and created is completely different. We should consider both differences. On the one hand you want to reduce polygons as much as possible. But on the other hand, you do not want the feet of player character to be hidden by the visible ground polygons. This means that it requires more polygons around the area where the players walk around. In initial phase of game production, they use the visible model as collision model in engine because they don't have enough time to create a collision model. The downside of using the visible model is that its causes performance degradation which slows down the production progress. The method we present in this paper can work around these issues.

Figure 1: The feet of the character are hidden by the visible ground polygons.

2 ALGORITHM SUMMARY

This section shows the overview our algorithms. First, we explain the data structures used in our algorithm. We are using 3 types of the data structures.
"*Simple 3-dimensional array*": It is constructed as simple array. The algorithm needs to search neighboring voxels. This array can easily access its neighbors using the index numbers.
"*Stack for searching*": This is a vector constructed as a stack. This is used as a temporary stack for searching the navigation voxels.
"N*avigation voxel Octree*": It is constructed as an octree. This is used for when the system paints the vertex colors. It is referenced by each vertex.

We are using these data structures for the following steps:

Step 1: Convert the collision meshes to voxels and store them in a "Simple 3-dimensional array".
Step 2: Give a start position for searching valid navigation voxels.
Step 3: Create the navigation voxels.
Step 3.1: Push initial voxel in to the "*Stack for searching*".
Step 3.2: Pop one voxel from the "*Stack for searching*".
Step 3.3: Check above the voxel with a distance of the character size. If there are no other voxels above the voxel, judge the voxel as valid and add it into the "*Navigation voxel Octree*".

Step 4: Next check for other voxels North, East, South and West from the popped voxel as seen from the top.
Step 4.1: First, check above the new position, then the position itself, afterward under the new position. (Figure 4). The first voxel that is found will be added to the "*Stack for searching*" and the remaining checks will be omitted.
Step 4.2: Repeat Step 3.2 to Step 4.1 until all voxels have been searched.
Step 5: Paint vertex color referring to the "*Navigation voxel Octree*".
Step 6: Reduction of polygons using QEM and referring to the vertex colors.

3 ALGORITHM DETAILS

In this section, we explain each step of the algorithm in detail.

3.1 INPUT DATA

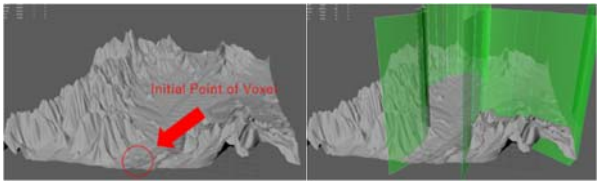

Figure 2: Input information. The left image shows its given initial position as locator position. In the right, the green polygons are given as wall collision.

These parameters are given as input data for the system.

- The target meshes.
- The ground collision meshes (same as target meshes)
- The wall collision meshes.
- Initial position for searching valid navigation voxels.
- Character size for detecting valid navigation voxels.
- The number of polygons for reduction target.

The ground meshes will be reduced by the system. The collision meshes are the same as the ground meshes in this case. The wall collision prevents the character to enter the prohibited area. It a common method in productions, (Figure 2). Initial position will be used in our algorithm for searching the "*Navigation voxel Octree*", (Figure 2)

The ground meshes will reduce until the number given by parameter.

3.2 VOXELIZE

For searching valid voxels, we should voxelize the collision meshes. The level scale is different between all game projects, we get this voxel size from the input.

Our system can handle multiple collisions meshes. Because environment levels always are created by combination of assets. There are usually no scenes created by a single mesh. We adopt "S*imple 3-dimensional array*" for storing the voxels. We need to access each voxel with index (x, y, z). When we try to access the next voxel, we just increment or decrement the x (y, z) count. This method can reduce calculation cost.

As a result, we can efficiently get the voxel array for searching.

3.3 NAVIGATION VOXEL

In this section, we explain the how to construct the data structure of navigation voxels. This process uses the 3 types of data structures we explained in section 2. "*Simple 3-dimensional array*", "*Stack for searching*", "*navigation voxel Octree*".

3.3.1 DEFINITION OF VALID VOXELS

Here we show how we decide what a valid voxel is. To do this, we use voxel size and the height of the character. The voxels that do not have other voxels above them according to the character's height are labeled as valid voxels. If there are voxels above the voxel, these voxels are invalid, Figure3. If voxels are valid we store that voxel to "*Navigation voxel Octree*".

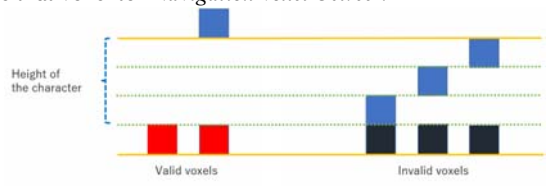

Figure 3: The red voxels are valid voxels. The black voxels are invalid voxels and the blue voxels are other voxels have not been checked yet.

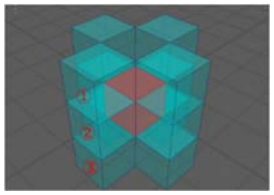

Figure 4: Search steps. The red voxel popped from stack. The voxels around are the ones being searched.

3.3.2 CONSTRUCT NAVIGATION VOXEL

This section explains how we construct the "N*avigation voxel Octree*".

First, detect the initialized voxel using the position given from parameter. If the initialized voxel is valid, push to *"Stack for Searching"* and start searching voxels. If not just show error message and stop searching. In Figure 4, the red voxel is target voxel popped from *"Stack for Searching"*. The system checks if this voxel is valid or not. If it is valid voxel, it added to "N*avigation voxel Octree*".

Then, we assume target voxel as (x, y, z) coordinate of *"Simple 3-dimensional array"*. We use Y-up coordinate because by default Maya is Y-Up. We search the voxels on the North, East, South and West side of the red voxel. The North means (x, y, z-1), the East means (x+1, y, z), the South means (x, y, z+1) and the West (x-1, y, z), Then we search the top and bottom voxel of each sides. North top voxel means (x, y+1, z-1), North bottom voxel means (x, y-1, z-1), if the above voxel is valid, it means

voxels below that voxel are invalid. So, the system stops searching as an optimization.

If the system finds a voxel it will get pushed into the *"Stack for Searching"*. Each voxel in *"Simple 3-dimensional array"* has a "searched" flag . Because this algorithm can check the same voxels several times in the searching process. For optimization we added this flag, the first time a voxel is searched it will get changed to "True". The next this voxel is encountered, it checks this flag and avoids doing the same calculation. After that it continues the process until the *"Stack for Searching"* is empty. When the processes ends, we get the Navigation voxels in "N*avigation voxel Octree*". It shows in Figure5.

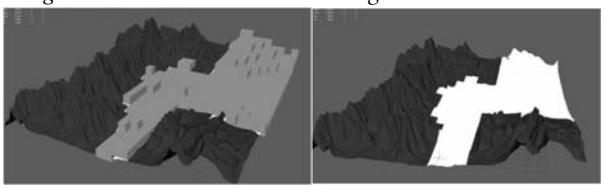

Figure 5: The left image is shows the navigation voxels that are generated. The right image shows the result of the vertex color.

3.4 PAINT VERTEX COLOR.

After we got "N*avigation voxel Octree*", we paint the vertex colors of the area of where the character can walk using "N*avigation voxel Octree*". The reason of painting the vertex color is because we can use this information for the polygon reduction process with QEM by [Garland and Paul 1997].

We check each vertex if it overlaps on a navigation voxel. If so we use the character height as distance from the vertex to the voxel. If the vertex is close enough to the voxel the vertex gets painted 1.0 Otherwise it will receive 0.05. In order not to create a singular point in space, we don't use 0.0. As a result, we get the area where the character can walk as a vertex color. Figure5.

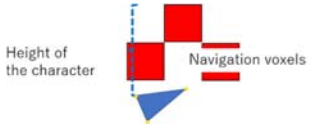

Figure 6: Paint vertex colors using navigation voxel.

3.5 REDUCTION USING QEM

QEM (Quadric Error Metrics) [Garland and Paul 1997] is major method of polygon reduction please refer to original paper.

3.6 CONSTRAINT VERTEX

The original QEM algorithm is designed for closed polygons. When this algorithm is applied to a planer mesh, it is different from the expected result. (Figure7 center image). To solve this issue, we added a constraint method QEM.

QEM algorithm has a "cost". When I get the constrained vertex information from input, the cost calculates plus 1.

$$new\ cost = cost + 1 \tag{1}$$

After increasing the cost, we reorder the valid pairs in the heap. The priority of the pairs is lowered and as a result the pair is not contracted. This algorithm is not perfect because if the number of constraints vertices is greater than the number of targets the constrained point also contracted. But it can be used in any cases without problems. Finally, we can get the expected result by using this method. (Figure 7 right image).

3.7 VERTEX COLOR AND PENALTY

We reduce the target meshes using the vertex color we explained.

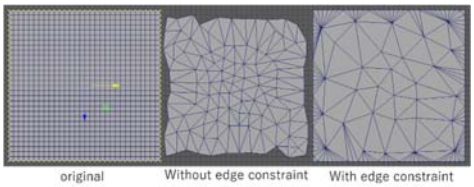

Figure 7: Constraint edge. On the left is the original is at 1800 polygons. The center plane is 179 polygons, and at the right is 180 polygons. We input the constrained vertices as a set to the program (edge vertices on the original plane.).

We already have the area of where the character can walk as the vertex color (in section 3.4). Next we also give the penalty from the user input.

The same idea is explained in section 3.6. The constrained vertices are expected never reduce. But the remaining vertex color area is expected to reduce in polygons. So, we figure out two methods about cost modification.

In the QEM algorithm, the pair contains two vertices v1 and v2. Each vertex also has vertex color (VC1 and VC2).

Method1:

$$new\ cost = cost * (1 + (VC1 + VC2) * penalty) \tag{2}$$

Method2:

$$new\ cost = cost + (VC1 + VC2) / 2 * penalty \tag{3}$$

The idea of method1 is to multiply by the result calculated from the values of the vertex color and penalty. This method results in a somewhat smoothed version of the original mesh. The result looks averaged.

The idea of method2 is to add the result calculated from the values of the vertex color average and penalty. This method keeps the harder edges of the original mesh. This is ideal for the area where the character can walk but there are few incorrect polygons.

4 RESULTS

This section shows the result of our method. First, we use the target mesh that has 175232 triangles polygons. This geometry consists of simple grid of polygons that were deformed by using

a height-map. We tested method1 and method2, reduce 94% and 98% and change penalty (10, 100, 1000)

We also show the ideal shape created by hand in Figure8.

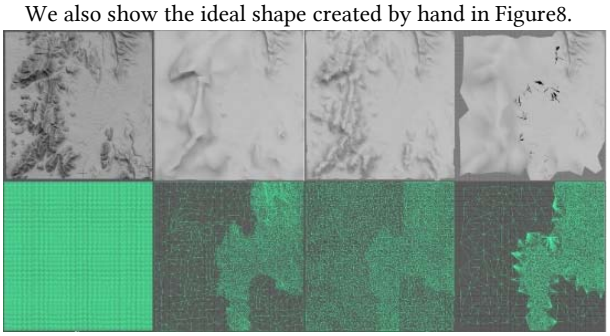

Figure 8: From left, original mesh - 175232 polygons, reduced by hand - 10683 polygons, method1 98% reduction with penalty 1000 - 10513 Polygons, method2 98% reduction with penalty 1000 - 10513 Polygons.

Table 1: Time Measurements

Target polygon count	Reduction polygon count	Method type	Penalty	Time
175232	10683	By hand	---	5h 30m
175232	10513	Method1	10	82.766 s
175232	10513	Method1	100	82.571 s
175232	10513	Method1	1000	84.159 s
175232	3504	Method1	10	82.545 s
175232	3504	Method1	100	82.401 s
175232	3504	Method1	1000	82.614 s
175232	10513	Method2	10	81.848 s
175232	10513	Method2	100	83.717 s
175232	10513	Method2	1000	89.644 s
175232	3504	Method2	10	81.833 s
175232	3504	Method2	100	83.249 s
175232	3504	Method2	1000	89.294 s

In method1 we are able to reduce polygons and keeping the shape that was painted by the vertex colors. However, the difference between the area with and without vertex colors is minor. Instead overall reduction is being done. Also, when raising the penalty, you can control the percentage of shape retention stepwise. In the case of Method 2, reduction is also keeping the shape of the painted by vertex color. However, compared to Method1, the shape is more strongly maintained. If you increase the reduction ratio, the influence of reduction is also increased. We can control the penalty, but it maintains a stronger shape. But unfortunately, as a result of the reduction, some polygons are incorrect as they are collapsed to one single point. When the reduction is high, Method1 gives the correct result. And when the reduction ratio is low, Method 2 gives the correct result. Also, the calculation speed of the, method and amount of reduction were measured. There is no significant change in speed when changing the penalty. However, when comparing to creating the same result by hand, we were able to greatly speed up the process. Finally, we tested using the data from an actual game production. Here we reduced the mesh from 999185 polygons to 19983 polygons. The result is good. The character can walk around this 3D model without any problems. If we compare to create everything manually, we were able to reduce the time considerably. We can now use efficient collision meshes even in the pre-production stage. This greatly speeds up the efficiency of any project. By doing so we achieved our original purpose which was to improve the production efficiency of the artist.

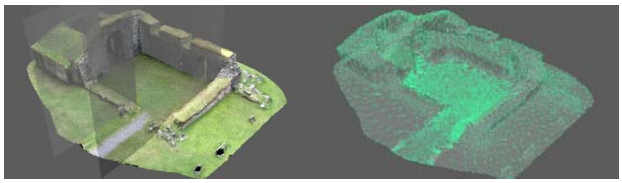

Figure 11: We tested using data from a real production. The top Image is original data of a ruin. The bottom data is the result of our approach.

5 DISCUSSIONS

We still have few issues.

1. Method1 and Method2 are not perfect. Method 1 does not reduce in the right areas. Method 2 sometimes creates collapsed polygons. Ideally, we would like to merge the two methods and take the best of each method

2 noise remains. Either method can produce noisy results near edges.

3. Sometimes the constraint algorithm does not work properly. If the reduction ratio is set too high, the vertices specified constraint might be reduced.

We succeeded in shortening the current production time, however there are still many areas to improve. Possible improvement can be expected by implementing a different logic for example by [Garland and Paul 1998], [Hoppe 1999].

6 CONCLUSIONS

In this paper we proposed a new method to efficiently reduce collisions. Our method uses Navigation Voxel which is intermediate data when creating Navigation Mesh, and it is possible to detect the place where the character walks. Then, it can create a collision that can express the ground of the character. We understand the remaining issues about regard to formulas that give penalties on costs, so we need to continue researching. Finally, when compared with the workflow created by the artist manually, we get a much more effective result about speed.

REFERENCES

Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics In SIGGRAPH '97 Proc., pages 209-216.

Michael Garland and Paul S. Heckbert Simplifying surfaces with color and texture using quadric error metrics VIS '98 Proceedings of the conference on Visualization '98 Pages 263-269

Hugues Hoppe. Microsoft Resear New quadric metric for simplifying meshes with appearance attributes, VISUALIZATION '99 Proceedings of the 10th IEEE Visualization 1999

the model Olavskirken ruin by Telemark fylkeskommune.
https://sketchfab.com/models/3a6c7c3425cb4cfa874eecf0860365d0?utm_source=triggered-emails&utm_medium=email&utm_campaign=model-downloaded