Real-time mesh cutting

Project report for the course
"Real-time graphics programming"
at Università degli studi di Milano

Pietro Prebianca - 921582

Abstract

One of the most common operations performed during 3d modeling is the cutting operation; to execute such operation, the modeler provides a cutting plane to the application, which subsequently produces two new meshes, eventually filling the two missing sections with new triangles.

Although this operation can be performed not necessarily in real time in modeling applications, it must be executed as soon as possible in interactive applications such video games; one of the most famous of these is "Fruit Ninja"[5] from Halfbrick studio[4].

This report describes implementation details of a "Fruit Ninja" clone, and its performances on different 3d meshes.

1 Introduction

The goal of this project was to build a fruit ninja clone for PC, and analyze its performance on different meshes. "Fruit Ninja is a very famous mobile game, in which the player cuts fruits that are thrown in the player view; these fruits move following parabolic trajectory in the player's view, before falling outside the scene for gravity. To implement such application, I relied mainly on these libraries: OpenGl[3] and Bullet[2].

OpenGL is a cross platform widely used API that allows to render 2D, 3D objects, control and customize part of the rendering pipeline; while for the physics simulation I relied on Bullet, a free and opensource physics engine that offers collision detection, soft and rigid body dynamics. In particular, the version of these libraries are respectively 3.3(OpenGL) and 2.84(Bullet). All mesh used during the application testing are OBJ files loaded through the Assimp[1] library(5.0). The language chosen to build such application is C++, due to its great qualities such as great memory control and the possibility to build and execute on different platforms.

The rest of the report is structured as follow: section 2 describes the scene structure of the application; section 3 talks about the method I implemented to slice objects, while section 4 talks about performances, highlighting bottlenecks and limits.

Finally the report ends with some considerations about the work done and proposing hints for future improvements. The source code of this project is available on GitHub at this link.

2 Application general description

The application implemented displays a scene in which, a set of predefined meshes are threw once at time, in the view frustum. Once an object falls outside the view frustum, it's deallocated and another mesh is loaded and thrown again and the cycle continues. During the loading of the mesh, in addition to the creation of its indexed mesh data structure, the application generates also its collision proxy for it, using part of the points read from the mesh file; Bullet offers many type of collision proxies that we can use: capsules, boxes, spheres, 2d boxes and many others. In order to avoid troubles derived by the usage of different proxies, and to make the application works with any mesh, I chose convex hull for the physics simulation.

Produced convex hull are passed to the physics engine, which place them under the view frustum; once a mesh is place, the physics engine adds it to the simulation and applies a little impulse to it, in order to make it fly in the view frustum. Since during the simulation, the user can cut object, new convex hulls are produced every time a cut occurs.

Figure 1 depicts the scene structure, while Figure 2 a cut example.

3 Mesh cutting

3.1 Exclusion ray-cast

Considering that the cut operation is expensive, an exclusion test is needed.

This test can be implemented with a simple ray cast, between the starting and ending point of the cut segment. Since the simulation is ran in world space, a ray must be expressed in world coordinate aswell. All objects move following a trajectory that lies over a plane centered in the origin; to project screen coordinates over this plane, we must first, convert screen coordinates to NDC coordinates.

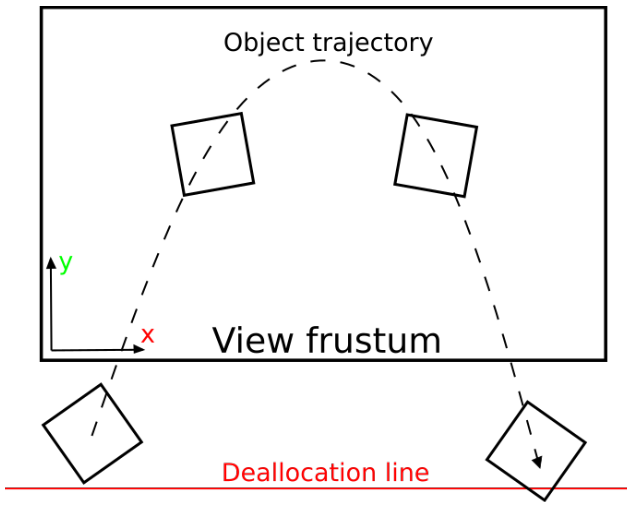

Figure 1: Scene structure.

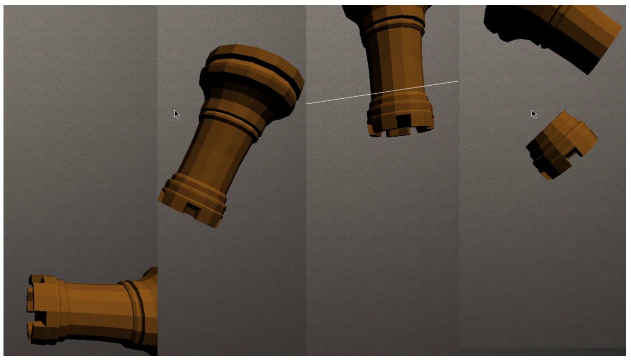

Figure 2: Cut example on the rook model.

x and y NDC coordinates can be calculated with this formula:

$$x = (2\frac{x_{mouse}}{screen_{width}}) - 1$$

$$y = (-2\frac{y_{mouse}}{screen_{height}}) + 1$$

The z component is constant. Finally, to convert everything in world space coordinates we have to multiply this vector with the inverse of the multiplication between the view and projection matrices.

3.2 Cut operation

If a mesh passes the exclusion test, then a cut must be performed; each cut produces two new meshes called positive and negative. The positive lies in the positive half space, inducted by the cutting plane, while the other in the negative (Figure 3).

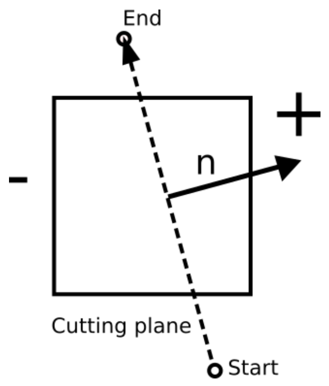

Figure 3: A mesh intersected by the ray defined by start and end points; the dashed line is the cutting plane, while $\vec{n}$ is its normal.

To produce 2 different meshes, we have to create two indexed mesh data structures; each of these structure will be populated with:

- Existing triangles
- New triangles over the section
- Other new triangles to fill the empty section

Existing triangles are those which are not involved in the cut, while all the others are new triangles built on the border of the cutting section and in the cutting section.

3.2.1 Triangle not intersected with the cutting plane

Let's start with the simple case; the involved triangle can be only in the positive or negative half space. To understand where a triangle is, 3 half tests(respect to the cutting plane) are done; if the majority of these test returns a positive value, then the triangle is in the positive half space, otherwise in the negative one.

This means that, if a point touches the plane, the triangle is considered as if it weren't intersecting the plane.

Before adding blindly a triangle to the positive or negative indexed mesh data structure, the application checks if the new vertex is already present.

The most naive method is to loop over all points of the fragment, and check if that point already exists; this surely works, but it's extremely inefficient.

A way better approach can be obtained with hash tables instead; each point added to a fragment, is added to a dictionary as well. This dictionary maps vertices to their indices; so, before adding a vertex, the program controls if there is already that key in the dictionary: if there is, then the index is retrieved, otherwise it adds the new vertex and update the dictionary.

3.2.2 Triangle intersected with the cutting plane

Depending on the direction of the cutting segment, we can intersect a triangle in 6 ways; let's use Figure 4 and 5 as references, to understand how this process works.

The first problem is to find the intersection points between the cutting plane and the triangle; given an edge of the triangle, a plane point and the cutting plane normal, it's possible to find the interpolation factor where the intersection happens. The following formula calculates the interpolation factor on the segment BC.

$$d = \frac{(\vec{p}-\vec{b})\vec{n}}{(\vec{c}-\vec{b})\vec{n}}$$

where $\vec{p}$ is a point over the cutting plane, $\vec{b}$ and $\vec{c}$ are the two vertex that defines the edge while $\vec{n}$ is the normal of the cutting plane.

At the end, we obtain two interpolation factor that falls in the interval $(0, 1)$; the limit case in which a triangle touches the plane in just one point, is managed as if there were no intersection.

With these interpolation factors, it is easy to calculate the new intersection points.

$$\vec{v} = \vec{c} * d + \vec{b} * (1 - d)$$
$$\vec{v'} = \vec{a} * d' + \vec{c} * (1 - d')$$

The same interpolation can be applied also to the other attributes such as normals, texture coordinates.

Now we have 5 points: 3 from the original triangle and 2, derived from the intersection. According to where $\vec{a}, \vec{b}, \vec{c}$ are respect to the cutting plane, we have to produce and assign 3 new triangles; in this example, two triangles are assigned to the positive fragment, while the other to the negative.

So we add these points to the appropriate data structures, always checking for duplicates, in the same fashion we did before.

All other cut situations work in a similar way.

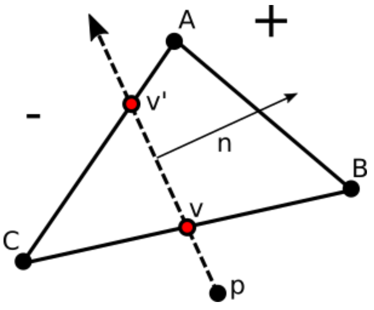

Figure 4: Triangle cut example.

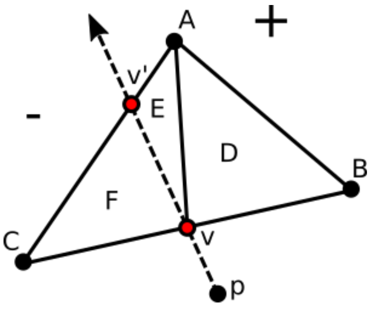

Figure 5: Triangles produced after cut.

3.2.3 Face filling

Yet, I didn't mention how the application fills the empty space generated by the cut; the method implemented to cover this aspect is straightforward, but has a drawback: it works accurately with convex sections. It is possible to fill the missing part of each fragment by creating a fan of triangles, considering all new points generated by the cut; each piece of the fan is defined by the section centre and 2 points on the border. These two points are those, which we generate by interpolation during the cutting loop; in other words, whether a triangle intersects the cutting plane, the application produces 3 new border triangles, plus 2 new section triangles(one for the positive section, one for the negative). But the problem is that we don't know the section centre, at the beginning of the cutting process; so the application sets initially the section centre's position to a temporary position. As we find triangles to be cut, the application sums all points generated by the cutting plane, counting also their number.

At the end, we divide this point sum with the number, to obtain the correct geometric centre of the section; Figure 6 provides a visual explanation.

This procedure fills the missing sections, but the resulting rendering would produce black sections, because we didn't update section normals;

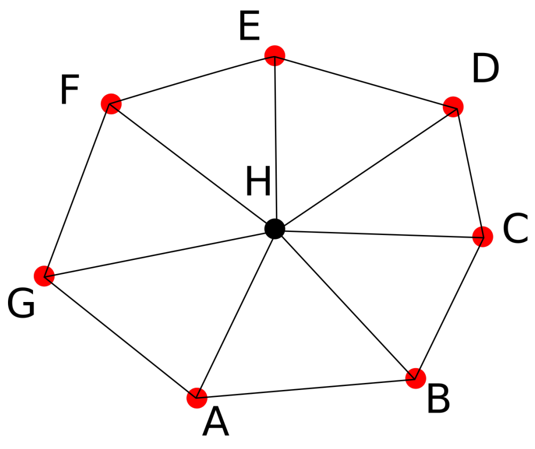

Figure 6: During the cutting loop, the application produces the red points(A, B, C, D, E, F, G) by interpolation: these points are summed (as they are generated); the resulting point at the end of the cycle, it's finally divided (in this case by 7). In this way we obtain the correct geometric centre H, which is used to update the temporary value in the indexed mesh data structure.

to fix this, we just need to set the normal for all section points in this way: we assign the normal of the cutting plane to negative vertices on the section, and the flipped normal for positive ones. Figure 7 depicts how vertex normals are arranged at the end.

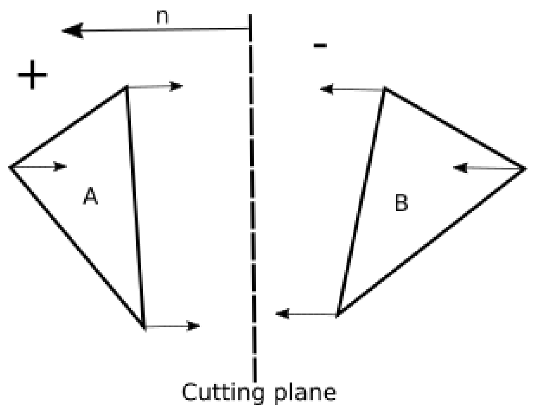

Figure 7: How normals are disposed for vertices of A and B triangles.

This method is accurate only with convex sections; for concave sections, more sophisticated methods are needed.

3.2.4 Convex hull generation

The physics engine detains a convex hull for every visible object; as the user cuts objects, new convex hulls are needed to provide a physics simulation for all fragments.

After the cutting loop, we must loop again over all points of these two fragments, for two main reasons; the first reason is because, the vertices local coordinates of fragments we've just created, are expressed according to the old mesh pivot. During the cutting loop, the application calculates also the new pivots for both fragments. The second reason is to create their convex hulls.

At the end of this process we have 2 meshes and their respective convex hulls.

4 Results

The following performances have been recorded on a laptop with these hardware specs:

- CPU: Intel Core i7-7700HQ 2.80GHz
- GPU: Nvidia GeForce GTX 1050
- Ram: 16 GB

These tests are related a set of meshes composed by some primitives, chess pieces and other objects (Figure 8). For each model, I've recorded 2 average times (average cut time and average convex hull building time) and frame rate range(maximum and minimum). Average times have been calculated on 10 cuts.

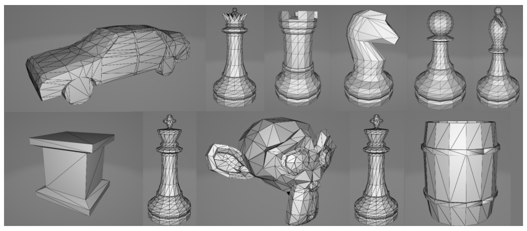

Figure 8: Some models used for for the project.

Table 1: Number of faces, vertices and edges for each model, used during the evaluation.

Model	# Faces	# Vertices
Car	2836	6184
Cube	24	12
Rook	926	1790
Pedestal	44	88
Horse	446	882
Icosphere	80	240
Bishop	1216	2354
Cylinder	124	192
Pawn	898	1800
Cone	62	128
Barrel	706	1263
King	965	1895
Sphere	760	445
Queen	1059	2137
Monkey	968	1966

Table 2: Average time for cutting and convex hull construction operations, with FPS peaks and drops, for each model, on 10 cuts.

Model	Avg. cut time (ms)	Avg. convex hull building (ms)	FPS
Car	747,159 ms	275,180 ms	60-12
Cube	0,099 ms	0,010 ms	60-57
Rook	19,552 ms	10,985 ms	60-56
Pedestal	1,703 ms	0,04 ms	60-58
Horse	11,881 ms	7,016 ms	60-56
Icosphere	2,300 ms	0,70 ms	60-56
Bishop	52,290 ms	25,58 ms	60-54
Cylinder	4,294 ms	1.40 ms	60-57
Pawn	28,633 ms	14,876 ms	60-57
Cone	1,899 ms	1,41 ms	60-56
Barrel	55,17 ms	20,46 ms	60-52
King	31,024 ms	14,985 ms	60-53
Sphere	15,556 ms	7,670 ms	60-57
Queen	33,245 ms	20,423 ms	60-53
Monkey	32,783 ms	18,243 ms	60-52

Despite bad results for the car model, the proposed method works fairy well on meshes with less than 3000 vertices; since the original "Fruit Ninja" was developed for the mobile devices, the resolution of meshes were not so high, so this project can be a valid solution with due precautions.

5 Conclusions and considerations

In the previous chapters, we have covered a very basic way to implement mesh cutting; this approach is simple and works fine, but it has limitations for high resolution meshes.

The main problem yielded by this implementation is the number of operations is linear with the resolution of the mesh.

A possible optimization would be the following: rather than looping through all triangles, we should try to limit the number of operations during the cut, by considering only triangles that are involved in the cut. This is achievable with a an half edge data structure: once we find a triangle involved in the cut, all the others can be found in its neighbourhood. I tried this approach, but I obtained very bad result due to the overload caused by hash functions; hash functions are needed for two main reasons:

1. Find opposite edges easily
2. Find all edges leaving a vertex

So the original problem has became: how to hash efficiently mesh vertices? But even if, we solve this problem, the half edge data structure is used only in the application stage, for processing operations and it cannot be used directly by the GPU; so we would have to find a good way to produce an indexed mesh from an half edge data structure.

In other words we would have to do a number of operations which is linear in anycase.

Another possible improvement of this project is related to shaders. Right now, all cuttable objects are rendered with a simple Lambert illumination model, that doesn't take into account possible textures; but if we adopt a more sophisticated fragment shader that consider them, there will be discontinuities in the final appearence of the object. These discontinuity are due to the fact, that the application doesn't generate uv coordinates on section vertices.

Finally, rather than using convex hulls, which are great to achieve realistic simulations, but not that good to build at runtime, we can adopt more simple proxies.

References

[1] Alexander Gessler et al. *Open Asset Import Library*. `https://www.assimp.org/`.

[2] Erwin Coumans et al. *Bullet physics Library*. `https://pybullet.org/wordpress/`.

[3] Khronos Group. *Open Graphics Library*. `https://www.opengl.org/`. 1992.

[4] *Halfbrick Studios Pty Ltd*. `https://www.halfbrick.com/`. 2001-2020.

[5] Halfbrick Studios Pty Ltd. *Fruit Ninja*. `https://www.halfbrick.com/games/fruit-ninja`. 2010.