INF 585: Marble construction simulation

Guillaume Loranchet and Flavien Vidal

March 2021

1 Introduction

As part of the INF585: Computer Animation course, we decided to simulate a marble construction game. In the lab class, we already worked on the interactions between several spheres and a cube. Based on this, we wanted to extend the interactions to more complicated objects to create an interesting parkour. The first step was to go from an infinite plan to a finite cylinder. The second step was to make the intersection works for any arbitrary shape. And finally, we build the circuit using the different objects and created a loop from start to finish.

2 Implementation

2.1 Sphere to plane intersection

As said previously, in one of the lab classes, we studied the interaction between a sphere and a plane, given the position of the sphere, its velocity, a point in the plane, and its normal vector. With quite simple geometric operations, we could compute two velocity vectors: one perpendicular to the plan (measuring the impact and the bouncing of the sphere) and the other parallel (for the friction). By adding those two vectors, we obtain the final velocity vector. We can also add bouncing and friction coefficient to reduce the impact of a given direction. To determine the position of the collision, we first evaluate the distance between the sphere and the plan. A simple method to find an approximated position would be to move back in the normal direction of the plan until the ball is fully above.

2.2 Intersection with primitive meshes

With the previous method, we could model a cube with 6 planes and by the spheres inside. However, using a set of infinite plans becomes quickly an issue when dealing with several objects. For a cube, we can easily check if the sphere is located inside or not. For a cylinder, we may simply create a box around it and restrict its domain to the one of the box. However, we could be more precise. Let say a cylinder is defined along a vector $\vec{AB}$ with radius r. We first compute the orthogonal projection of the sphere center (P) on $\vec{AB}$, called d, with a dot product. We then check if d is indeed between A and B and if the distance between the center and its projection is smaller than the radius.

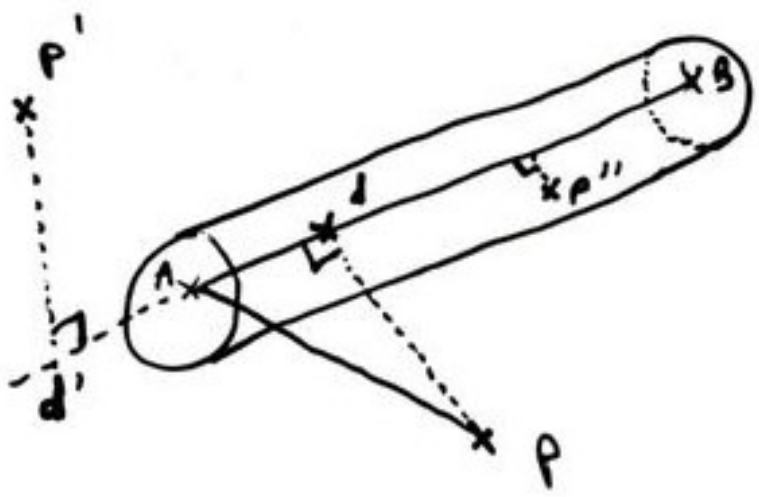

Figure 1: Sketch of the *IsInsideCylinder* method

2.3 Intersection with arbitrary meshes

The first method works well for simple objects, typically cubes or cylinders. Nonetheless, our goal is to insert more complicated shapes and to find an intersection algorithm that works even with no prior knowledge on the object. To do so, we use the fact that we are only working with triangulated meshes. We first determine if the sphere is colliding with any plan defined by each triangle. If so, we then project the position of the sphere center on the plan. We finally computed the barycentric coordinates to quickly see if the point is inside the triangle. The main advantage of this method is that it avoids costly computations for each triangle as we only have to do a simple dot product to see if the sphere is intersecting the plan. With this operation, we can already filter the majority of the triangles. It also benefits from the fact that we compute all the normals beforehand, by averaging the normal of each one of the triangle vertex.

2.4 Creation of the parkour

In the beginning, we place all the objects manually, defined by a translation, a rotation, and a scaling. It works for a few objects but it becomes tedious and not very practical, especially to have clean transitions between two objects. To facilitate the manipulations we added, for each object, an input and output point. We thus only have to place them once for each type of object (for example the spiral, or the in/out cylinders) relative to the center and update their positions when the object is modified.

Finally, we built the parkour iteratively by attaching the output point of an object with the input point of the one coming right after. With this method, if we move one object, the rest of the structure will follow accordingly. Finally, we wanted to make the circuit loop (as in the original toy game) to make the animation more satisfying. Instead of creating an elevator like in the toy game, we use the freedom given by the virtual scene. At the end of the parkour, we have an invisible cube that inverses the gravity of the marble if its center is inside the cube. We do the same to change again the gravity when it reaches the top.

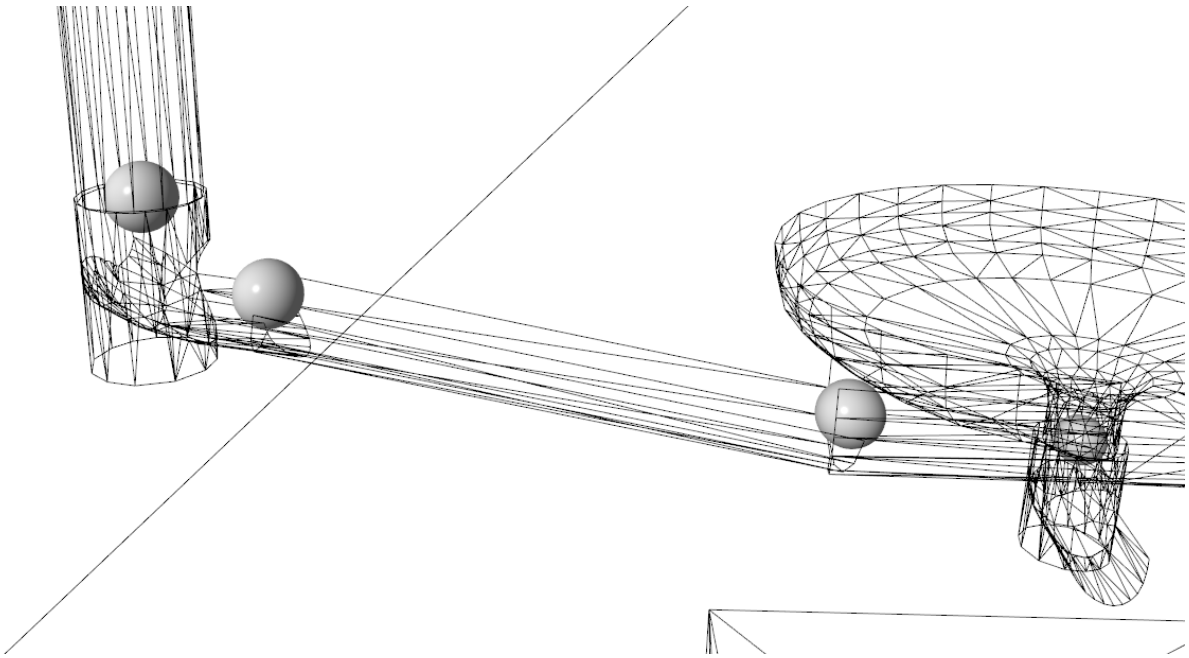

Figure 2: Balls representing the input and output points

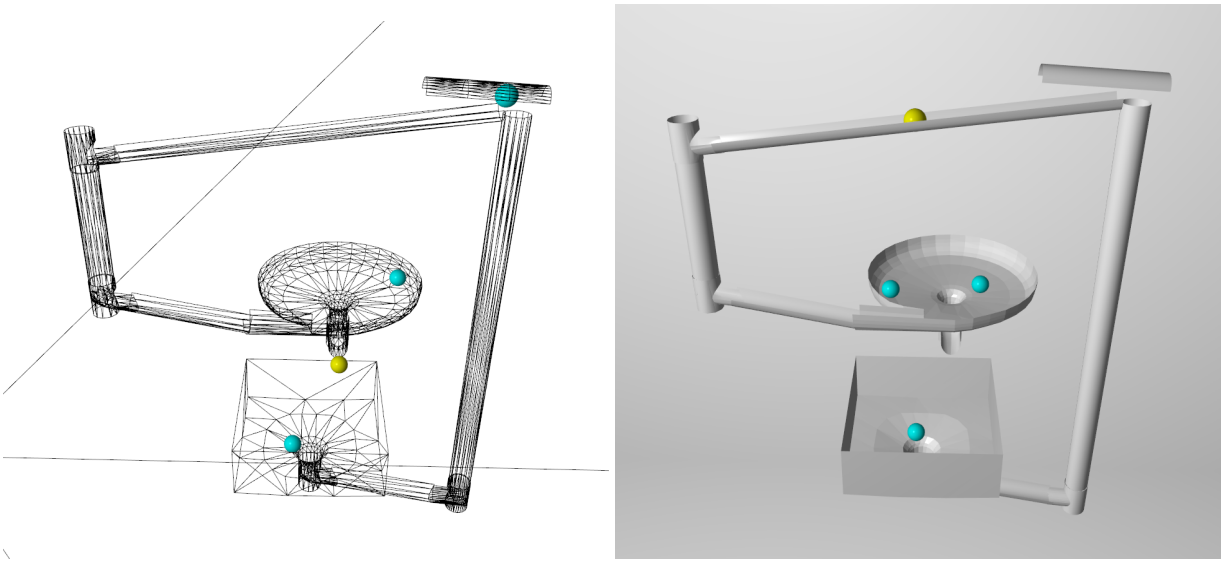

Figure 3: Final circuit (Left: wireframe, Right: solid)

2.5 Optimisations

However, adding non-primitive shapes quickly increases the number of triangles in the scene. As the intersection function has to a least do one operation for each triangle and each sphere, the computational time was too important (less than 20 frames per second for only a few marbles). The first step to optimize the process was to have bounding boxes around cylinders (and obviously cubes) and before finding any intersections, we check either the sphere is inside the box. If it is, we know the sphere is not intersecting any other object. It already improved quite a lot the frame rate but it was nonetheless still expensive when a lot of spheres were on the spiral (for example) at the same time. As the computational time was at least linear with the number of triangles, we decided to simplify our objects.

For that purpose, we used the decimate geometry and merge by distance tools in Blender. The first one simplifies the geometry by collapsing edges (seems to be the most effective to reduce the number of triangles in particular while preserving the overall shape) and the second simply merges vertices based on their proximity (see figure 4). With these two optimization schemes, we reach around 35 fps for more than 10 spheres.

Finally, we reduced the number of substeps when less necessary. To make the marbles realistic, they had to be fairly heavy which means they travel a longer distance for a given time step, and thus more chance to miss an intersection (side note: we also nearly removed the perpendicular component to avoid bouncing). To prevent this issue, we multiplied by 10 the number of steps. It is even more important to increase the number of steps as we are looking for an (almost) exact intersection. Indeed, it sometimes led to strange behaviors when we also considered the ball under the plan. However, we don't need that level of precision all the time. For example, we only check one time out of 10 for collisions between spheres or if the sphere is almost still.

3 Conclusion

In the end, we managed to have a looping circuit with different types of objects. The most challenging aspects were probably to find an intersection method that works with any arbitrary objects and to have an optimized program. From this point, the two improvement directions are to either add some lights and shadows to make the animation more satisfying to watch or to work on the interaction by simplifying the creation process. Finally, we could also enrich the set of objects to stairs or wheels for example.

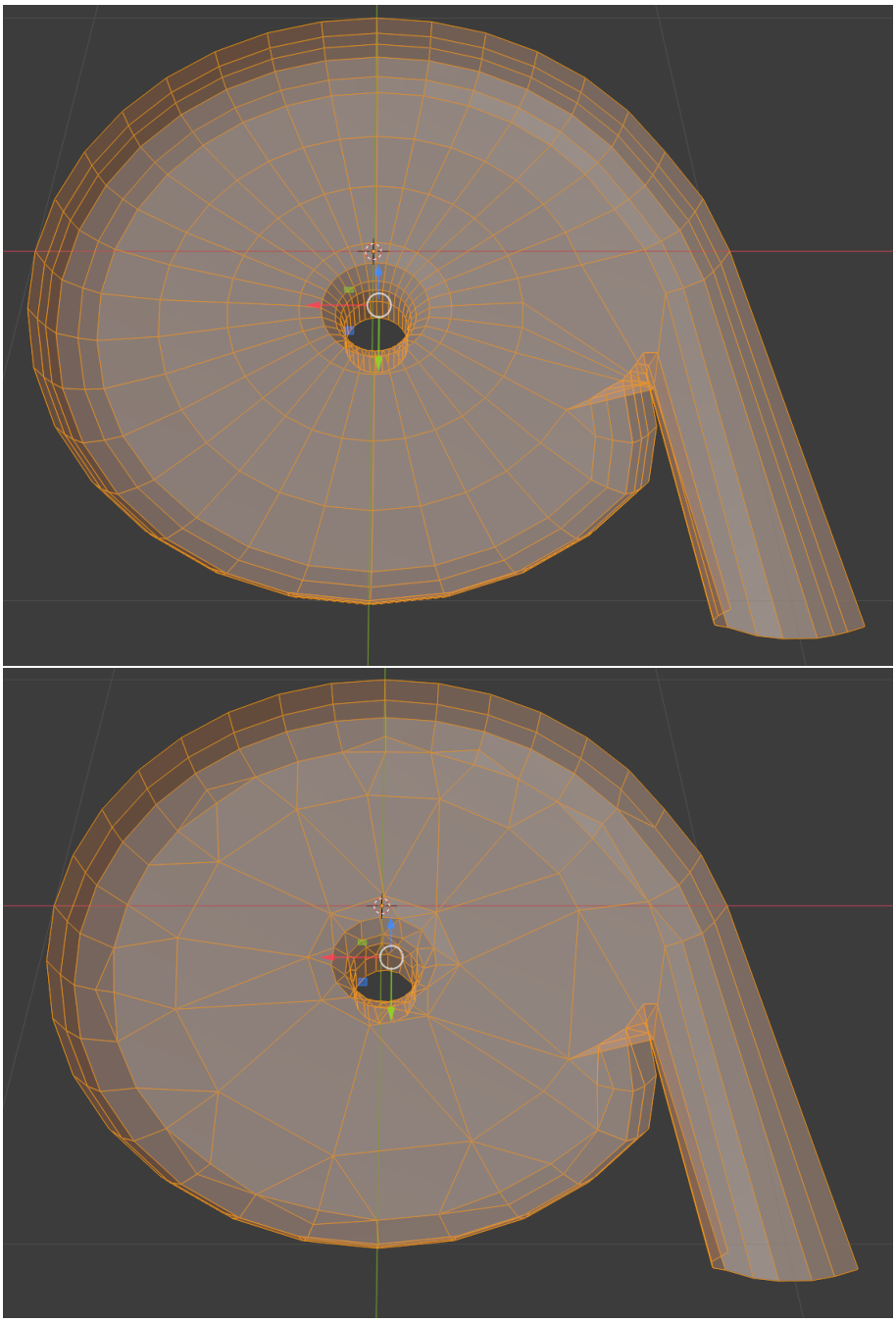

Figure 4: Top: Before decimation (835 triangles), Right: After decimation (416 triangles)