A US marine playing in a ball pit

ARTHUR TRAN-LAMBERT, Ecole Polytechnique

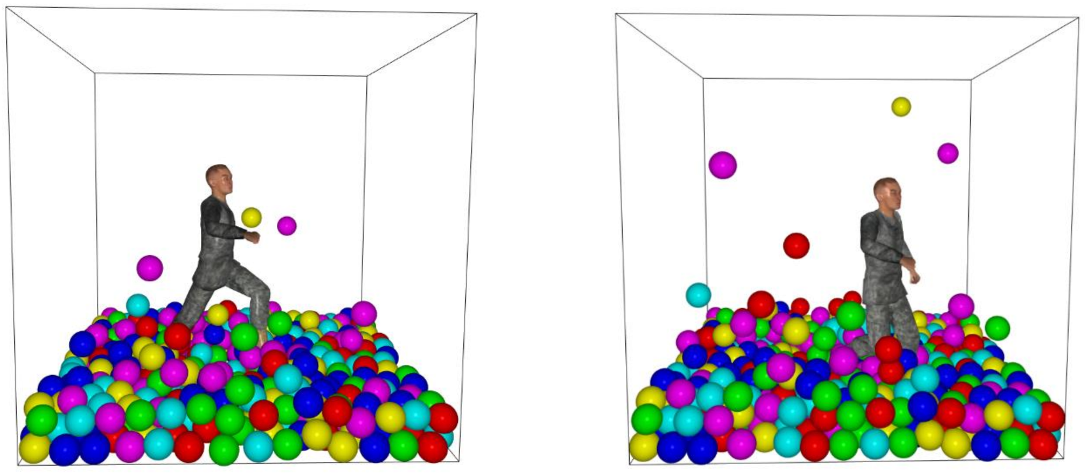


Figure 1. This work is part of the INF585 Computer Animation course and combines animation techniques such as character rigging and physically based rigid bodies simulation models to create an interactive demo of a marine evolving in a ball pit.

1 Introduction

For my final project, I chose to extend my favorite lab (Rigid spheres simulation) as playing around with these numerous colored spheres reminded me of these ball pools for kids, but this one was deserted so far and rather sad. To instill some life in it, what I am trying to do is to make it **interactive** with a character and recreate the kind of animation you can find in video games for example (a bit like at the start of this video from the latest FNaF game although I doubt they use actual physics to handle all their balls, in this case it looks more like sprites displayed in front of the camera). However, this will require a **sufficient acceleration structure** to handle all the additional collisions as I would like to use a rather **complex rigged character model** as well as a decent number of balls: right now the fps count drops pretty significantly when combining both.

In this report, I will first describe this acceleration structure I implemented and display its results in terms of performance. In part 3 and 4, I will explain how I handled the collisions of a moving character with its environment and then address the user interaction part. Lastly, I will mention different ways of improvement that I can think of in part 5, some of them crucial for performance but not necessarily linked with computer animation.

2 Collision Handling Acceleration

Our of detecting collisions so far is highly inefficient: using the naive algorithm, we have to perform a **quadratic** number of tests between each pair of spheres to compare their distance with their radius.

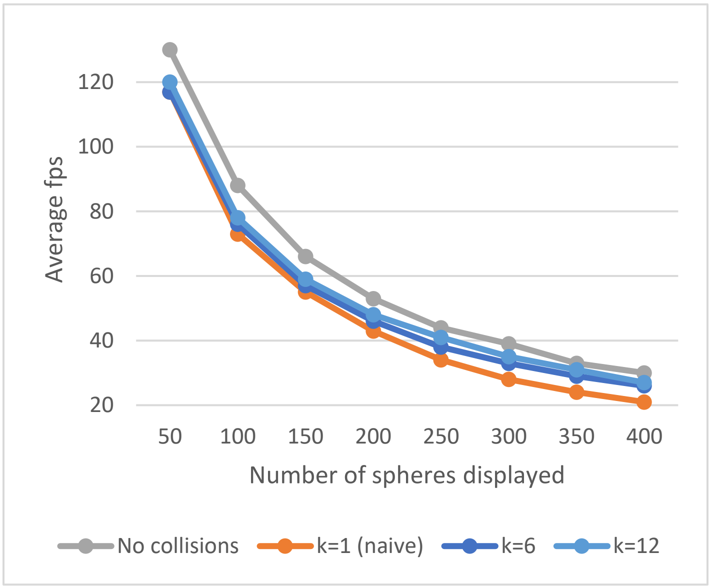
A simple method I chose to improve this is to perform a **regular spatial subdivision**, that is using a 3D grid to store the position of each sphere in discrete cells dividing the cubic space. That way, detecting collisions for a given sphere requires to check only what is contained in the same cell as well as in its neighbors, which greatly reduces the number of tests to perform (as the balls tend to be rather uniformly distributed, at least on the plane).

To implement this, I used a **grid3D** that serves as a hash table associating each coordinate (an int3) in the grid with the indices of the balls contained in the associated cell (each ball is now given an index when it is created + a reference to the cell it is currently in). A new *associateCell* method in the simulation code computes at each iteration the position of the sphere in the grid, then checks if it has changed from the last iteration (to avoid numerous unnecessary updates which I found out was costly), in which case it adds the index to the list stored at the cell coordinate in the grid3D (and changes the cell reference of the sphere).

Now, to detect collisions for a given sphere, we only have to loop over all the cells in its surroundings (27 cells maximum, but often less when a sphere is close to a border or a corner) and get in constant time the indices of the sphere belonging to those cells. By subdividing the space in sufficiently small cells (but still bigger than the radius of a ball), we therefore avoid a lot of unnecessary tests only at the expense of space memory (**$O(k^3+n)$** with *k* being the grid size and *n* the number of balls), but in our case this is much more acceptable than a computation in quadratic time.

2.2 Performance analysis

To better observe the gain in performance brought by this technique, let us compare the evolution of frame-per-seconds of the simulation in different configurations.



n	k=1	k=6	k=12	No col.
50	117	117	120	130
100	73	76	78	88
150	55	57	59	66
200	43	46	48	53
250	34	38	41	44
300	28	33	35	39
350	24	29	31	34
400	21	26	27	30

Figure 2. A comparison of the average frame-per-second obtained for a given number of spheres depending on the collision detection method (k is the grid size). It is important to note that the character model and its collisions haven't been added yet. Also, we can remark that using a grid of size 1 gives the same results as the naive version where we don't use any grid.

We can observe that using a grid of sufficient size has a **non-negligible impact on the framerate**, gaining progressively up to 6 or 7 fps when the number of spheres to handle is very high. Computation time is reduced by a large margin and becomes less and less impactful compared to rendering when n increases: indeed we can notice that, with k=12, the average fps becomes very close to what we could obtain if we are not handling collisions at all, while when not using a grid (k=1) the framerate continues to drop steadily. This will become even more noticeable once we add the marine into the pool and handle all of its collisions, which is the object of the next section.

3 Character Collisions

Using the code of lab 6 (skinning), we can now add our fully rigged character model into the ball pit. Of course, all the balls pass through it at first and we must find a way to make them collide with the body. My first, most basic attempt was to use a static cylinder surrounding the character, which was computationally efficient but had two major drawbacks: it obviously did not take into account the geometry of the body (no gap between the legs etc.) but would also fail to adapt its shape and velocities when the character is moving (with its legs going back and forth for example).

Using two cylinders for the legs could improve the results but still would not take into account deformation at the knee or the shape of the feet for example. Fortunately, our rigged character comes with a skeleton with **joints**, which are all closer to each other than the radius of a ball. My method is then to use the position of this joints as **center points for newly defined spheres** of diameter equal to the width of the body at this point. This way, the character is represented as a collection of spheres moving with its skeleton, and their collisions with the balls are handled almost the same way as any other.

However, one thing differs for these new spheres: their movements is constrained to that of the animation of the character, which means that their velocities are left unchanged by collisions. This is also tantamount to assigning them an infinite mass: this remains realistic as we can suppose that the balls in the pool are light enough compared to the weight of the marine to not influence his movement at all when he is moving. The formula to compute the new velocity of the ball after collision with the body becomes:

$$\boldsymbol{v}_{new} = \boldsymbol{v} + 2(\boldsymbol{v}_{body} - \boldsymbol{v}.\boldsymbol{n})\boldsymbol{n}$$

where $\mathbf{v}_{\text{body}}$ is the velocity of the sphere of the body colliding with the ball and $\mathbf{n}$ the unit vector from the center of the ball to the center of the body sphere.

The last thing left to do is to compute the body velocity at each joint. For this I added a new buffer to the skinning data structure to store it simply by subtracting the previous position to the current one at each iteration and dividing by Δt. For now, the body stays idle so it doesn't affect the simulation much, but this will serve for the next section. To detect body collisions for a given ball, we now iterate over the joints of the body only if it is contained in the same or a neighbouring cell : the grid subdividing the space has its use again! Let's compare the performance once more, this time we will observe the real benefits of the optimization.

As we can see on Figure 3. our **optimization really is vital when** combining the character with a large number of balls. Without it, the framerate can quickly drop below 10 fps or so, whereas it remains close to the best possible performance (the one where we don't handle collisions at all) with a large enough grid, staying higher than 24 fps. This is just what we need to turn this into a real-time interactive simulation, which we are going to do in the next section by adding user commands and character movements.

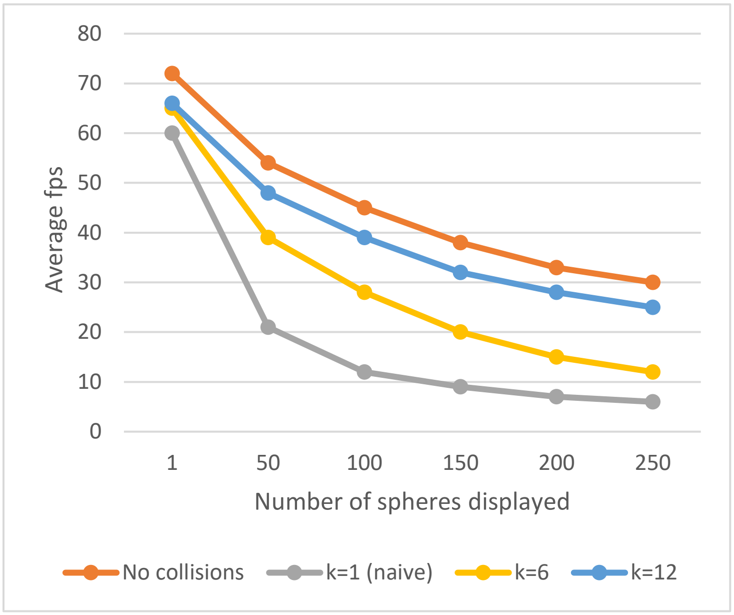


Figure 3. Comparison of the average fps obtained for a given number of spheres WITH the character on the screen depending on the collision detection method. The difference is even more noticeable this time as the body can add a lot of collision test.

n	k=1	k=6	k=12	No col.
1	60	65	66	72
50	21	39	48	54
100	12	28	39	45
150	9	20	32	38
200	7	15	28	33
250	6	12	25	30

4 Adding Some Interactions

A great thing about the marine we skinned in lab 6 is that he comes with a set of animations (idle, walking, running) that we can use to control it like a player character in a game. To do that, I modified the key callback function to translate the skeleton in global space when the *up* key is pressed, and make it rotate around itself when the *left* or *right* keys are (at first I used them to make the character move on the general x or z-axis but it was quickly becoming very confusing as soon as he wasn't facing north) and clamping its position inside the box in a *move* function that is called at each iteration.

Accompanying this change in global coordinates, I also reload the animation of the skeleton when a change in speed is detected (only the animation, initially I was reloading the whole skeleton and it took some time, provoking a 1 second lag each time there was a transition), which is performed almost instantaneously. For now, I chose the walking animation because it was still more "stable" than the faster running animation where feet could somehow pass through balls at times, I don't exactly know how

What's more, the camera *and* the gravity can be controlled with the mouse: I've assigned the latter to be the "down" vector of the camera, that way the user can move the balls around easily (can be disabled by a checkbox). The character is not affected by gravity however, else it would be a nightmare to control as one can imagine. There are many things that can still be worked on, and I will present some of them in the final section.

5 Possible Improvements

The first thing that jumps out when moving the marine around is that the transition between the two animations is very abrupt and unnatural, with legs teleporting between two states. It would be interesting to smooth it out by **interpolating between the two positions** for the first second of the transition, but maybe it is a bit more complicated than that if we have to take into account which phase in the walk cycle the character is in.

Collisions with the body seem to work very well, but maybe using balls to represent body parts isn't the optimal method because of some poorly oriented normal: switching to more adapted cylinders or bounding boxes like in some video games may be more accurate, although it would require some specific modeling adjusted for this precise mesh.

Handling a very large number of spheres also inevitably poses a lot of problems: using pair-wise collisions sometimes causes the balls to push themselves into the walls or prevent the propagation of a shock. This can be addressed by some global approaches seen at the end of the lecture about rigid spheres such as precomputing a contact graph or using a global constraint-based method. However there is also the matter of performance, which I tackled a lot in this project but as we saw with my different measures, collision detection isn't the only thing affecting it: even without collisions, the **rendering of numerous spheres alone slows down the simulation** significantly (dropping from 130 to 30 fps with 400 balls). The GPU calls are therefore thing to rework if we want to have a very smooth simulation while displaying hundreds of objects. Indeed, we have to make a lot of calls to the draw function for every sphere at each frame even though they are the same object with different coordinates. Instead, maybe we could use **OpenGL instancing** (https://learnopengl.com/Advanced-OpenGL/Instancing) and define a uniform array to store the offsets of all spheres and then perform only one call to *glDrawArrayInstanced.* I think that it could work theoretically, although I did not prioritize its implementation for this project it as it was more linked to the rendering part and not so much related to animation anymore. But it could be fun to see our marine drowned in a sea of colored plastic balls if it works !