DH2323 Project: UP!

Nicolas Zimmermann
960713-T092
nzim@kth.se

Virgile Hernicot
970923-T055
hernicot@kth.se

Zak Cook
961218-T090
zakc@kth.se

Abstract

UP! project intends to model the behavior of helium balloons in an enclosed space. It features 3D modeling of a scene and balloons, collision detection, user interactions and its own physics engine. The outcome is a realistic simulation with natural behavior, but with minor problems including balloon fusion, and finally lots of room for improvement.

1. Introduction

Having seen various topics throughout the course, we wanted to model a scene that brought together physical, mathematical and algorithmic concepts. After brainstorming, we thought to model a simple scene, a bunch of helium balloons being released and rising to a ceiling. The most important concept for our project was collision detection: how we will model colliders and which class hierarchy to use to maximize efficiency. Following collision detection is doing an ad-hoc approximated physics engine for our scene. And of course, we needed 3D models for our balloons and scene components.

2. Specification

Tools Our main tool is Processing[1], which is perfectly suited for our project. We wanted to implement collision detection from the bottom up, while not having to worry about rendering and shading, and we have had previous experience with the software. Although downsides are a messy project structure, primitive IDE and an outdated Java version, we believed we could pull off our scene. Then to model our balloons, we made use of a 3D Blender asset that we then imported into Processing. We didn't shape our 3d models from scratch, but tweaked some existing ones to our liking (size, orientation, origin at gravity center and texture parameters).

Scene We wanted to model a scene of balloons rising and reaching a ceiling. We wanted these balloons to collide, and when they collided, induce some sort of rotational velocity. Our scene therefore consists of multiple balloons starting at the bottom of a square room, rising to reach a pyramidal ceiling, and stabilizing there.

Collision detection Since our aim is to implement non-trivial collision detection, we wanted to do a form of bounding volume trees [2]. Having our 3D balloon model, we would place by hand a hierarchy of bounding spheres on the model, and have our collision detection algorithm run on those. We also want an appropriate class structure that fits the hierarchy and the algorithm. Finally, since our collision detection is *discrete*, we wanted to include *backtracking* [3] in some way.

Physics Looking up on how to integrate physics into our scene, we quickly realized that we would have to do a lot of approximations: fluid dynamics, friction, balloon deformation and more was simply out of our reach. We decided that we would implement basic gravity and bouncing, and from there develop additional physical effects to make our scene look more real, using angular motion.

3. 3D modeling and interactions

3.1 Room structure

The scene consists of a closed room where helium balloons are set free. The room is composed of four vertical walls and a pyramidal ceiling to make the collision with the balloons more interesting.

Following the spirit of the labs of track 1, the elementary object is a triangle described by 3 vertices. It allows us to generate all the needed geometry and draw them easily using the `vertex` method of Processing's `PShape` class. This model is also compatible for corner, edge and plane collision detection.

[1] https://processing.org/

A wall is composed of two triangles oriented vertically and a roof tile is only one triangle. The complete structure can be seen on Figure 1.

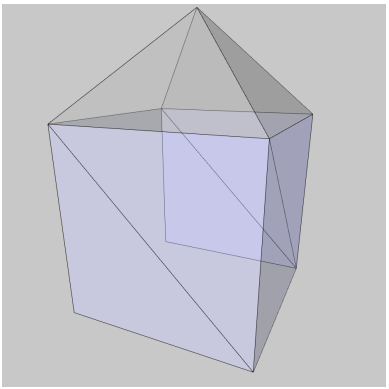

Figure 1. Structure of scene

3.2 Balloon object

Visuals Visually, the center of attention in the scene is the balloons, hence they need to have a realistic shape, size and behavior. To stay in the scope of the course, we started from an existing 3D model available here. Using Blender, the ground and the attached rope were removed. We also fixed the orientation so it would sit vertically in Processing and set the origin to the gravity center to draw it accurately. In order to have balloons of various colors, a .mtl file was created for each hue, setting the appropriate diffuse color (Kd). [2]

Geometric representation To correctly evaluate a balloon in the 3D space, the PShape was inappropriate to use in Processing alone. Making the use of our class hierarchy detailed in 4.1, several scales of sphereColliders are attached to each balloon. The first one is a sphere encompassing the whole object to have a rough estimate of its spacial situation. The second scale is here to match the non even proportions of the balloon: a thin base and a bulkier head. Finally, three extra spheres fill the gap in between to complete the spacial coverage of the balloon. The structure is depicted in Figure 2.

3.3 User interactions

Camera rotation For the user to better evaluate the scene, a navigation system using the arrow keys has been created. It consists of a spherical movement around the origin of the scene at a constant given distance. The keyboard input listeners increment two rotation angles and the camera is rotated by updating the `eyeX`, `eyeY` and `eyeZ` parameters of the `camera` function. These values are computed using the spherical to Cartesian coordinates conversion.

Adding balloons We added a system to easily add more balloons to the scene. By pressing "CTRL" key, the game will pause and the camera view will change to see the scene from the bottom. Then we can click to add balloons on the scene.

[2] https://en.wikipedia.org/wiki/Wavefront_.obj_file#Basic_materials

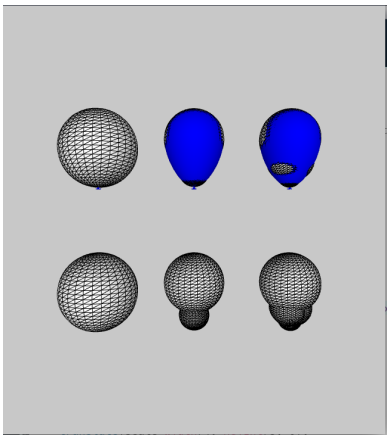

Figure 2. Balloon spheres representation

4. Collision detection

4.1 Class structure

As mentioned in our specification, our main focus in this project was collision detection. We started by implementing an appropriate Java class structure, the core class being `SphereCollider`.

```
public class SphereCollider {
  private Object3D mParent;
  private PVector mPosition;
  private float mRadius;
  private Set<SphereCollider> mChildren;
  private Set<Triangle> mTriangles;
}
```

- `mParent` corresponds to the object that this collider is attached to,
- `mPosition` and `mRadius` describe the relative position and the radius of the collider,
- `mChildren` represents the set of child colliders of the instance,
- `mTriangles` represents the triangles of the physical object the collider contains, in the case that the instance has no child colliders.

4.2 Algorithm

We go through every Object3D and check for collisions with every other Object3D 1-to-1, whenever a collision occurs, we backtrack both objects and change their velocities.

We record each object that had such collision and go through them once more in the event the backtracking made them collide with another Object. The operation has to be done a certain amount of times to avoid the object "melting" together.

Each Object3D contains one basis SphereCollider that defines the boundaries of the object. This SphereCollider is made of a hierarchy of other SphereColliders and having as children, either Triangles or other SphereColliders. We decided to build the balloons entirely out of SphereColliders without any Triangle for collisions, to diminish the amount of computations and because SphereColliders gave good results. The walls are made out of triangles at their roots, since SphereColliders would not give a good representation of a flat surface. To check if two objects intersect, we first check the SphereCollider at the top of the hierarchy and then while it collides, we go down in the hierarchy until we reach the root.

This implementation allows us to avoid dealing with collisions between triangles since every movable object is entirely made of SphereColliders, hence we have to deal with two sorts of collisions; between two SphereColliders and between one SphereCollider and a Triangle.

The detection of a collision between two SphereColliders is very simple, we just have to check that the distance between the two centers is less than the sum of the radius. The Triangles revealed to be much harder than we initially thought; there are 3 cases to verify to check the collision between a SphereCollider and a Triangle, with the inside of the triangle, its edges and its vertices[1].

- Surface: we project the center of the SphereCollider on the plane of the triangle and then check that the projection is with the 3 vertices of the triangle. Checking that the projection is within the triangle can be made by verifying that the area of the triangle is the same as the sum of the three triangles made by the projected point and each couple of vertices.
- Edges: we project the center of the SphereCollider on the the line given by each couple of vertices and check that it is between those vertices.
- Vertices: we check the distance between the vertex and the center of the SphereCollider to be less than the radius of the sphere. It also required to deal with them in the right order to avoid bugs like a sphere hurting an edge before the surface of another triangle.

So we first deal with every surface of the group of triangles and then only with their edges and vertices.

Each of these cases induced different treatment of resulting velocity and backtracking that will be explained in the next sections.

4.3 Backtracking solutions

Vector solution A solution we came up with was to use vector mathematics to find the displacement. Consider the situation in Figure 3. The red and blue circles represent colliders at tick t_0 (dotted) and t_1. The blue circle has centers A_0 and A_1, and the red one has centers B_0 and B_1. Both displacements have to reduced by an equal proportion t, to achieve the collision point. We define the collision point to happen in when the blue circle is in corrected center A'_1, the blue one in corrected center B'_1. These two points have to be precisely apart by $r_A + r_B$. This leads to the system of equations 1.

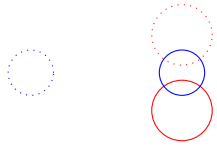

Figure 3. Situation for vector backtracking solution

$$\begin{cases} t\overrightarrow{A_0A_1} = \overrightarrow{A_0A'_1} \\ t\overrightarrow{B_0B_1} = \overrightarrow{B_0B'_1} \\ \|\overrightarrow{A'_1B'_1}\| = r_A + r_B \end{cases} \tag{1}$$

$$r_A + r_B = \|t\underbrace{(\overrightarrow{B_0B_1} - \overrightarrow{A_0A_1})}_{V_1} + \underbrace{\overrightarrow{A_0B_0}}_{V_2}\| \tag{2}$$

$$= \sqrt{t^2\|V_1\|^2 + \|V_2\|^2 + 2t\|V_1\|\|V_2\|\cos\theta_{V_1V_2}} \tag{3}$$

We then can solve this second degree equation to find two solutions for proportion t:

$$0 = t^2\|V_1\|^2 + (2t\|V_1\|\|V_2\|\cos\theta_{V_1V_2})^2 + (\|V_2\|^2 - (r_A + r_B)^2) \tag{4}$$

And then choose the smallest positive t, that satisfies $t < 1$. This seems to us like a correct solution, but implementing it in practice did not yield good results. It seemed that there were too many errors due to floating point multiplication, as we kept getting solutions that were in bounds but off by a small amount.

Binary search This is an easier but computationally more requesting solution, we simply try to find the best proportion t to rewind the movement of both objects until they are the closest to each other. We use classical binary search technique to that end.

Triangles Since the object containing the triangles is static, we simply have to backtrack the SphereCollider until its distance to the surface/edge/vertex (depending of what collision happened) is equal to its radius. This is a simple calculation required only some geometric manipulations.

We ended up using binary search for our backtracking, as it gave the best results.

5. Physics

Archimedes' principle The most interesting property of a helium balloon is its ability to rise up in the air. This is

due to Archimedes' principle and helium being less dense than air. The direct force applied to the balloon is called the buoyancy force, is opposed to gravity and has a magnitude of $B = \rho_f V g$, where V is the volume of the balloon, ρ_f the density of helium and g the gravity constant. Since our 3D environment doesn't come with real world units, the volume of the balloon isn't very meaningful and we had to estimate it, keeping the same order of magnitude.

Bounce velocity and angular motion In the event of a collision between two movable objects (which are only balloons) we had to simulate a bouncing effect, and an induced rotation on the objects. As physics quickly become too complicated in such a case, we had to radically approximate the calculations to come to a realistic simulation.

When the collision happens, both balloons have a certain velocity vector. By modeling the balloons as dumbbells with uneven masses, we could separate the transferred velocities into angular velocity and vector velocity by projecting on the perpendicular and parallel axis of the balloon's dumbbell. By tweaking the length and masses of the dumbbells as well as the bouncing coefficient, we managed to achieve realistic collisions.

In the case that the collision is between a balloon and a wall, the same sort of separation is applied, but only upon the balloon, using its own velocity.

Air friction After a collision, the balloons will have gained angular motion. But how can we slow down the balloon to realign it along the vertical axis? By using the rotational vector of the balloon, we add a velocity multiplied by factor of the sine of the current balloon rotation to its angular velocity:

- when the balloon is horizontal, it therefore has an angle of $90°$ with the vertical axis, meaning that we will add a larger amount of angular velocity to realign
- when the balloon is vertical, it therefore has an angle of $0°$ with the vertical axis, meaning that we will add no angular velocity

This solution is quite a hack, because when the balloon is vertical but upside down, it will have the same effect as if it were not.

6. Results

Through trial and error, we finally have arrived at a solution that looks close to a real simulation. Screenshots of a simulation in progress and stabilized is depicted in Figure 4.

During the simulation, our user interaction is smooth: moving around the camera and placing new balloons is fluid. Balloon rotations, being quite tricky to implement is also a positive aspect of our simulation. And most importantly, our collision detection works: recursion through the colliders works well and collisions are never missed.

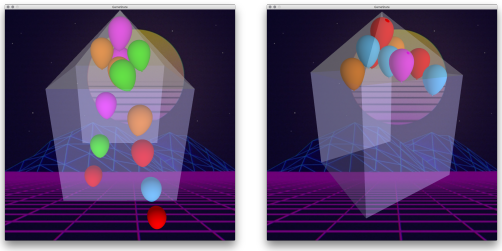

Figure 4. Scene simulation, in progress and stabilized

Although, a few things are missing. Collisions always cause a bounce, there is never a sliding effect, which could often be the case in collisions with walls. Also, sometimes balloon conflicts are not resolved, due to not using enough iterations to solve the collisions. Finally, things get slow after around 20 balloons, there is most likely room for improvement.

Finally, here are two video demos of our simulation: Balloons and Spheres.

7. Future work

Further improvements to our project would include string simulation that we would attach to the balloons, and much better textures. Perhaps a proper ray tracer that does light bounces and renders the balloons surfaces shiny. Another feature we thought about implementing when we first started working on our project was mesh deformation, but that was when we were using Unity and not Processing.

And of course, there is room for improvement with collision detection. Dividing the colliders among areas of the 3D space would definitely speed up things up. Lastly, although the physics looks realistic, we could be using more accurate computations.

References

[1] K. Fauerby. Improved collision detection and response. pages 11–17, July 2003. URL http://www.peroxide.dk/papers/collision/collision.pdf.

[2] J. T. Klosowski, M. Held, J. S. B. Mitchell, H. Sowizral, and K. Zikan. Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):21–36, Jan 1998. ISSN 1077-2626. doi: 10.1109/2945.675649.

[3] F. Yang. Lecture notes: Collision detection in computer games, April 2018.