# Resolution of 2D Rigid Body Collisions

Philip Burridge
Dan Camarda
Alexander Cederblad
Rasmus Haapaoja
Fredrik Johnson

March 13, 2014

## Abstract

This report will cover the process of creating a physics simulation of collision impulses and their resulting frictional impulses for basic geometries. This specific simulation has been chosen due to the large use of physics simulators in today's technology. Collisions between basic geometries requires the understanding of forces acting together (gravity and friction being two examples), geometries' physical properties and their resulting interactions with one another. Due to the mathematical limitations of computers (inability to calculate continuous integration), two seperate numerical methods for integration will be covered as integration is essential for these kinds of simulations. As this simulator has been implemented in *C++* and *OpenGL*, the pseudo-code for the physics implementation has also been covered.

# Contents

# Chapter 1

# Introduction

For this project we have chosen to simulate a general system involving collision impulse between rigid bodies in two and three dimensions. The implementation of the problem of resolving collisions includes calculating collisions between these bodies. Because of this and the fact that the focus of this project is the physical system and the numerical methods of simulation, the simplest geometries were chosen to narrow the focus. What are the simplest geometries for collision detection, you ask? Circles and spheres are the answer to your question. Since they can only collide at *one* point with other convex geometries, they are the best choice to keep the focus on the simulation. Figure 1.1 shows a snapshot of this kind of simulation with circles.

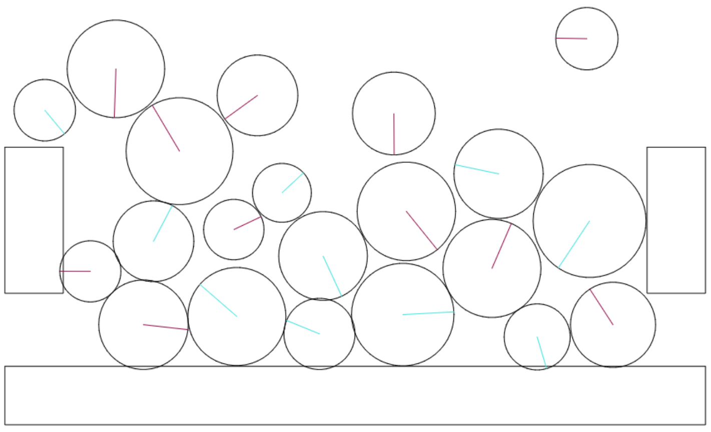

Figure 1.1: A snapshop of a simulation of circles interacting with each other. The line from the origin indicates the rotation and orientation.

# Chapter 2

# Physics

## 2.1 Gravitational influence

In order for our system to start moving, *Newton's second law* is implemented 2.1. This enables the system to be affected by gravity $\mathbf{F}$.

$$\mathbf{F} = ma \Leftrightarrow a = \frac{\mathbf{F}}{m} \tag{2.1}$$

Now that the objects have a linear velocity, the simulation can start checking for collisions between objects.

## 2.2 Resolving an impulse

To explain how to resolve an impulse[1], let us first think about two objects, A and B, colliding with each other, both with their own separate velocities at the collision point. We need to know the *normal* of the collision and the *collision point.* Since this is a computer simulation there will be a *penetration* between the colliding object, this is also helpful to register in a collision. Figure 2.1 displays a collision between two circles including this information.

Assume that we know the normal, $n$, of the collision. The relative velocity $\mathbf{v}_{AB}$ can be calculated as shown in (2.2), where $\mathbf{v}_{ACP}$ and $\mathbf{v}_{BCP}$ are the two objects' velocities at the collision point.

$$\mathbf{v}_{AB} \cdot \mathbf{n} = (\mathbf{v}_{ACP} - \mathbf{v}_{BCP}) \cdot \mathbf{n} \tag{2.2}$$

During a collision, an instant force i.e. an impulse will act on both objects, knocking them apart from each other. This change in a object's relative velocity $\mathbf{v}'_{AB}$, as explained in (2.3), tells us that the relative velocity will be inverted proportionally to the *coefficient of restitution* $e$. The restitution

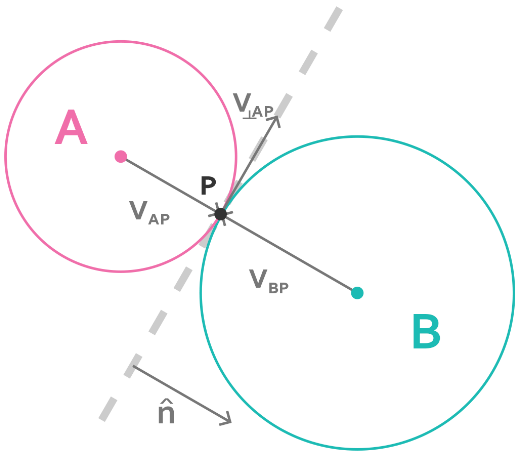

Figure 2.1: A collision including the normal of the collision and other quantities related to the collision. The collision normal is chosen from A to B by convention.

value will determine how much energy is lost during a collision. When $e$ is 1, no energy is lost i.e. a perfect rubber ball and when it is 0, it is a non-elastic collision i.e. a lump of clay hitting the floor.

$$\mathbf{v}'_{AB} \cdot \mathbf{n} = -e\mathbf{v}_{AB} \cdot \mathbf{n} \tag{2.3}$$

To calculate the unknown impulse $j$, we first only take into account the objects' velocities. Equations (2.4) and (2.5) show the change in velocity during a collision, by convention body A will add a positive velocity in the normal direction and body B will add a negative velocity.

$$\mathbf{v}'_A = \mathbf{v}_A + \frac{j}{M_A}\mathbf{n} \tag{2.4}$$

$$\mathbf{v}'_B = \mathbf{v}_B - \frac{j}{M_B}\mathbf{n} \tag{2.5}$$

After some rearranging, we can derive a final equation for the impulse coefficient in (2.5).

$$j = \frac{-(1+e)\mathbf{v}_{AB} \cdot \mathbf{n}}{\mathbf{n} \cdot \mathbf{n}(\frac{1}{M_A} + \frac{1}{M_A})} \tag{2.6}$$

As you might have guessed by now, this equation only helps in resolving the *linear* velocities of the rigid bodies. The following steps explain how to also account for the *angular* velocities during a collision. The angular velocity is obtained as shown in (2.7), where the impulse is positive for the object A and negative for B, as in the case of the velocities in (2.4) and (2.5). Vector $r_{AP\perp}$ and $r_{BP\perp}$ are perpendicular to the collision normal. $I$ is the moment of inertia of the object, determined by its geometry [2].

$$\omega'_A = \omega_A + \frac{\mathbf{r}_{AP\perp} \cdot j\mathbf{n}}{I_A} \tag{2.7}$$

To calculate the total impulse, the angular velocity is also taken into account and in the same approach as previously done with the linear velocity, the result can be derived as in (2.8). $M$ is the mass of the objects.

$$j = \frac{-(1+e)\mathbf{v}_{AB} \cdot \mathbf{n}}{\mathbf{n} \cdot \mathbf{n}(\frac{1}{M_A} + \frac{1}{M_B}) + \frac{(\mathbf{r}_{AP\perp} \cdot \mathbf{n})^2}{I_A} + \frac{(\mathbf{r}_{BP\perp} \cdot \mathbf{n})^2}{I_B}} \tag{2.8}$$

Now that the impulse coefficient is known (2.4) and (2.5) can be implemented in the simulation. The scalar is multiplied with the normalized collision vector in both equations. This is the basic simulation; when a collision occurs, solve for the properties of the collision and then apply the impulse to the objects' velocities and angular velocities.

# Chapter 3

# Implementation

## 3.1 Collision detection

To be able to resolve a collision impulse, an actual collision must first be registered. As already mentioned, there is some information that we need to know about a collision to be able to resolve the collision impulse: the *normal* of the collision, the *collision point* and the *penetration*, as previously illustrated in 2.1.

The collision detection will be different for each geometry. Circles are the simplest ones, rectangles can be handled in a few ways but the most common is the Axis-Aligned Bounding Box (AABB) method.

### 3.1.1 Circle to Circle

This is the simplest of all collisions to calculate. Since by convention we need to find a collision normal from object A to object B let us call the circles *circle A* and *circle B*. To calculate the normal, the origin position of B is subtracted by the origin position A (3.1). It is good practice to normalize the normal vector to ensure a proper collision impulse.

$$\mathbf{n} = \mathbf{p}_B - \mathbf{p}_A \tag{3.1}$$

The collision position is the point on the circle's edge in the direction of the normal. That is the position of origin of circle A added with the normal vector with the same length as the radius of circle A.

### 3.1.2 Circle to Rectangle

To find a collision between a circle and a rectangle the closest point to collision *on the rectangle* needs to be calculated. This is done by clamping the circle's position to the rectangle edge i.e. the bounding box (3.2).

$$p_{\text{closest}} = \text{clamp}(p_{\text{circle}}, p_{\text{rectangle}}) \tag{3.2}$$

The normal of the collision will then be the vector from this *closest* position and the origin of the circle if by convention we have chosen the circle as body A and the rectangle as B. The closest collision point is illustrated in 3.1.

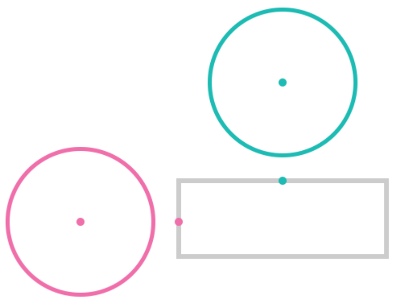

Figure 3.1: Illustrating the closest point on a rectangle during a collision detection between a circle and a rectangle. Note that this is *not* a collision.

## 3.2 Correcting penetration position

As mentioned in the previous section there will be a slight penetration between the objects because of the discrete step in time between each calculation. This is not a serious problem since most computers can run the simulation at a high framerate. However if the objects travel at a high enough speed they *will* penetrate each other as illustrated in 3.2, they might even skip collision if they travel past each other during a time step.

This effect can be handled in a few different ways. One common way is to model the penetration as a spring and separate the objects with respect to a spring coefficient and the objects' masses. In our implementation we have done a pseudo-spring solution to the problem. This effect is not clearly visible by simply looking at the simulation. However, we have the ability to spawn objects partly penetrating another object which clearly shows the effect. More on this in a later section (3.4) on the OpenGL implementation.

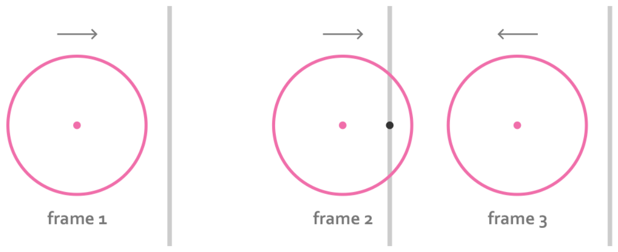

Figure 3.2: A collision during three time frames illustrating the penetration where the dark dot in the second frame represents the collision position.

## 3.3 Numerical integration

Since computers live in a world of zeroes and ones, we need a way to integrate numerically. For our simulation, first-order differential equations will be sufficient to approximate a solution. The most general first-order differential equation (3.3), where $\dot{x}$ denotes the first-order derivative and $f$, for example, denotes the velocity. Then the numerical integration $x$ would give the position.

$$\dot{x}(t) = f(t, x(t)) \tag{3.3}$$

### 3.3.1 Euler method

The most basic and widely used approach is the *Euler method*[1]. For this type of simulation, the *Euler method* is suitable for the majority of applications; however, another method will also be covered. The *Euler method* (3.4) works by adding the current state with its derivative multiplied by the step in time.

$$x_{n+1} = x_n + hf(t_n, y_n) \tag{3.4}$$

In this particular simulation, the method needs to be applied in two different situations. First of all to obtain the new velocities (3.5) of the objects for the next frame. Which is calculated by multiplying the acceleration with the time step and then add the product to the velocity in the previous frame.

$$v_{n+1} = v_n + ha_n \tag{3.5}$$

Since objects, of course, needs to be able to spin and move around in different directions for the simulation to be somewhat realistic. The positions $p$ in (3.6) and orientations $\Omega$ in (3.7) also needs to be updated in the scene.

This is simply done by integrating the velocities $v$ and angular velocities $w$, respecivley, by multiplying with the time step.

$$p_{n+1} = p_n + hv_{n+1} \tag{3.6}$$

$$\Omega_{n+1} = \Omega_n + h\omega_{n+1} \tag{3.7}$$

Since this method is one of the more simplistic numerical methods, it is important to keep in mind that an error will compound with each iteration. Many times the Euler method is not good enough, the compound error increase as long as the simulation is running, as explained by [3].

### 3.3.2 Runge-Kutta method

For our second method, we chose Runge-Kutta fourth order, often referred to as the classic Runge-Kutta method [4]. This method calculates the approximation in four increments as shown in (3.8).

$$\begin{aligned} k_1 &= f(t_n, x_n) \\ k_2 &= f(t_n + \frac{h}{2}, x_n + \frac{h}{2}k_1) \\ k_3 &= f(t_n + \frac{h}{2}, x_n + \frac{h}{2}k_2) \\ k_4 &= f(t_n + h, x_n + hk_2) \end{aligned} \tag{3.8}$$

The step size, h, is arbitrarily chosen, too big or too small will accumulate a larger error to the approximation. Like the Euler method, each step uses the previous computation to calculate a new value. To determine the next value from the present value, the results from (3.8) are weighted. The two middle values are twice as important as $k_1$ and $k_4$ as shown in (3.9).

$$\begin{aligned} dtdx &= \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \\ dtdv &= \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \end{aligned} \tag{3.9}$$

The final value is the sum of the weighted values and the present value (3.10).

$$\begin{aligned} x_{n+1} &= x_n + dtdx \\ v_{n+1} &= v_n + dtdv \end{aligned} \tag{3.10}$$

In our simulation Runge-Kutta is applied for position, orientation, velocity and angular velocity. The improvement from Euler is clear when the bodies are piled up next to each other with zero velocity. With Euler, the balls continually move and never reach a steady state. The subject is futher discussed in the *Discussion* chapter.

## 3.4 OpenGL implementation

The simulation is implemented in *OpenGL* which is a computer graphics API supported by most graphics cards. Our implementation was written in *C++*. Listing 3.1 shows pseudo code for the implementation.

```
while simulation is running:
    find collisions;
    apply gravitational force;
    resolve collision impulse;
    integrate resolved velocities;
    update position;
    draw objects at new position;
```

Listing 3.1: Pseudo code for the simulation loop.

To make the user able to interact with the simulation during runtime, the user is able to spawn new circles with a random radius. This is done by left-clicking the mouse, which will create a new circle at the mouse position.

Some restrictions exist when implementing the geometries. Rectangles are, for example, not allowed to be affected by the physics, i.e they are only allowed to act as static objects. This only allows us to use them as floors and walls, not as actual falling bodies.

### 3.4.1 Running the simulation

Navigate to the folder named *release* and then the OS-specific folder of your choosing. Click the executable file named *run-simulation.** to run the simulation. The software is not tested on any Linux distribution. These instructions are also located in the *README.md* file.

# Chapter 4

# Discussion

While we are quite pleased with the results of our two dimensional physics simulation, there are some definite limitations in the work. Due to the fact that our simulation only accounts for two seperate types of geometries, there is much to be desired in that respect. While the results of the simulation do mimic reality with the collisions of circles to circles and circles to rectangles, a wider array of geometries would test the capabilities of our simulator even further. This increase in collision types would certainly give us a better understanding of whether our simulator is a reliable way to imitate reality.

Another glaring problem when running the simulation is that after the majority of the high-impulse collisions have occurred and the circles come to rest upon each other as well as on the rectangles, they continue to spin. Obviously, if these bodies have any type of frictional surface, they will at some point come to a rest and stop spinning. This most likely due to the fact that our friction is directly dependent on the impulse of the collisions. When two bodies are resting against one another, the collision impulse is very small (almost non-existant). This gives us a friction along the perpendicular tangent to the collision which is even less and thus, we have a frictional force which is nearly non-existant when two bodies are resting against one another. In retrospect, this is likely a case where a static friction should be implemented in addition to the dynamic friction for the two different cases.

# Bibliography

[1] Hecker C. Collision Response. Game Developer Magazine. 1997 March;http://chrishecker.com/images/e/e7/Gdmphys3.pdf
Part 3 of series in 4 parts. The series:
http://chrishecker.com/Rigid_Body_Dynamics.

[2] Nordling C, Österman J. Physics Handbook for Science and Engineering, eight edition. Studentlitteratur; 2006.

[3] Fiedler G. Basic Integration; 2006. Available from: http://gafferongames.com/game-physics/integration-basics/.

[4] Runge-Kutta method; 2012. Available from: http://www.encyclopediaofmath.org/index.php/Runge-Kutta_method.