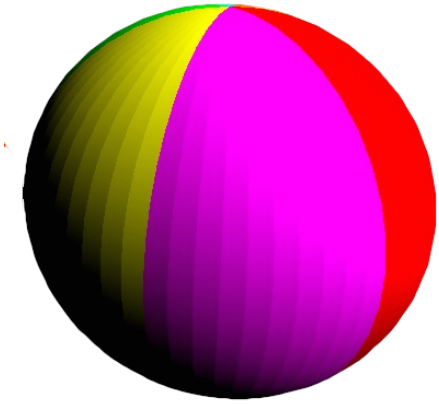
Beachball Physics



Fundamental game physics supplement to TSBK03 Advanced Game Programming

by Ingemar Ragnemalm 2023-2024

Introduction

With this text, my intention is to provide a compact written course material that includes the most vital parts of Sergiy's presentation [8] that are not covered by the course book [7], in order to provide a more complete material than lecture slides, while still more compact than the books that Sergyi based some of his material on.

Most of this text are essentially high school physics, but my focus will be on the parts that are relevant for games and real-time animations. Therefore, I have limited interest in the exact formula for, e.g., rotational movement but rather how to model the forces that cause that motion, preferrably in *discrete time*. The issue of discrete time is already covered in the course book, more specifically in the numerical integration chapter, but will appear here too.

It should be stressed that the discrete time case is central to game physics and the big difference from classic physics. Therefore, numerical integration is vital, as well as working with *time steps*, taking steps by Δt.

The text assumes that you have the prerequistes for a course like TSBK03, so that you are comfortable with basic algebra, including 3D space, 3D coordinates, vectors in 2D and 3D, dot and cross products, matrices and matrix multiplications, and graphics programming with OpenGL. See chapter 4 in Polygons Feel No Pain [6] for a brief review on the algebra.

The following topics will be covered:

1) Movement in discrete time

2) Fundamental laws

3) Projectiles

4) Collisions

5) Rotations

6) Friction

7) Springs

8) Aerodynamic drag

9) Sailing ships

10) Waves

Some of these are covered by the book, but here I will attempt to cover the fundamentals in more detail, with more examples.

The main sources that are, in varying degree, used for this text:

• Sergiy Valyukh's lecture slides. [8]

• "Physics for game programmers" by Grant Palmer [1]

• "Physics for game developers" by David M. Bourg [2]

• "Computer Animation" by Parent [3]

• "Game physics" by Eberly [4]

• Baraff & Witkins [5]

This supplement comes with a range of demos, all using, apart from my usual units, GLUGG for building models and SimpleGUI for controls.

The title, "Beachball Physics", refers to some of the demos, where beachballs were selected as

objects where many effects are relevant. (As a bonus, sailing is also close to beaches.)

I expect you to find sections 1 to 3 very basic, but the point here is not to leave things undefined. In number 4, things get more practially useful, closing the gap between my books [6] and [7] a bit. Then, I think, things will get much more interesting. Just getting that kinematic rotation is IMHO a very neat thing to do. You will see that many applications (especially the sailing boat) are mainly a matter of applying forces in the proper way, but that is, in my view, much of what game physics is about.

At this time, the parts relevant to the game physics part of the course are sections 1 to 8. Section 9 is currently not used and section 10 are extensions of the "other graphics topics".

WORK IN PROGRESS

This is a very early version of this supplement. The text as well as the demos are very new and not tested much. Errors are to be expected. Let me know if you can see any weaknesses, errors or omissions.

2024: Several errors and improvements fixed suggested by Andreas Sahlin! Thank you!

Table of Contents

1. Movement in discrete time

The most fundamental thing in game physics is movement, not least for particle systems, but also for other moving solid objects. Movement is nicely described as position being the integral of velocity and velocity the integral of acceleration

$$\mathbf{p}(t) = \int \mathbf{v}(t)dt$$

$$\mathbf{v}(t) = \int \mathbf{a}(t)dt$$

but from there we need to do that in discrete time and we get

$$\mathbf{p}(t) = p(t) + \mathbf{v}(t)\Delta t$$

$$\mathbf{v}(t) = v(t) + \mathbf{a}(t)\Delta t$$

where Δt is the time from evaluation step to evaluation step, which corresponds to the frame rate.

Here questions arise with better integration methods and what order these steps should be performed. In the TSBK03 course book [7] this is discussed in the chapter on numerical integration. In the following, this primarily means that the time step Δt should be included whenever we want the physics simulation to be reasonably accurate.

The Δt variable is usually the time beteen frames. However, you may want to limit the Δt when the actual time steps gets very large. Very large Δt causes big jumps that can both be disturbing for the player as well as it can cause problems in the physics simulations.

2. Fundamental laws

In this section, I will briefly review some of the most fundamental laws of physics. No surprises here, we are just setting the foundations.

Newton's three laws of motion

Newton's laws are fundamental to classical physics, which describes physics on the level we need, the way we percieve physics in real life. It describes forces, movement, rotations etc, just what we need.

Newton's three laws of motion are as follows:

Newton’s first law: the law of inertia.

If no forces act on an object, it will keep the same speed as it has.

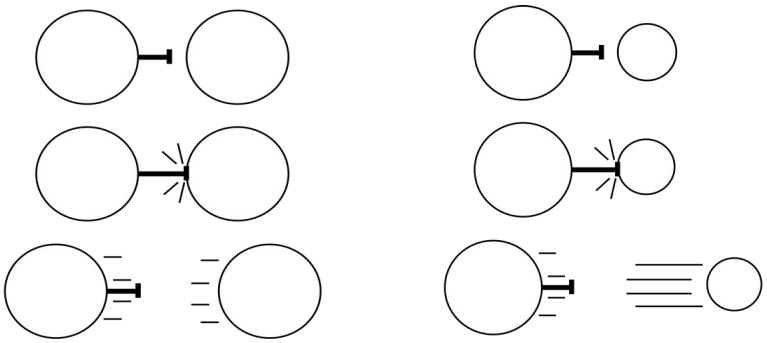
Newton’s second law: F = ma

A force on an object will cause an acceleration inversely proportional to its mass.

Newton’s third law: the law of action and reaction

If one object puts a force onto another, there will be a force onto the first object in the other direction.

The second law is fundamental and appears everywhere. The third law is perhaps the most interesting one, since it is a reminder on how we can apply forces on an isolated system, like a constraint system. As long as we balance the forces, we will not add unwanted movement to the system.

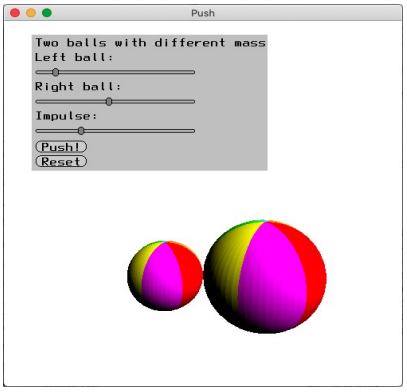


One object applies a force on the other and both move, but if one object is smaller (less mass) it will move faster according to F = ma.

However, in practice you will often consider some object stationary, which means that you give them infinite mass. In those cases, there will be no effect of the force on the stationary object.

The effect of these opposing forces also serve as an example of the second law, since they have different mass and therefore will move at different speed from the same force.

This leads to our first demo (and our first beachballs), the demo "push". Like most of the demos here, it is based on GLUGG (OpenGL Utilities for Geometry Generation) as well as SimpleGUI, and is entirely procedural with no external models or other files. As such, I think it both demonstrates Newton's third law but also the general setup for the demos. I hope you will find them straight-forward and informative.



"Push", two beachballs

In this demo, you can adjust the mass of each ball as well as the impulse, and get the effect described above. According to the second law, the speed is affected by the impulse by division by the mass, and the same impulse is applied to each ball.

This demo, as well as the arrow demo below, are pretty trivial, but they are just the beginning. I promise more interesting demos following these.

The conservation laws

There are also the four fundamental constants, the *conservation laws*, in such a system:

$$\sum m_i = \text{constant}$$

$$\sum E_i = \text{constant}$$

$$\sum P_i = \text{constant}$$

$$\sum L_i = \text{constant}$$

In words, the mass, the energy, the momentum (P) and angular momentum (L) all stay the same for a closed system. The last two are a consequence of the sums of force and torque above. We may lose kinetic energy, but only by having it converted to heat or potential energy.

3. Projectiles

Although we often have projectiles in games, I must say that our interest in the more detailed simulations is limited. Most of the time, we just want the projectile to fly straight in very high speed, i.e. a bullet, and the interesting thing is the collision. Not even gravity is significant for this case. Gravity is more interesting for relatively slow moving projectiles like arrows, and for such cases, wind and aerodynamic drag may have some interest. This makes Palnatoke and the apple a good example. (If you didn't know, the story about Wilhelm Tell is a later version of the same story. The viking version is older.)

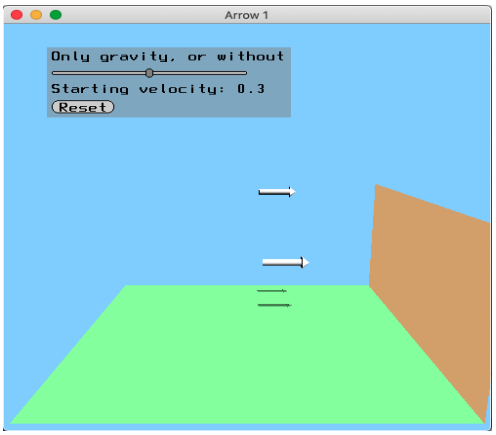
Spin effects is often of limited interest, but there are exceptions. Golf simulations (Palmer's favorite scenario, it seems [1]) is not my favorite case since the balls are small and hard, so wind and spin effects are relatively small. It is important for a realistic golf simulation, but not exciting as separate demos. However, I can imagine scenarios where you can throw something like a beachball, which has very strong sensitivity to wind, so let us not rule it out completely. The beachballs also tend to have distinct patterns which are good for demonstrating rotations. Thus, let us use that beach ball as model and we will return to wind and spin in a later section.

Thus, in the following, we have some different models for the examples, first arrows and, for several cases, beach balls.

Arrows - only gravity

Gravity only is very simple. It is just a downwards force F = gm. So if this is the only concern, Palnatoke need to aim higher in order not to hit his son. (Fortunately, Palnatoke was a good shot and knew this, but... if you know the story, he pulled out *two* arrows. But the full story is beside the point.)

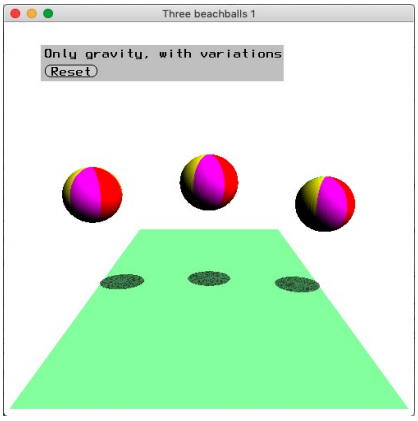
The "Arrow" demo shows this, although the result is trivial: With gravity, the arrow moves down. We can make this more formal by calculating the force and applying that, but in the end, the mass will go in and out and we still simply accelerate by g. Air has very little effect on arrows. However, things get more interesting when we throw beach balls with air drag and spin.



Arrow

Beachballs with gravity and simple collisions with the floor

The integration and simple collision handling is very sensitive to operation order. Our demo "Three beachballs 1" has only gravity, but each of three beach balls handles the acceleration and collision slightly differently. The collision detection and handling here are very primitive, just a matter of testing height and switching direction. We will return to that.



Three beachballs 1

The leftmost and rightmost balls add gravity to velocity, and then velocity to position, while the middle one uses the reverse order.

The leftmost and middle balls will, upon collision with the ground, reverse direction and move the ball up to 1 (the radius), while the rightmost ball only reverses direction.

The result differs substantially. The leftmost ball is stable while the middle gains energy and the rightmost loses energy and will even sink through the ground after a while.

However, the stability of the leftmost is a coincidence. "Three beachballs 1a" is a variation of this. Here, the rightmost ball has the nice property of moving the ball up as much as it went down into the floor. This makes it much better, but if you play around with the numbers (for Beachball 1a it is the time step) you will see that it is fairly stable but not perfect.

This stands as an example of how small things affect even the simplest situation when we deal with physics.

4. Collisions

Detecting and handling collisions are truly vital problems in games, and has therefore quite a bit of attention in the course book [7]. There, we deal with a complete physical model with torque, rotations, impulses and inertia. This is fine, but rather complex. In addition, some simpler cases (primarily collisions between objects of same mass) are described in volume 1, Polygons Feel No Pain [6]. This is very simplistic and I wish to fill in the gap a bit.

Here, we will discuss the fundamentals in some more detail, focusing on the 2D case for simplicity. Hopefully, this can be a useful start for understanding the 3D case better.

Let us start with momentum, the conservation of momentum. For two objects not affected by external forces, the sum of their momentums are constant before and after a collision:

$$v_1 m_1 + v_2 m_2 = \text{constant}$$

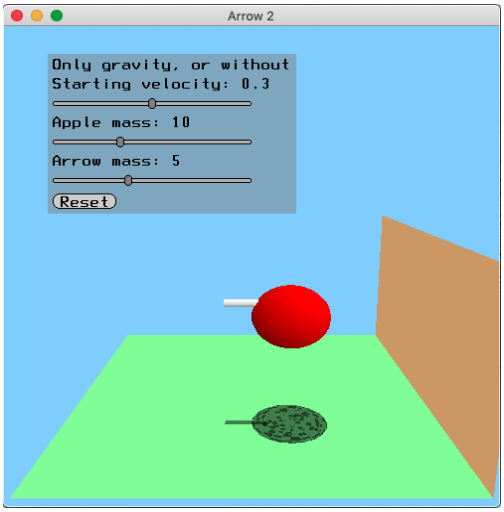
We can use this formula to solve collision problems.

Plastic collisions

Palnatoke shooting an apple from the head of his son is a classic scene that we can model as an inelastic (plastic) collision. We know that the initial speed of the arrow is v1, while the apple is still. After the collision, they both have the same speed v2. The arrow has mass m1 and the apple m2. This is a straight example of the conservation of momentum. The solution is as follows:

$$v_1 m_1 = v_2(m_1+m_2) \Rightarrow v_2 = v_1 m_1/(m_1+m_2)$$

This is demonstated in the demo "Arrow 2", where the arrow hits an apple (red spheroid) and the velocity changes accordingly.



Arrow 2, plastic collision

Like with Arrow 1, the result is not remarkable, which is not strange since this is the easy case that was trivial to derive. The point is that the change of velocity is based on the masses, that is the formula above.

If you have two objects that are both moving, with intial speeds v_{1b} and v_{2b}, you get the more general solution

$$v_{1b} m_1 + v_{2b} m_2 = v_a m_1 + v_a m_2 = v_a(m_1 + m_2)$$

and we have a solution as

$$v_a = (v_{1b} m_1 + v_{2b} m_2)/(m_1 + m_2)$$

Thus, plastic collisions are simple to resolve. For elastic collisions, things are more hairy.

Elastic collisions

For elastic collisions, we add conservation of kinetic energy, the kinetic energy is constant

$$v_1^2 m_1 + v_2^2 m_2 = \text{constant}$$

Let us take two billiard balls as example, idealized to giving totally elastic collisions. We only consider one-dimensional movement. As above we let them have the masses m_1 and m_2 and the velocities before collision v_{1b} and v_{2b}. After collision, they will get the velocities v_{1a} and v_{2a}.

$$v_{1b}m_1 + v_{2b}m_2 = v_{1a}m_1 + v_{2a}m_2$$

$$v_{1b}^2 m_1 + v_{2b}^2 m_2 = v_{1a}^2 m_1 + v_{2a}^2 m_2$$

These formulas has one trivial solution, no collision, so $v_{1b} = v_{1a}$ and $v_{2b} = v_{2a}$. The other solution is found by realizing that

$$v_{1b}^2 m_1 + v_{2b}^2 m_2 = v_{1a}^2 m_1 + v_{2a}^2 m_2$$

$$\Rightarrow$$

$$(v_{1b}^2 - v_{1a}^2)m_1 = (v_{2a}^2 - v_{2b}^2)m_2 = (v_{1b} - v_{1a})(v_{1b} + v_{1a})m_1 = (v_{2a} - v_{2b})(v_{2a} + v_{2b})m_2$$

We now rephrase the first equation to

$$m_1(v_{1b} - v_{1a}) = (v_{2a} - v_{2b})m_2$$

and combining these two we get

$$v_{1b} + v_{1a} = v_{2a} + v_{2b} \Rightarrow v_{1a} = v_{2a} - v_{1b} + v_{2b}$$

or

$$v_{1b} + v_{1a} = v_{2a} + v_{2b} \Rightarrow v_{2a} = v_{1a} - v_{2b} + v_{1b}$$

which makes the whole thing almost look too simple! From there, it is straight-forward to eliminate v_{1a} and get v_{2a}, and then get v_{1a} in a similar way, and it will solve as:

using

$$v_{1b}m_1 + v_{2b}m_2 = v_{1a}m_1 + v_{2a}m_2$$

eliminate either from above and get

$$(v_{2a} - v_{1a} + v_{2b})m_1 + v_{2b}m_2 = v_{1a}m_1 + v_{2a}m_2$$

$$v_{1b}m_1 + v_{2b}m_2 = v_{1a}m_1 + (v_{1a} - v_{2b} + v_{1b})m_2$$

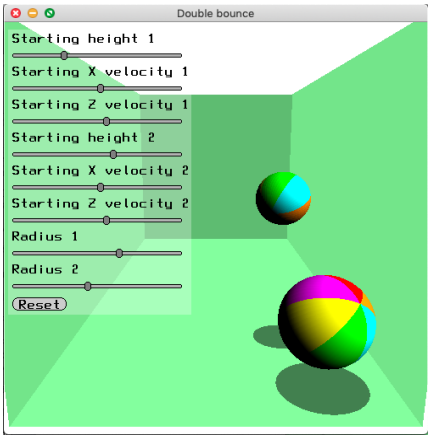
$$\Rightarrow$$

$$v_{1a} = (v_{1b}(m_1 - m_2) + 2v_{2b}m_2)/(m_1 + m_2)$$

$$v_{2a} = (2v_{1b}m_1 + v_{2b}(m_2 - m_1))/(m_1 + m_2)$$

This derivation was briefly hinted in Polygons Feel No Pain [6] but too brief to understand. My fault, sorry.

This is demonstrated in the demo "Double bounce", where two beachballs collide, and I am computing the collision response using the formulas above. The mass is the square of the radius, which is correct for a beach ball.



Double bounce

In the demo, you will notice that the balls bounce off each other very well, but they do not rotate. We will, however, handle this too. See the rotation chapter ("kinematic rotation" and "Double bounce 2") on how this is done.

Note that the case with equal mass, which I use in TSBK07 [6], where the two objects just exchange velocities, the formulas simplify greatly. The term with $(m_1 - m_2)$ disappears and the other term includes $2v_{2b}m_2/(m_1 + m_2)$ (similar for the second) which only leaves v_{2b}. After this, it comes out just like I say there:

$$v_{1a} = v_{2b}$$

$$v_{2a} = v_{1b}$$

To make the solution complete, we add a *restitution* parameter ε, where 1 is elastic and 0 is plastic. We use this to blend between the plastic and elastic collision formulas. To do this, we just multiply the formulas above with ε for the elastic part and (1-ε) for the plastic part. This will give us the final formula (adapted from Palmer [1], page 163):

$$v_{1a} = (v_{1b}(m_1 - \varepsilon m_2) + (1+\varepsilon)v_{2b}m_2)/(m_1 + m_2)$$

$$v_{2a} = ((1+\varepsilon)v_{1b}m_1 + v_{2b}(m_2 - \varepsilon m_1))/(m_1 + m_2)$$

Try to set ε to 0 or 1 and you will see that the expressions end up as the elastic and plastic cases above!

5. Rotations

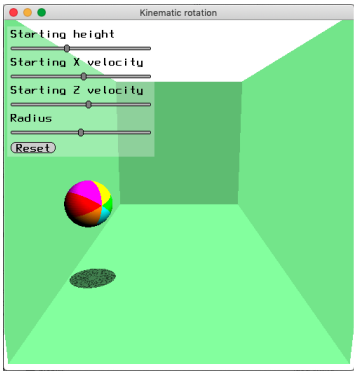
Making objects rotate is certainly important. This is covered in the book, so we will here only make a brief overview and see if we can dive into basic cases.

This will be done in several steps. First just plain rotation without physics. Then, rotations with physics in 2D. Finally, we generalize to 3D.

Kinematic rotations: Rotation directly from collision

Here comes the simplest case: Faking the rotation by assigning rotation by the direction of collision. No forces are involved, so this is pure kinematics.

If we want to make this more realistic, the ball should also lose speed in such a collision, but for the demo "kinematic rotation", I skipped that effect, and made a beachball bouncing around in a box with rotations.



Kinematic rotation, with rotations without physics

I must admit that the rotation, in this case, is added with no consideration of Δt, it is just arbitrarily decided in order to show how I can get a pretty convincing effect with little effort.

In 2D, this is pretty easy, a rotation around Z. You store the angle and the rotation speed as two scalars.

With full 3D rotations, it will be a bit more complicated. We model the current rotation with a matrix and another that represents the differences to that, the rotation speed. In the following code, we use the variable *rotmat* for the current rotation, and *diffrot* to model the rotation speed.

This is done by two steps:

Step 1: Every time the ball hits a surface, its rotation speed is set by the following code:

```
void SetRotation(vec3 v, vec3 n)
{
    vec3 r = cross(v, n);
    vec3 vn, vp;
    SplitVector(v, n, &vn, &vp);
    diffrot = ArbRotate(r, -Norm(vp)/radius);
}
```

The vector **n** is the normal of the plane and **v** is the velocity of the ball. We split the velocity vector in order to find the component that goes along the tangent of the surface (**vp**) and set the rotation speed from that.

The rotation is then given directly by the norm of **vp**. The ball has the circumference of radius*2π and the rotation speed will be the same as the norm of the velocity over the radius.

Step 2: Every frame, the rotation is updated by the rotation speed as:

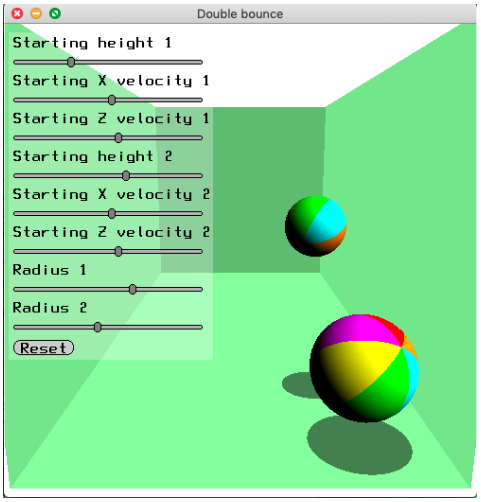
```
rotmat = Mult(diffrot, rotmat);
```

or in C++

```
rotmat = diffrot * rotmat;
```

Note that the rotation speed, the change in rotation, is multiplied to the left, that is after the current rotation.

I applied the same thing to the "Double bounce" demo above, which is now called "Double bounce 2"



Double bounce 2

There is little news in this demo, just improving the double bounce demo to something more convincing. But it does look so good. If you want to improve this but still work with kinematics only (or mostly), you should consider slowing down the balls based on the difference at the contact point, but then we quickly go closer to the kind of physics described in the course book [7].

Simplified case, 2D

For rotations, a vital concept is the *torque*, which means how a force adds angular momentum to an object. In 2D, the torque is a scalar. For a force vector F and a radius to the center of mass r, the resulting torque T is measured in Newton*meters, Nm . Then the torque is

$$T = rF$$

granted that the force is perpendicular to the direction towards the center of mass. If it is not, the sin part of the cross product will affect it and it will become

$$T = rF \sin(\varphi)$$

How easy it is to rotate a body, that is how much it will rotate given a certain torque, is given by the *moment of inertia*, J. In 2D, this is also a scalar, J. T will add to the angular momentum L. Then the sequence comes out like this:

$$L = L + T\Delta t$$

$$\omega = L/J$$

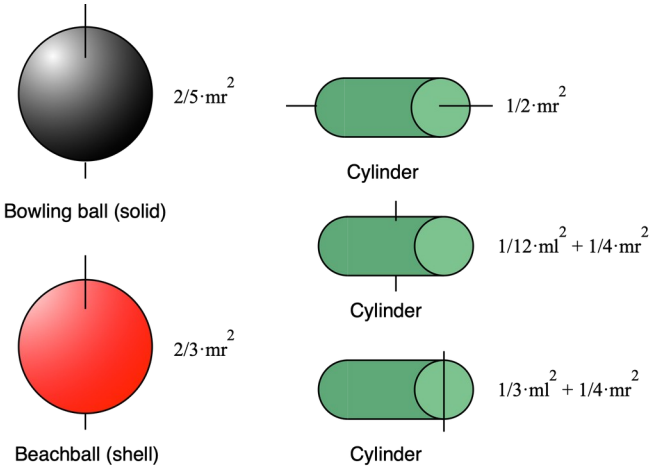
$$R = R + \omega$$

Although the moment of inertia is a scalar in 2D, it depends on what point the object is rotating around. Palmer [1] lists some shapes and I added the last one:

• Solid sphere rotating around the center: $2/5 \cdot mr^2$
• Spherical shell of radius r rotating around center: $2/3 \cdot mr^2$

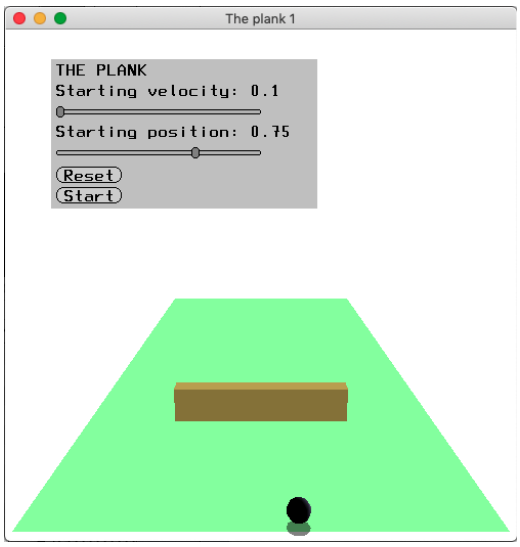
• Cylinder of radius r and length l, rotating around an axis along the cylinder, centered: $1/2 \cdot mr^2$
• Same cylinder, rotating along an axis across the middle of the cylinder: $1/12 \cdot ml^2 + 1/4 \cdot mr^2$
• Same cylinder, rotating along an axis across the end of the cylinder: $1/3 \cdot ml^2 + 1/4 \cdot mr^2$

These are illustrated in the following figure:



Inertia of some selected shapes

We will start with a special case, rotating a plank. This is similar to the case of rotating a rod, a thin cylinder, with axis at the end. We assume that l >>r so the inertia is $1/12 \cdot ml^2$.



The plank

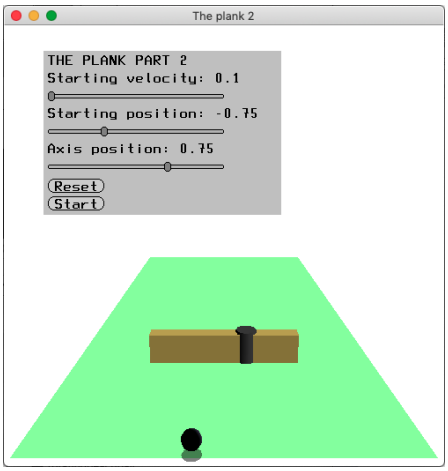
We take a snippet of code from the demo:

```
// Calc plankVelocity and omega
float f = ballVelocity.z * ballMass;
plankVelocity.z = f / plankMass;
ballVelocity.z = -ballVelocity.z; // Really force backwards
// Plank center is at origin!
float t = -ballPosition.x * f; // torque, really r x f
l = l + t; // really *Δt but this is an impulse
// Inertia for a thin rod/plank rotating around its center is
// 1/12·ml^2
```

```
// The plank length is 2.5+2.5 = 5
float j = 1/12.0*plankMass*5*5;
omega = l / j;
```

This is a very simple case. We update the velocity by the impulse from the ball, and then torque - rotational momentum - rotation speed, using the inertia from the table above.

We have one more demo for this, the same plank but this time anchored by an axis.



The plank 2

For this case, there will be no translation of the plank, while the intertia changes as the axis moves. It is easy to calculate by using the special cases above. We split the plank into each side of the axis like this:

```
float d1 = 2.5 - axisPosition;
float d2 = 2.5 + axisPosition;
float j = 1/3.0*plankMass*d1*d1 + 1/3.0*plankMass*d2*d2;
```

Of course, the plank must rotate around its axis, which is the standard problem (rotation around arbitrary axis) covered by my course book [6].

Rotations with physics in 3D

The book mainly covers the 3D case, which we will briefly summarize here. We will now have to model both torque and rotations in 3D by vectors and matrices. For a force vector $\mathbf{F}$ and a radius to the center of mass $\mathbf{r}$, the resulting torque $\mathbf{T}$ is measured in Newton*meters, Nm . Then the torque is

$$\mathbf{T} = \mathbf{r} \times \mathbf{F}$$

The moment of inertia is now a matrix, describing the behavior in different directions. The inertia of any shape can be calculated by an integral over the shape, see [7].

The sequence is given in [7] but spread out a bit, so to clarify the sequence, it works like this:

• Forces give torque by $\mathbf{T} = \mathbf{r} \times \mathbf{F}$

• Torque adds to angular momentum by $\mathbf{L} = \mathbf{L} + \mathbf{T}\Delta t$

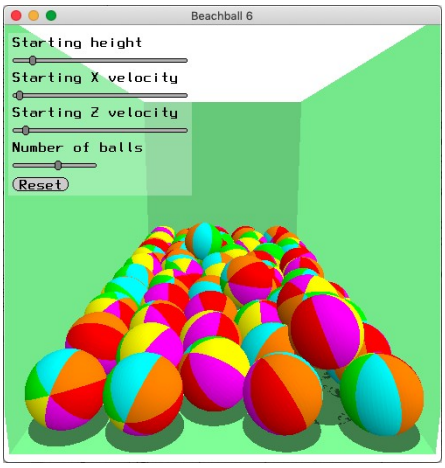
• Angular momentum maps to rotational speed by multiplication with inverse of inertia: $\mathbf{L} = J\boldsymbol{\omega} \Rightarrow \boldsymbol{\omega} = J^{-1}\mathbf{L}$

• The change of rotation is then given as $dR = \omega^* R$, where R is the rotation matrix and ω^* is a matrix derived from $\boldsymbol{\omega}$ (see [7]).

Another take at the "fake" rotations (preliminary)

With "kinematic rotation" as well as "double bounce", the balls hit floors and walls so often that it looks very good even though they do not affect the rotation of each other. Our next demo, "Many

Beachballs", introduces a larger number of balls. This introduces a number of new problems. Rotations would look wrong when the balls pile up on each other if they only adapt rotations to the walls. Also, the balls tend to shake a lot. This problem remains, and therefore I consider the demo preliminary.



Many beachballs (preliminary)

However, the first problem, making the ball rotations match when they collide, is solved, in a rather unusual fashion as far as I know. When two balls collide, I convert their rotation speed matrices to vectors (a vec3 which defines the rotation direction as well as, in its magnitude, the angle), take the difference between them, divide by two, and then assign them to new rotation speeds.

```
vec3 r1 = MatrixToVec3(ball[i].diffrot);
vec3 r2 = MatrixToVec3(ball[j].diffrot);
vec3 rr = ScalarMult(VectorSub(r1, r2), 0.5);
ball[i].diffrot = Vec3ToMatrix(rr);
ball[j].diffrot = InvertMat4(Vec3ToMatrix(rr));
```

The functions for converting from a rotation matrix to the vec3 rotation look like this:

```
vec3 MatrixToVec3(mat4 m)
{
    float a = m.m[9] - m.m[6];
    float b = m.m[4] - m.m[1];
    float c = m.m[2] - m.m[8];
    float angle = acos(( m.m[0] + m.m[5] + m.m[10] - 1)/2);
    if (isnan(angle))
           return(SetVec3(0,0,0));
    float root = sqrt(a*a+c*c+b*b);
    if (root < 0.001)
           return(SetVec3(0,0,0));
    vec3 r;
    r.x = a/root;
    r.y = c/root;
    r.z = b/root;

    return ScalarMult(r, angle);
}
```

There are two "if" statements to handle singularities that would create nan (not a number) result. Going from vector to matrix is just a variant of the usual rotation around arbitrary axis:

```
mat4 Vec3ToMatrix(vec3 r)
{
   return ArbRotate(r, Norm(r));
}
```

I consider this an experimental solution since it is not a method I have seen elsewhere. Others would tell me to use quaternions, and that would work, but... I just had to try this variant.

I based my "MatrixToVec" code on a code example found online, the site "Euclidean Space" [9].

There is a variant of this demo called "Many Beachballs Bad". In that demo, the rotations of the balls are not affected between the balls, which lets them rotate in strange ways when they are piled on top of each other. This is what the "not bad" version corrects.

If you want physically correct rotations, see the rigid body physics chapter in the TSBK03 book [7]. A big topic there is how to handle rigid objects of arbitrary (generally convex) shape, including collisions causing rotation.

Rotations based on particles

However, there is another way to deal with rotations that can be much easier, and that is to use a system based on particles connected by *springs* or, similarily, connected by *constraints*, also in TSBK03 but let us elaborate on the concept a bit. This is based on connected point masses. Consider handling movements of a car by putting one point mass at each wheel. When these point masses are pushed around, make the body of the car follow them, while the springs will make the point masses strive to get closer to the rectangular shape that they are organized in.

And then the formulas in this section will be what you need to handle collisions, including elastic and plastic as well as varying mass, and your object can even rotate. See the section on springs. In the demo there, you have a object that will rotate spontaneously.

6. Friction

Friction is described in the book but included here for completeness, with some extra information and demos.

When an object is on contact with another, movement along the objects will be subject to friction. Let us consider the case with one object sitting on the ground, a plane with infinite mass. As long as the object is not moving, the friction force F_f is the same as the forces applied to the object along the plane, up to a maximum. This maximum is the force along the normal to the plane, F_n, times the friction coefficient μ_s, where s stands for *static*.

$$F_f = \mu_s F_n$$

Once the objects are in motion, the friction will be another, somewhat smaller force. This is modelled with the *kinetic* friction coefficient μ_k:

$$F_f = \mu_k F_n$$

The opposite is true when objects are sliding along each other. The friction force will then act to reduce the movement, until they come to rest.

The dynamic friction coefficient is usually (even always) smaller than the static one. Some examples from Sergyi's lecture material include:

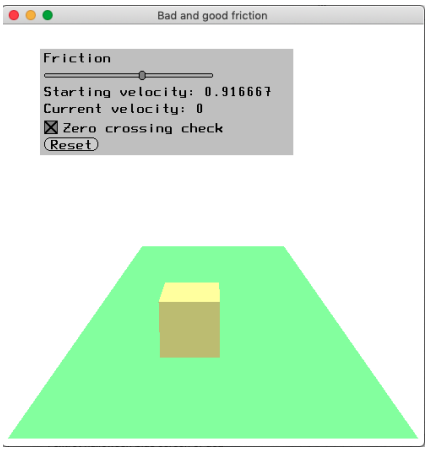
Material	μ_s	μ_k
Steel on steel	0.7-0.74	0.57-0.6
Steel on steel, lubricated	0.12	0.07
Cast iron on cast iron	1.1	0.15
Rubber-concrete	1.0	0.8
Ice-ice	0.1	0.03

The friction coefficient is usually between 0 and 1, but it is notable that this is not the case for all materials. For some materials, it can be larger than 1! One personal (not scientific) example (anecdote) that is not in the table is a car model with rubber wheels against a desk at Katedralskolan, which I personally measured to about 1.1 when my high school teacher claimed that it could not be over 1.

This is the physical description of friction. However, this gives us a practical problem *due to the discrete time*. If the force is too large to make the object come to rest, it will overshoot and cause movement in the *opposite* direction! If this happens, the object will move back and forth with small movements, and most likely one is larger than the other and the object will start crawling in an arbitrary direction.

We have a demo for this, the "sliding block" demo. Here, a cube is given a velocity, and it is also given a friction force opposed to its movement. This means that the cube will decelerate towards zero, and then it should stop, ideally. However, it does not!

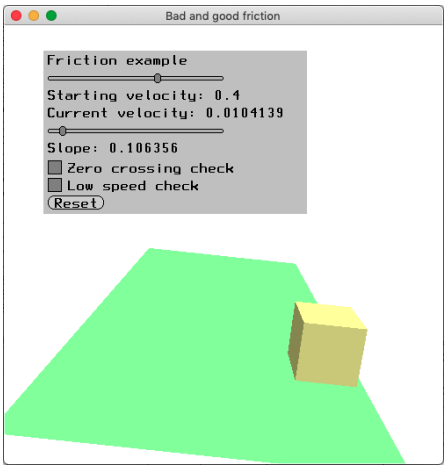
For different initial velocities, the cube will, after the initial movement is reduced to zero, start sliding right or left, randomly. However, if you add a check for the velocity crossing zero, the problem is eliminated. Another option is to reduce the friction at low velocities so the resulting force is always smaller than what would cause a velocity in the other direction.



The sliding block demo

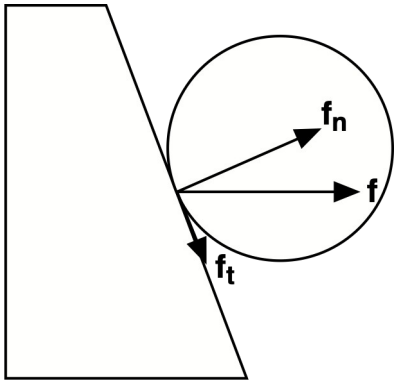
This is only a small example, where we know that there are no other forces involved. You can easily consider more complicated situations, like sliding uphill, where the velocity may or may not switch direction and that this is perfectly correct.

I have a preliminary demo for a block on a slope, but this is preliminary since I don't think it is doing everything right yet.



Sliding block 2 (preliminary)

When considering friction, collisions and rotations are affected. One example of this is the golf ball case, a case that Palmer [1] uses extensively is his demos. See the following figure:



Golf ball physics

In this figure, the force **f** is split to two composants, one along the normal, $\mathbf{f}_n$, which will be applied to the ball as in the collisions section, while the one along the tangent, $\mathbf{f}_t$, will affect both rotation and translation depending in the friction. The movement will also be affected by aerodynamic drag,

which is detailed further in Palmer's examples [1]. However, spin and drag give stronger effects on beachballs, which is why it is my choice.

7. Springs

Springs are also covered in the book [7]. It basically says that the force is proportional to the distance to the resting position of the spring, Hooke's law:

$$\mathbf{F} = -k\Delta\mathbf{x}$$

where k is the stiffness of the spring and **x** is the offset from the resting position. There may also be a dampening force. See the book [7].

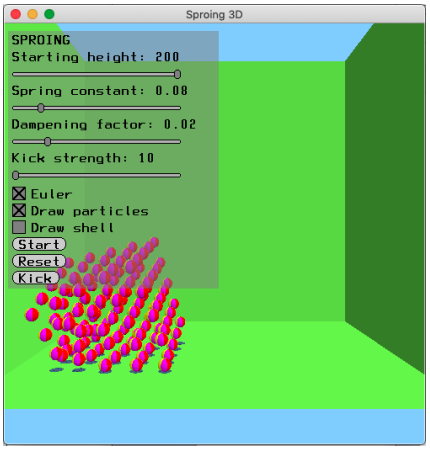
It is possible to express the movement of a spring as a differential equation and solve it analytically. This, however, has no purpose in a game or procedural animation situation. Rather, we will, again, calculate and apply the force at discrete time intervals.

This requires us to use some integration method that will neither add or remove energy when we do not desire that but create a stable animation. Euler integration tends to go very wrong while Verlet integration often is all we need. However, with proper dampening, Euler can work pretty well.

I have a demo for this, a simple 2D demo with the name "Sproing". In this demo, I have made a 2D array of particles, represented by beachballs (of course!), connected by a set of springs, in 8 directions from each particle. I chose to make no springs to the layer two steps away, although that is often used. Still, my system is fairly stable. It has sliders for the spring stiffness as well as dampening, and you can try running the system with only 4 or 6 springs per node and see what happens.

Sproing, a 2D spring system

There is also a 3D version of this system, called Sproing 3D:



Sproing 3D

Like the 2D version, the particles only connect to the closest layer, in this case 26 vertices. Springs in yet another step are often recommended, but as you can see, a single step works up to a limit. The system is pretty stable.

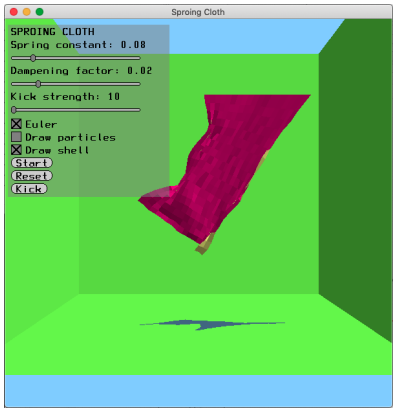
In 3D, we can clearly see that the number of springs grows rapidly. I make no attempt to accelerate this on the GPU, but for a large system it is clearly advisable. See section 5.6 in the course book [7], "Processing particle systems by multi-pass shaders".

The demo is easily configurable in the source. It is also possible to show the surface as polygons instead of the particles. See the screen shot below.

Sproing3D displaying the shell instead of the particles.

The system is quite stable even with Euler integration, but if you turn off dampening, you will see a big difference in stability. Also, a high spring constant will cause the system to explode very quickly.

Yet another variation of this is to simulate cloth. For this, we are back to the 2D system above, but we anchor the top row of particles, so the rest are hanging below it. We have yet another demo for this, "Sproing Cloth". Here, I start with the system raised above the anchored row, so it falls down from these. As always, there is some randomness from the start so it has a chance to deform nicely, which I think it does.



Sproing Cloth

What is the relevance of this for games? I think it is highly useful for many effects, not least visual, like clothes. It is also useful for special scenes that make a point in soft bodies, which can be part of the mechanics. Then, we get to the case with collision detection of these soft bodies.

8. Wind effects

This section will discuss wind effects, aerodynamic drag, spin and more. Many of these effects are of limited importance, but I hope we will find some cases that can be of interest. One of the most interesting ones are left to part 10, sailing.

Aerodynamic drag - dropping large things

If you drop something that is large compared to its weight, the aerodynamic force will be important.

The drag force works like this:

$$F_d = C_d \cdot A \cdot \rho \cdot V^2/2$$

where F_d is the resulting force, C_d is the drag coefficient, which depends on the shape, A is the area of the object facing the wind, ρ is the density of the fluid/gas and V is the velocity of the object relative to the wind.

Sergyi has a table of examples (see also below, under "Laminar & turbulent flow"), but to summarize, the drag coefficient is about 1.1 for objects with a flat surface facing the drag, around 0.5 for round objects and can be as high as 1.4 for a concave object.

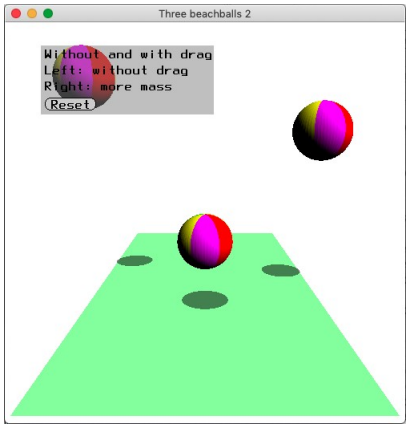
The density ρ for air is, comfortably, around 1.0 kg/m^3. It varies from 1.225 at the ground to 0.9 at high altitudes, according to Palmer [1]. It should be noted that the density, and thereby the drag (hydrodynamic drag) is several orders of magnitude higher. According to Palmer [1] fresh water has a density of 1000 kg/m^3 while seawater has 1025–1030 1000 kg/m^3. This drag is part of the sailing simulation part below, where it is significantly different in different directions due to the shape of the hull. More about that later.

These exact numbers are not necessarily of much interest to us, as long as the effect is good. What matters to us most is the fact that the drag varies *linearly with area and by the square of the relative velocity*.

I repeat: Drag is linearly proportional to the area and proportional to the *square* of the relative velocity!

In the case of the arrow, the area is very small and therefore the drag is very small compared to its momentum. So let us turn to bigger objects.

"Three beachballs 2" shows the beachball example but with drag on two of the balls, and higher mass on the right:



Three beachballs 2

As expected, the middle one slows down fast, the right one slower.

What happens, physically, is that this: The gravity causes a force f = m*g, the drag force is

added to that, $f = m*g + f_d$, and then the acceleration is $a = f/m$, so the drag is divided by the mass.

In code, the drag can be calculated like this:

```
    float fd = Cd*area1*1*v2.y*v2.y/2; // drag Fd = Cd·A·ro·V2/2, A = r2*M_PI, Cd = 0.1, ro = 1
    float fg2;
    if (v2.y > 0) fd = -fd; // Must be against the movement
    fg2 = -g*mass + fd;
```

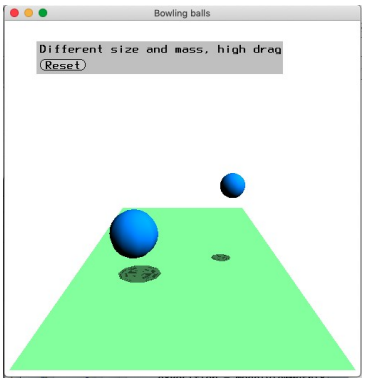
Let us look at the *mouse and elephant* example.

We give the elephant the radius r_e and the mouse r_m, where $r_e >> r_m$. This gives the elephant a mass of $m_e = k_m r_e^3$ and an area of $a_a = k_a r_e^2$. Similarly for the mouse, producing mass m_m and area r_m.

When both start falling, they have no velocity and get the acceleration g, which means a downward force of $m_e g$ for the elephant and $m_m g$ for the mouse. Both will momentarily start falling with the same velocity.

But then they both gain velocity, and the drag becomes $F_d = C_d \cdot A \cdot \rho \cdot V^2/2$. The drag is proportional to the square of the radius while the gravity force is proportional to the cube of the radius. The higher mass will cause a higher force to overcome the drag and the elephant will fall faster.

We can solve this problem with analytical integration, but that is not how you get the effect in a game situation. So let us do it numerically, by iterations! This I call *bowlingballs*, with a large and a small bowling ball (representing the elephant and the mouse). Unlike beach balls, I consider them to be solid. They also get extremely large drag so we get some time to spot the difference.



Bowling balls

And, for these bowling balls, the elephant does indeed fall faster. The integration is like "Three beachballs 2", but with different mass and area.

But let's also complete the formulas. We have a lot of constants that we lump together to simplify things.

$$k_d = C_d \cdot \rho \cdot /2 \Rightarrow F_d = k_d \cdot A \cdot V^2$$

but the area is proportional to the square of the radius, so then we can do

$$A = k_a r^2$$

$$k_{ad} = k_d k_a$$

and get

$$F_d = k_{ad} r^2 \cdot V^2$$

The mass is porportional to the cube of the radius so:

$$m = k_m r^3$$

$$F_g = m \cdot g = k_m r^3 g$$

$$F_{total} = \max(F_g - F_d, 0)$$

which gives us the acceleration

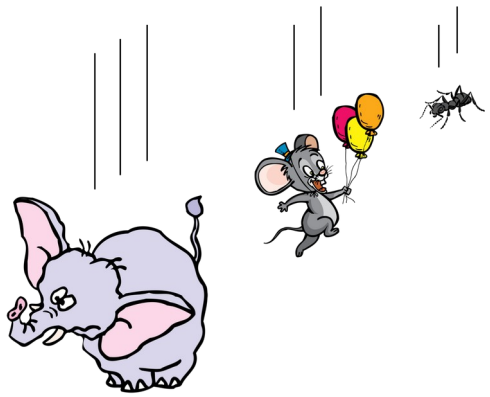
$$a = F/m = \max(F_g - F_d, 0)/(k_m r^3) = \max(k_m r^3 g - k_{ad} r^2 \cdot V^2, 0)/(k_m r^3)$$

Simplify again with $k_{adm} = k_{ad}/k_m$ and we get

$$a = \max(g - k_{adm} \cdot V^2/r, 0)$$

In the end, these formulas end up to a linear function of 1/r, and it should be clear that a small radius will, for objects that are otherwise similar, will get a bigger impact from drag than the larger one. Thus, the larger object will fall faster, and reach its final velocity later, when F_g and F_d balance and no more acceleration occurs.

We may initially assume that the objects are relatively large, but we can see that we could make the case of a mouse and an ant and the ant falls slower. You know the old saying, the bigger they are, the heavier they fall, and that is correct - with same density and when aerodynamic drag is considered.



Yes, the elephant will fall faster!

That was for similar objects, but of course we can vary other parameters. How about beach balls; what would happen if we do this with beach balls of different size? Since it is a thin shell, both drag and mass are proportional to the square of the radius so they would fall at the same speed!

Spin effect - throwing that beach ball

Spin is important in golf, table tennis and other sports dealing with balls. The effect is perhaps most central to table tennis, where spin is used and varied for every single shot, and you can put very strong spin on the ball due to the rubber coating of the racket. For demonstrations, our favorite case, the beach ball, fits just fine.

The effect of spin will push the object up or down. This is called the *Magnus force*. We take the equation from Palmer[1], page 124:

$$F_M = 0.5 * C_L \rho v^2 A$$

where C_L is the shape dependent lift coefficient, ρ is the fluid (air) density, A is the area and v is the velocity magnitude.

C_L includes the rotational speed. For a sphere this is

$$C_L = r\omega/v$$

where r is the radius, ω is the rotational speed and v is again, the velocity magnitude which is

thereby turned into a linear dependency.

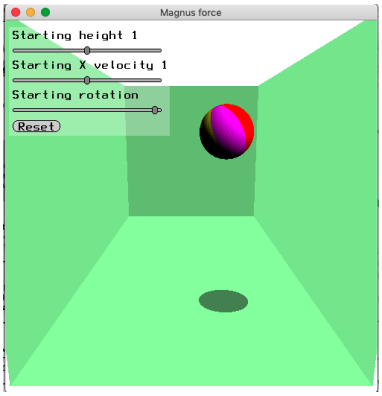
And this will tell us what we need: The Magnus effect is proportional to the velocity, perpendicular to the velocity and the spin axis, and for most cases we can set the other values more or less arbitrary to get the effect we want. And then I get the "hack" version of the Magnus force:

$$F_M = k\omega v$$

where k is a constant that approximates the air density, area and shape dependency. I may set k to whatever fits my purpose!

Finally, most importantly, backspin will lift the ball, forward spin will push the ball down.

And we have a demo for this, the "Magnus" demo:



Magnus force demo

This demo, of course, exaggerates the effect quite a bit, but the effect should be clear, higher velocity and higher spin will create more force.

This asssumed that the spin is around the z axis. A more general expression is

$$\mathbf{F}_M = k(\boldsymbol{\omega} \times \mathbf{v})$$

where $\boldsymbol{\omega}$ is the rotation vector and $\mathbf{v}$ the velocity vector. With this expression, we can make spins that causes sideway forces.

Laminar & turbulent flow

But a beach ball would not fall straight down, you say. It will wobble around in the air. This is due to wind and turbulence.

Laminar flow is the straight wind that we described in the above. Turbulence will affect the C_d parameter. It typically gets lower in turbulence. Palmer [1] lists the following cases:

Laminar and Turbulent Drag Coefficients

Shape	**Laminar C_d**	**Turbulent C_d**
Sphere	0.4–0.47	0.2
2:1 Ellipsoid	0.27	0.13
Circular cylinder	1.2	0.3
2:1 Elliptical cylinder	0.6	0.2

Palmer states that this is "all you need to know about turbulence" but without saying when to use this knowledge. Possibly we can apply it by making the force different indoor and outdoors? I would suggest that we could also may try to model the turbulence, how it changes the forces. However, I must argue that this kind of simulations are a bit out of scope for our topic. We can probably model it with randomness.

However, if you want a more physical model, you can use models like the Navier-Stokes. Wikipedia lists a number of alternatives. [11]

Aeroplanes

Aeroplanes pose more than one additional concept. They are also quite interesting from a game perspective, and realistic physics can be very important, both for pure flight simulators as well as dogfight games.

The primary topic here is the lifting power from the wings. It is proportional to the velocity and directed along the plane's up-vector. Do we need to know more? It depends on how accurate simulation we want.

There another issue with aeroplanes, relevant for propeller planes, is the gyro effect from the propeller. The rotation of the propeller affects how fast they can turn, differently for different directions.

TO DO: Cases above in some detail, especially the gyro effect of the propeller.

A third issue is the one of turbulence. Here, I would argue that some randomness to velocity, lift and rotation will do it.

So, for these issues I argue that we can do most of the effects without precise physics, but there are some cases where we want to enhance the detail.

Wind effect on objects - Buster Keaton or confetti

Do you remember that amazing scene with Buster Keaton in a town being torn down by a hurricane, from "Steamboat Bill Jr.", 1928? This scene is of interest for us since it shows the effect of extreme wind. That is indeed a case that could make a game-like situation. Imagine a game with a storm scene like that! (And Keaton did it for real!)

So how do you model this? You need wind force, collision detection, rules for when parts are torn off, and objects certainly should rotate and move due to wind and turbulence.

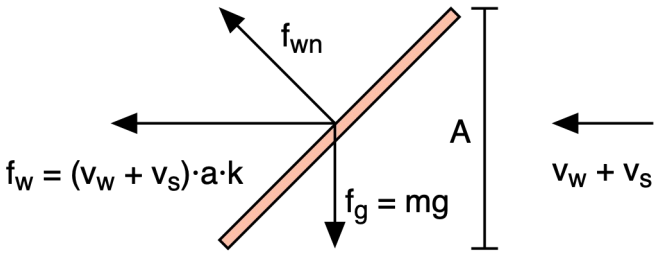
Some parts may just fall down onto the ground, others can be torn off and fly away. I can only draft this interesting scenario.

An object flying in the air, most likely flat panels for this scenario, will be affected by a force caused by the difference between its velocity and the wind velocity times the area that the object has seen from the direction of the wind. Thus, the wind effect will be proportional to the dot product of the normal vector of the panel and the wind direction. It will also get a force upwards, lifting or pushing down. Given a significant rotation, the panel will move up and down in the air.

The effect is similar to that in a sail, see the sailing ship section. A different case when this is relevant is for simulating confetti flying in the air.

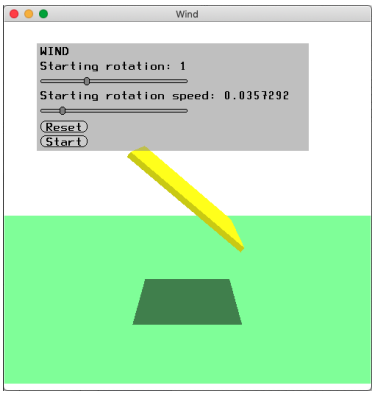
In our model, we assume k is constant and we just care about the area seen from the wind direction. It is really dependent on the objct shape, the C_d parameter above, which can vary with direction.

The wind $\mathbf{v}_w + \mathbf{v}_s$ (wind plus movement) multiplied by the area a and the shape constant k gives a force $\mathbf{F}_w$, which is then projected to $\mathbf{F}_{wn}$ normal to the panel which will then lift or push down the panel as well as moving along the wind direction.



Wind on panel

This is demonstrated by the demo "Wind" where a rotating panel is affected by wind. See the screenshot below. In that demo, a single panel is rotating in wind, which will make it move with some variation caused by the orientation of the panel.



Wind demo

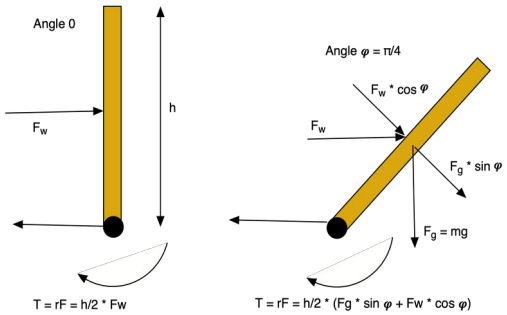
For the town case, this is quite suitable for things like loosened panels from rooftops. Another case for the town being torn down is objects that are partially archored. Such an object will rotate around its anchor point, potentially coming loose with a significant rotation. A suitable model is to accumulate the force, the tear on the anchor, so it will break after some time.

Now, let's also do the Buster Keaton falling wall:

As above, splitting vector into composants is the key, but now we also have a fixed axis, the contact point of the wall to the ground. So the situation is similar to "The plank", with forces creating rotation.

In the figure below, we have a wall with height h and wind creating a force F_w when hitting the plank perpendicularly (depending on the area, but the area is constant so we bake that into F_w). This causes a torque of h/2 * F_w.

When the wall falls due to this torque, it rotated to an angle φ and F_w gets reduced by cos φ, but we also get torque from the gravity, F_g = mg, projected to the normal of the wall, which then is F_g * sin φ.



The falling wall

For this case, the wall speed $\mathbf{v}_s$ is limited but not zero. I choose to ignore it here.

But we are not done yet. The forces above also have a composant along the tangent of the wall, which creates a force against the contact point with the ground. If F_w is small, this force will simply get an opposing force from the ground and cancel out, but if F_w is large enough, we can consider a breaking point when it comes loose and we move to the loose panel above!

9. Simulating sailing ships

Above, we considered the wind effect on projectiles. However, a truly interesting case for wind is *sailing ships*. We have seen several games where this is important, like the old game "Sid Meier's Pirates!" and many others.

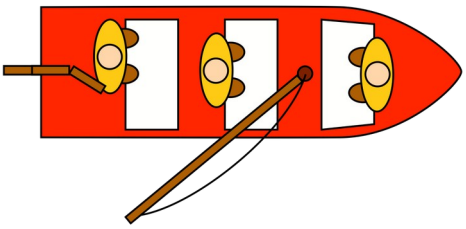
If you have been sailing, many of these effects are well known to you, but can you simulate them in a realistic fashion?

In my opinion, a decent iterative simulation of sailing is much more interesting than modelling how a boat moves up and down (although that can be valuable for hard-coding a realistic movement), since the sailing physics is something that the player can control and master with a visible effect. Therefore, you can build a whole game around that.

I have no ambition to make a perfect simulation of sailing, but to cover the most fundamental issues that are enough for our purposes: Direction of wind, sail and ship, the three most typical sailing situations, and some dependency of the slant of the ship, which affects the effective area of the sail. Finally, the water drag is an important component, more important that you may think at first, because it will reduce sideways movement. You can go much further with more detail (hinted in the picture below), but that has no place in fundamental game physics.

There are still a couple of parameters to adjust. How much dragforce will velocity cause, how much force will the sail produce from the wind, how heavy is the boat? All these are parameters that are really physical parameters that you need to adjust to match the desired effect.

Much of the modelling is made by *splitting vectors* along another vector. There are multiple cases for that operation. You split a vector into one component parallel to the given direction vector, and one perpendicular. This will operate on the boat direction, boat velocity, wind direction and the direction of the sail.

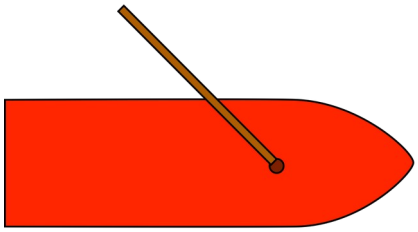


(Relatively) detailed small boat with curved sail, rudder, and people.

You may notice that the people in the boat are dressed in orange. This represents *life jackets*! This is not important for a simulation but only a good thing to include as a good example. So please put life jackets on your simulated sailors!

The sail effects is based on the relative (apparent) wind direction, which is the difference between the actual wind v_w and the velocity of the ship v_s. So, $v_a = v_w - v_s$. On to of that, we have the drag from the water, which creates a backwards force, assuming that there is no flow in the water. In the following, we assume that there is no flow, the water is still.

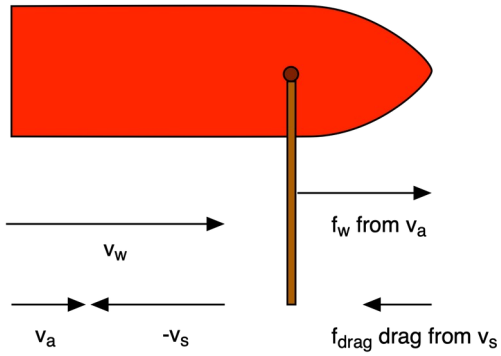
We assume (not at all correctly) that the sail is *planar* and *friction free* which gives us a simpler model like in the figure below. The wind against the sail will then only cause a force perpendicular to the sail. We can model this force with the aerodynamic drag formula, which means that the force is proportional to the area.



Our boat model: Planar sail

There are three main cases that I want to cover, to scud, sailing downwind, sailing with the wind (swedish: läns), beating/tacking, sail almost against the wind (swedish: kryssa), and to sail with the wind from the side, sidewind (swedish: halvvind, sidvind).

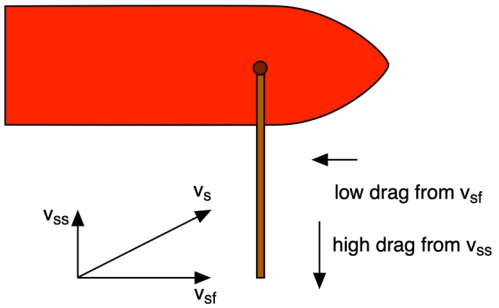
Downwind is the easiest case. We just take the apparent wind and make that create a force due to aerodynamic drag.



Downwind, force from apparent wind minus drag

Drag from the water is important here. The keel is designed to keep the drag low from the front, to make the boat move fast forward, but high sideways. So, in order to calculate the proper drag, you need to split the boat velocity (relative the water, if that moves) and split it along the direction of the boat. The sideways component should then be multiplied by a small factor (see the section on aerodynamic drag above), and the component along the boat with a higher, different for forward movement and backwards.

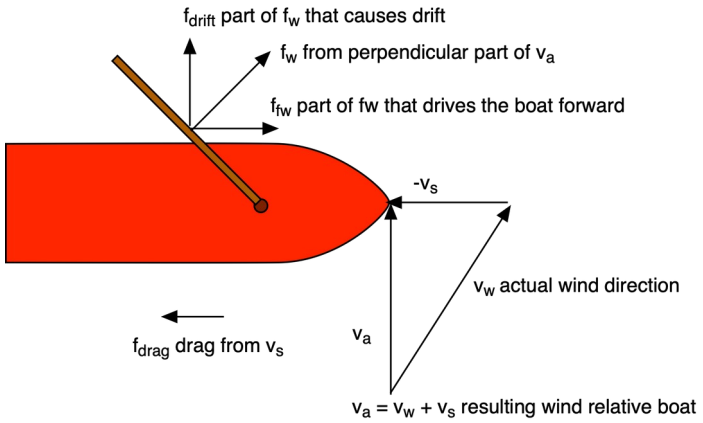
But we can also decide to simplify the model even more and eliminate the sideways drag as well as the drift (consider the sideways drag infinite = no drift) and decide that the boat goes straight forward no matter what. This is a simple model that will eliminate several components of the model. Is it good enough? You have to decide that.



Direction dependecy of drag effect from water

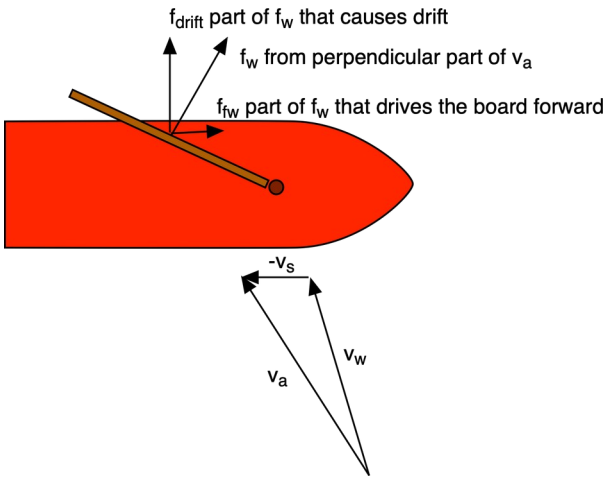
Sidewind works like this: The sail is always at an angle out from the boat, somewhere around $\pi/4$ (45°). The effect from the wind can now be split into to parts, one along the sail and one perpenducular to the sail. The one along the sail will not affect the boat since the sail is friction free.

The perpendicular one will create the force f_w. This force now be split into two composants, one along the boat and one perpendicular to the boat. The one along the boat wll drive it forward, opposed by the drag form the water. The perpendicular one will push the boat sideways, which causes *drift* (swedish: avdrift), but as mentioned above, the keel of the boat is designed to give a higher drag force in that direction and limit the drift.



Sidewind

Beating, which means sailing against the wind, is the most interesting case and where the sailor really needs to find the optimum. It works exactly like the sidewind case, only with a steeper angle.



Beating

So, the total system for this simplified case consists of a force from apparent wind that drives the boat, another part from apparend wind that causes drift (but limited by the keel), and one drag force from the water.

All in all, in a game, you need the player to steer the boat and control the angle of the sail. From there, you can calculate the forces to see how much (and if) the boat moves forward.

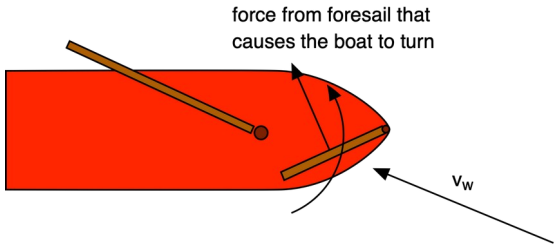
Controlling the sail

Most of the time, you don't set the angle of the sail explicitly. You rather pull in the sail more or less, and the resulting angle always push the sail downwind. You rarely want to force the sail upwind since that would push the boat backwards!

There is, however, an exception to this, and that is a trick to turn the boat quickly. When you have two sails, a mainsail and a foresail, reversing the forsail can get that effect, and is useful when you fail to turn to swich side when beating. If you lose too much speed, your turn may fail, the boat

is still, you have no steering speed, but then you can steer with the foresail, get the wind back in the sail and you are back in the race!

This is now suggested as yet another possible feature for sail racing. However, from a gaming point of view, controlling both the mainsail and the foresail this way might be more controls than the gamer can handle. This is, obvioulsy, a design and user interface question.



Steering with the foresail

Additional issues

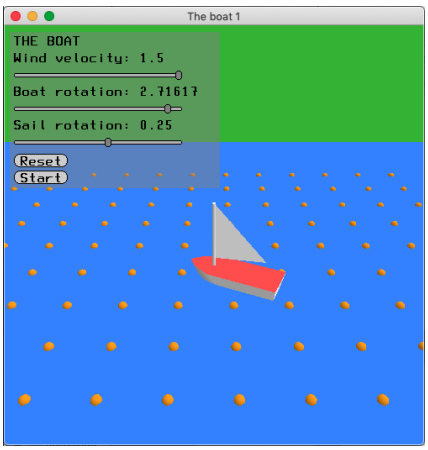
I would like to mention two more things that I find relevant, *tilt/hiking* and *jibing*.

The more f_d the boat gets, the more it tends to tilt. This reduces the effective area of the sail, pretty well approximated by the cosine of the angle. Because of this, sailors often lean on the side of the boat or even hang on the side in order to compensate (as in the figure below). I think this is called "hiking" (swedish: "burkning"). So, we don't hang on the side only in order to keep the boat from capsizing, but also for keeping the speed up!

Trying hard to keep the sail effective

Yet another game relevant issue is the *dangers of downwind*! When you sail in downwind, the sail may switch side. This is AFAIK called jibing (swedish: "gippa"). This can happen very quickly and may be pretty dangerous, since the boom moves very fast across the boat and may knock people overboard or hurt them. That is usually considered a problem, but in a game, dangers are only part of the fun!

And of course I have a demo of this. "Boat 1" gives you a simplified model, with flat, friction free sail. The boat is centered on the screen but a number of orange balls represents the movement. The wind comes from the left. You can manipulate the sail direction as well as the boat direction. The demo models the mapping of the wind towards the sail, splitting wind direction towards the sail and then the sail effect towards the boat direction. Drag and drift are included but are small.



Boat 1

There are many modifications that can be made. There are no islands, no opponents, and no jibing or tilt. Furthermore, the sail direction and boat direction are set directly. In reality, as mentioned above, the direction of the sail is always downwind, and the boat direction can only be changed by the rudder, which makes turn in low speed hard or even impossible.

All in all, simulating sailing boats may be both easier and harder than you first think. Much of the problem is solved by projecting vectors on each other, splitting vectors, but there are a lot of issues that you may or may not want to include in your model.

10. Waves

When talking about boats, waves is a special part of the problem that needs to be handled separately.

Moving water, including waves, is a very diverse subject with many methods, with many different choices depending on situation and ambition level. These were covered only briefly in the book [7]. These methods include:

• Kinematic methods, model waves programatically. This includes modelling from harmonic functions but also, as a much more realistic extension to the former, Gerstner waves.
• Heightfield approach, modelling water movement as exchange of matter from cell to cell, which essentially means between pillars of water.
• Particle systems.
• Field-based models, volumetric and grid models using fields of velocities and more.

Out of these, we will primarily cover Gerstner waves, as being a relatively easy but still nice looking solution.

Surface waves

Sometimes it is enough to get the visual impression of waves without changing the actual geometry. We will here mention two approaches:

• Bump mapping

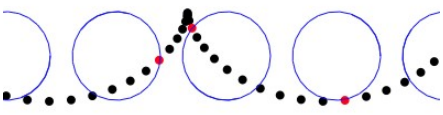
• Scrolling textures with displacement

Bump mapping for waves is a matter of modelling the normal vectors of the waves but not changing the geometry. The bump mapping technique is covered in SHWMTS [7] but the next problem is how to model the variations.

Scrolling textures is an interesting trick to simulate the refraction of the water and thereby giving an impression of waves in a very simple way. This is based on using a texture that is used as offset for another texture, displacing where the texture lookup is made, and scrolling this texture over the other. I have an older demo for this.

Gerstner waves

Trochoidal waves or *Gerstner waves*, named after Franz Josef Gerstner [12], is a popular model that produces pretty good waves with pure kinematics. It can be derived from a mathematical model of waves, from the Euler equations, but in practice it can be modelled kinematically with rotating circles, which makes it easy to implement. What you do is to take points on a surface and offset them by a point on a circle, and the point on the circle should vary over the surface. See the figure below.



Geometric effect of Gerstner waves.

It is possible to put much detail in the components of the wave, but a simplifed form can look like this:

```
vec3 p = in_Position;
p.y += sin(time + p.x)*amp;
p.x += cos(time + p.x)*amp;
```

```
gl_Position = projectionMatrix * modelviewMatrix * vec4(p, 1.0);
```

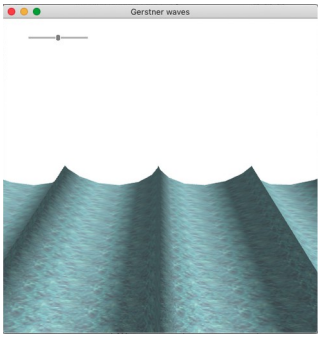
This is straight from my vertex shader, and the geometry is a tesselated plane. I only made the amplitude configurable here. Scaling time would also be desirable and simple. Then we get this more general formula:

```
vec3 p = in_Position;
p.y += sin(speed * time + wavenumber * p.x)*amp;
p.x += cos(speed * time + wavenumber * p.x)*amp;
gl_Position = projectionMatrix * modelviewMatrix * vec4(p, 1.0);
```

This will produce a simple but pretty convicing wave if the parameters are within reasonable bounds. When you experiment with this, you will easily find cases where the wave deteriorates and self-intersects. You need to avoid these.

I have made a simple demo using the formula above. See the screen shot below. It only has diffuse light, but specular light is obviously desirable.

Beyond this, you can add some noise, ripples, random variations, multiple waves, but the Gerstner wave model is a good foundation to build from.



Simple Gerstner wave in a 3D scene

An obvious limitation of Gerstner waves is that the model does not support interaction with external forces. It is just a model of how the waves move as an independent system. Models that handle such interactions include particle systems like SPH (see below) and field-based models based on models like Navier-Stokes' equations.

Smoothed Particle Hydrodynamics

--to do--

-- picture, demo, more detail --

Conclusions

What you have here is a compact summary of the topics I find most relevant for the TSBK03 course, and the subject in general. I hope that this fills the gap between the old high school physics and the advanced physics described in the TSBK03 book. My ambition has been to put much of Sergyi's material in (digital) print, while doing my best to make it even more relevant to the game physics subject. I also take the liberty of including some non-physical shortcuts that are useful for pseudo-physical effects. This includes the numerous new demos that I hope are useful and enlightning. My plan is to upload them all to my demo archive [10].

References

[1] Grant Palmer, "Physics for Game Programmers", APress 2005

[2] David M. Bourg, "Physics for game developers", O'Reilly Media 2002, 2nd ed 2013

[3] Rick Parent, "Computer Animation", Morgan Kaufman 2008

[4] David H. Eberly, "Game physics", Focal Press US 2010

[5] David Baraff, "Physically based modelling", SIGGRAPH 1999, especially parts on rigid body dynamics, https://graphics.stanford.edu › papers › phys_model, retrieved 2022-10-15.

[6] Ingemar Ragnemalm, "Polygons feel no pain".

[7] Ingemar Ragnemalm, "So how can we make them scream".

[8] Sergiy Valyukh, TSBK03 lecture slides for lecture 7-9. https://computer-graphics.se/TSBK03/

[9] Matrix to angle code, EuclideanSpace. https://www.euclideanspace.com/maths/geometry/rotations/conversions/matrixToAngle/ retrieved 2023-03-16.

[10] The Polygons Feel No Pain demo archive. https://computer-graphics.se/demopage/

[11] Wikipedia on turbulence modelling, https://en.wikipedia.org/wiki/Turbulence_modeling retrieved 2023-03-22.

[12] Gerstner, F.J. (1802), "Theorie der Wellen", Abhandlunger der Königlichen Böhmischen Geselschaft der Wissenschaften, Prague. Reprinted in: Annalen der Physik 32(8), pp. 412–445, 1809.