Technology Demonstration

Wind and Air Drag System

Real-time simulation of drag forces on arbitrary scaled meshes coupled with a dynamic environmental air speed provider

Dan Toškan,
March 2020

ABSTRACT

Fluid simulation has been around for a long time, but has mostly been absent from video games. The fundamental problem keeping this from happening is the fact that fluids (and their close relatives, gases) cannot be simulated accurately without splitting them up into very tiny voxels (3D pixels) of space, and using some behavior observed in the real world, to accurately predict what will happen next with the fluid or gas - like how will it flow around an obstacle, and how much force it acts with. But games don't need to be hyper-realistic simulations, and as such don't need the level of detail provided by these methods. Attempting less intensive approaches (by, for example, increasing the size of the individual voxels), substantially reduces the accuracy and gives unnatural results. Simulations of this detail are a significant overhead to run in real time while also rendering modern video games, and have therefore rarely been implemented with great accuracy. There are a number of methods we can use to optimize the voxel approach, but for our purposes, this was never going to work both efficiently and accurately. Instead, we make use of the fact that all objects in a video game are neatly defined with triangles (meshes), which we can leverage to solve our questions: how does airflow, from a given direction, affect a triangle? With what force is it pushed back? How large is the torque exerted on it, and around which axis does that torque act? If we answer these questions for a general triangle in a general orientation - and if we can do that every frame, for every triangle of each objects mesh - then we have created a real-time model of air drag. Add in a 3D input for the speed of air at a specified point, and we've also got a wind system. This may sound counterintuitive, but compared to dividing the space around the triangles into tiny voxels, and running calculations for each of those, it results in a substantial performance improvement. This paper provides an overview of how the model predicting drag forces and torques works in *Project Borealis*, what shortcuts and optimisations we implemented and why, how we deal with a large number of both triangles and objects, and methods for avoiding strange edge-cases.

Contents

1 Introduction

In physics, the amount of drag exerted onto a body is approximated with the drag equation:

$$F_D = \frac{1}{2}\rho v^2 C_D A$$

Here, F_D denotes the drag force in the direction of the flow velocity v. The force also scales with the density of the substance, ρ, the cross-sectional area of the object A, and a drag coefficient C_D, which in this equation is assumed to be constant, varying depending on the shape of the object and its orientation. In general this acts as a sort of "estimation multiplier". For demonstration purposes, the following are drag coefficients of some common shapes:

Shape [***orientation***]	C_D
Ball	0.47
Cube	1.05
Cube at 45°	0.8
Cone	0.5
Teardrop	0.04

But a table with all the possible orientations of every possible triangle sadly doesn't exist. As such, we need a way of calculating C_D based on the shape and position of any given triangle. This equation is still an important learning opportunity.

1.1 Basis of the drag equation

The kinetic energy T of a group of objects traveling with the same velocity is

$$T = \frac{1}{2}mv^2$$

where m is the total mass of the objects, and v is the velocity they are traveling with. This is the amount of energy they carry along with them. It is also the amount of energy you would need to stop them. Now if we imagine a plane of some shape oriented head-first into the wind (or, in equal effect, traveling at some speed head-first), air is impacting it every unit of time. How much air? Well, if the air is traveling with velocity v relative to the object, then, in a unit of time t and on a surface of area A, a mass $m = \rho Avt$ of air will hit the surface, if ρ is the density of air. That air carries a kinetic energy of $T = \frac{1}{2}(\rho Avt)\,v^2$. To stop it, we need to exert a force F on it for a period of time t. The work done by this force is $W = Fs$, where s is the path the force has acted along. In the reference frame of incoming air, it is not moving anywhere, and the object is the one moving around, with the opposite velocity $-v$. Also, due to changing reference frame, the drag force F felt by the object translates a force of $-F$ in the reference frame of air. The work done by this force is thus $W = -F\,(-vt) = Fvt$. So, back in our object reference frame, to stop the incoming air, the work W done by the drag force will have to be equal to the kinetic energy of the air.

$$W = T$$
$$Fvt = \tfrac{1}{2}\left(\rho A v t\right) v^2$$
$$F = \tfrac{1}{2}\rho v^2 A$$

Looks familiar? This is the base of the drag equation, comparing it to the original reveals the stand out C_D drag coefficient, which is the thing I earlier named "estimation multiplier". I hope this shows what I meant.

2 Drag model

From the previous example, we see that drag is really just the result of changing the kinetic energy of incoming air. So to have a general solution for a triangle, we have to know two things:

- How much air is being affected?
- How much is its kinetic energy changed by?

The first is simple, since we already found that the mass m of affected air is directly proportional to the area A of the object $m = \rho A v t$. However, this is not simply the area of the obstacle (the triangle), but the area as seen by incoming air - since the air doesn't care about areas that it will not hit. So our surface area A has to be the triangle projected onto the plane of incoming air (the normal of this plane is the velocity of air). So, say we have the sides $\vec{a_l}$ and $\vec{b_l}$ of our triangle in local mesh space, and the transform G to world space that gives us $G\vec{a_l} = \vec{a_g}$ and $G\vec{b_l} = \vec{b_g}$ respectively, we can find the projected area of the triangle by simply doing a cross product of the triangle sides, projected onto the incoming air plane, which has the normal $\vec{n} = \frac{\vec{v}}{\|\vec{v}\|}$.

$$A_P = \tfrac{1}{2}\left\|\left(\vec{a_g} - \vec{n}\left(\vec{a_g} \cdot \vec{n}\right)\right) \times \left(\vec{b_g} - \vec{n}\left(\vec{b_g} \cdot \vec{n}\right)\right)\right\|$$

Now that we have the exact area the air will encounter in its path, we can focus on the second question, which is: How much is this triangle changing the kinetic energy by? Say we change the air velocity by Δv, that means we change its kinetic energy by

$$\Delta T = \tfrac{1}{2}m(v + \Delta v)^2 - \tfrac{1}{2}mv^2$$
$$\Delta T = \tfrac{1}{2}m\left((v + \Delta v)^2 - v^2\right)$$
$$\Delta T = \tfrac{1}{2}m\left(2v\Delta v + \Delta v^2\right)$$
$$\Delta T = m\Delta v\left(v + \tfrac{\Delta v}{2}\right)$$

We can speculate that if the triangle is facing the air directly, it will completely stop the air, thereby changing its velocity by $\Delta v = -v$, and if it is parallel to the air flow, it will not change the air speed and so $\Delta v = 0$. There is no way to exactly calculate this, so we have to make an assumption which retains physical behavior. We considered several options here, but eventually, some key assumptions have to be made:

- The change in velocity Δv is directly proportional to the initial velocity v, such that $\Delta v = fv$, where f is some function. This makes sense, because if nothing else changes, but the air speed doubles, we will also slow it down twice as much, if the triangle doesn't move.
- The function declared above is $f = \cos\phi$, where ϕ is the angle between the incoming air direction $\vec{n}$ and the triangle (surface) normal $\vec{N}$, meaning $\cos\phi = \vec{n} \cdot \vec{N}$

While the first point is quite intuitive, then second assumption is a little more difficult to justify. Besides having the neat property of being quickly calculable, since we have both $\vec{n}$ and $\vec{N}$ available, it does have some physics backing it up. If you imagine a coordinate system originating at the triangle's center, with the x axis being the direction $\vec{n}$, and the z axis aligned such that the face normal $\vec{N}$ lies in the xz plane, $\cos\phi$ is the x component of the surface normal $\vec{N}$. Because the surface normal $\vec{N}$ is the direction in which impacting air will be deflected towards, it makes sense to use the x component of it as a velocity change factor, because the x axis also happens to be direction of incoming air $\vec{n}$, and that is what we used to define this imagined coordinate system.

The change of velocity is therefore:

$$\Delta v = v\cos\phi$$

Putting all of this to use, and repopulating our general equation that the work done by our drag force F is equal to the change in kinetic energy of incoming air, we get:

$$\begin{aligned} W &= \Delta T \\ Fs &= m\Delta v\left(v + \tfrac{\Delta v}{2}\right) \\ Fvt &= \rho\left(A_P vt\right) v\cos\phi\left(v + \tfrac{v\cos\phi}{2}\right) \\ F &= \rho A_P v^2\cos\phi\left(1 + \tfrac{\cos\phi}{2}\right) \end{aligned}$$

Keep in mind, that because the surface normal $\vec{N}$ will (generally) point towards the direction of incoming air, the value $\cos\phi$ will always be negative. So no need to worry about the right parenthesized expression being more than 1. Now the final part is to interpret the meaning of the result we got. As mentioned above, the surface normal $\vec{N}$ is the direction in which the triangle deflects air, so this is the direction our drag force acts in. And neatly, the $\cos\phi$ factor in the middle of the expression for F, makes sure the result will always be negative. That's good, since the drag will push the triangle *inward*, in the opposite direction of the surface normal $\vec{N}$. The final drag force is thus

$$\vec{F} = \rho A_P v^2\cos\phi\left(1 + \tfrac{\cos\phi}{2}\right)\vec{N}$$

2.1 The pillow effect

Now that we have a good way to estimate the drag force on a triangle, we have to find out where this force will act. If we just applied it to the center point, then things wound never spin - for example, something like a piece of paper tilted by 45°, would just travel in the direction it is tilted in and never change its orientation, which is completely unrealistic. If we are going for air drag, and we want more than just linear damping, we cannot ignore where the drag force should be applied. We call the phenomena of drag having a non-central acting point, the *"pillow effect"*. This is the effect of air that has hit the leading edge of the triangle, flowing across the length of the triangle, providing a cushion for the air that hits the trailing edge of the triangle. This is demonstrated in Figure 1.

Figure 1: Flow around a streamlined body (Deutsches Zentrum für Luft- und Raumfahrt, 1915)

As can be seen, the air traveling across the surface causes the rest to swerve elegantly out of the way of the surface. The force making it swerve is coming from the higher pressure air that is in contact with the surface, so a drag force per area is still felt by the obstacle. This seems very complicated to model. We assume the force dF acting on an area dA is a function of the distance l that it traveled across the surface. The only thing needed now, is a function $f(l)$ that describes how v changes with l.

$$F(l) = \rho v^2 \cos\phi \left(1 + \frac{\cos\phi}{2}\right) \int_0^l f(l) dA$$

$$F(l) = \rho v^2 \cos\phi \left(1 + \frac{\cos\phi}{2}\right) \int_0^l f(l) h(l) dl$$

$h(l)$ in the second equation is the thickness of the triangle at distance l along the the surface.

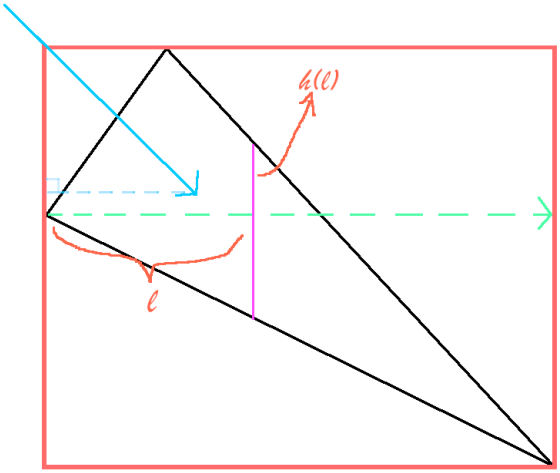

Figure 2: Visual representation of integration

The solid blue line is the direction of incoming air, coming from above the triangle, and the dashed blue line is its projection onto the triangle, say $\vec{v_{proj}} = \vec{v} - \vec{v} \cdot \vec{N}$. The green dashed line shows the axis along which we will be integrating. The purple line is the *"thickness"* of the triangle at distance l along the integration axis. If we split up the integration into two parts - before and after the middle point - we can easily define two functions $h_1(l)$ and $h_2(l)$ that describe the thickness of the triangle. Assuming L is the length of integration and b is the distance along the integration axis at which the middle point is reached, we define H as the thickness of the triangle at $l = b$. In such a setup we obtain:

$$h_1(l) = H\frac{l}{b}$$
$$h_2(l) = H\left(1 - \frac{l-b}{L-b}\right)$$

Assuming we come up with an appropriate function $f(l)$ that describes how force changes along l, we can find the force origin along the green axis by calculating the average distance along l that drag acts on:

$$l_{Avg} = \frac{\rho v^2 \cos\phi \left(1 + \frac{\cos\phi}{2}\right)\left(\int_0^b l f(l) H \frac{l}{b} dl + \int_b^L l f(l) H \left(1 - \frac{l-b}{L-b}\right) dl\right)}{\rho v^2 \cos\phi \left(1 + \frac{\cos\phi}{2}\right)\left(\int_0^b f(l) H \frac{l}{b} dl + \int_b^L f(l) H \left(1 - \frac{l-b}{L-b}\right) dl\right)}$$

$$l_{Avg} = \frac{\int_0^b l f(l) \frac{l}{b} dl + \int_b^L l f(l) \left(1 - \frac{l-b}{L-b}\right) dl}{\int_0^b f(l) \frac{l}{b} dl + \int_b^L f(l) \left(1 - \frac{l-b}{L-b}\right) dl}$$

In a similar way, the average origin along the thickness can be found as well, if we define c as the projection of the top-left triangle side onto a unit vector $\vec{h}$ that is in the direction of the triangle thickness $h(l)$ (upward). Also, let d be the projection of the bottom triangle side onto $\vec{h}$. The midpoint of the thickness is $h_{1m}(l) = \frac{l}{b}\left(c - \frac{H}{2}\right)$ and $h_{2m}(l) = d + \frac{l-b}{L-b}\left(d - c + \frac{H}{2}\right)$ for each integral, respectively.

$$h_{Avg} = \frac{\rho v^2 \cos\phi \left(1 + \frac{\cos\phi}{2}\right)\left(\int_0^b h_{1m}(l) f(l) H \frac{l}{b} dl + \int_b^L h_{2m}(l) f(l) \left(1 - \frac{l-b}{L-b}\right) dl\right)}{\rho v^2 \cos\phi \left(1 + \frac{\cos\phi}{2}\right)\left(\int_0^b f(l) H \frac{l}{b} dl + \int_b^L f(l) H \left(1 - \frac{l-b}{L-b}\right) dl\right)}$$

$$h_{Avg} = \frac{\int_0^b \frac{l}{b}\left(c - \frac{H}{2}\right) f(l) \frac{l}{b} dl + \int_b^L \left(d + \frac{l-b}{L-b}\left(d - c + \frac{H}{2}\right)\right) f(l) \left(1 - \frac{l-b}{L-b}\right) dl}{\int_0^b f(l) \frac{l}{b} dl + \int_b^L f(l) \left(1 - \frac{l-b}{L-b}\right) dl}$$

Now we just need a constant definition for $f(l)$, and we are able to analytically find the force origin. If the triangle point we started integration at has the coordinates $\vec{A}$, then we can express the origin location as

$$\vec{O} = \vec{A} + l_{Avg} \frac{\vec{v}_{proj}}{\|\vec{v}_{proj}\|} + h_{Avg} \vec{h}$$

2.1.1 Air speed falloff

Analysis of fluid simulations, like this one from Mr CFD Company (https://www.mr-cfd.com/), acted as a basis for creating the force falloff function $f(l)$.

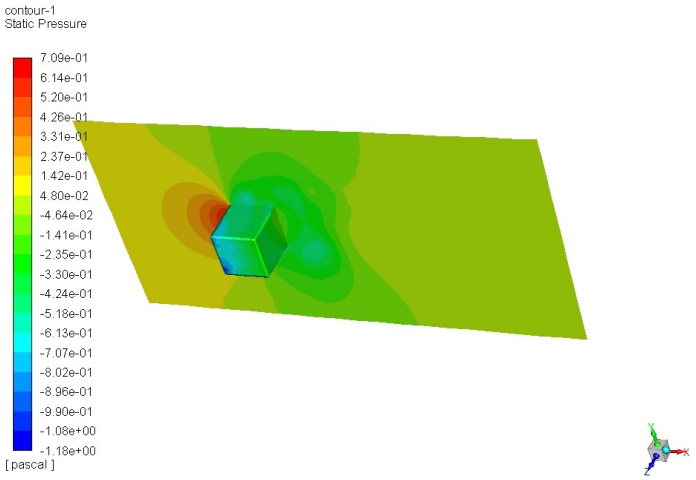

Figure 3: Pressure zones on the faces of a cube

The pressure zone in the front resembles a circle, so we use a circular falloff as an approximation: $f(l) = \sqrt{1 - \left(\frac{l}{L}\right)^2}$

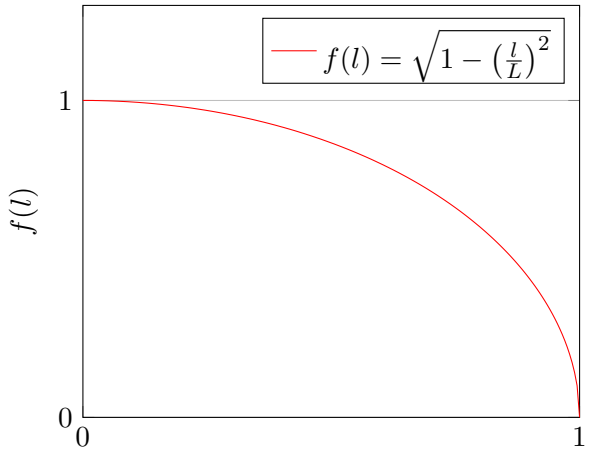

Figure 4: Circular falloff function

However, this falloff is not equal for all surfaces or objects in general. To allow for this function to change with a parameter, we used a simple linear interpolation between it and the constant function $g(l) = 1$. The interpolation parameter α is what we refer to as the *"pillow effect factor"*. So to conclude, the function we modulate force with is

$$f(l) = \alpha\sqrt{1 - \left(\frac{l}{L}\right)^2} + (1 - \alpha)$$

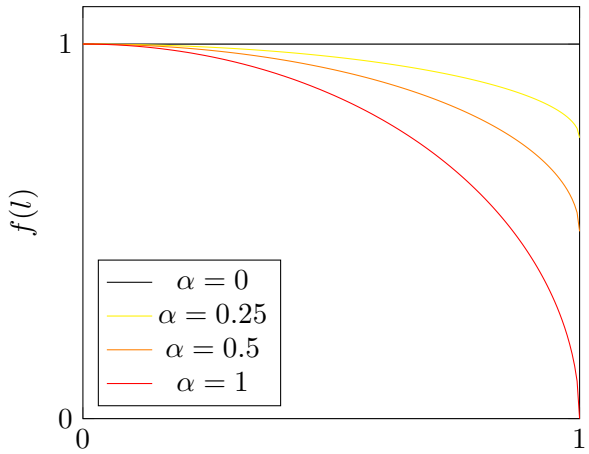

Figure 5: Falloff functions for different pillow effect factors

Solving the equations for l_{Avg} and h_{Avg} gives us a pretty long polynomial.

Using a substitution S

$$S = \sqrt{1 - \frac{b^2}{L^2}}$$

Gives us a result for l_{Avg} as

$$l_{Avg} = \frac{2b^3(p(S-2)+2) + bL^2(p(4-5S)-4) + 3L^2p(b-L)\sin^{-1}\left(\frac{b}{L}\right) + 3bL^2p\cos^{-1}\left(\frac{b}{L}\right)}{4\left(b^2(p(S-3)+3) + bL(5p-3) - 3bLp\cos^{-1}\left(\frac{b}{L}\right) + 2L^2p(S-1)\right)}$$

For h_{Avg}, the result is a little more complicated. But with pre-computing some factors and using substitution, calculating the result isn't absurdly difficult.

$$h_{A1} = 6L^4pS(H-2c) + 4b^2L^2(c(-2p(S+10)+6\pi p+20) + d(p(S-12\pi+48)-48)$$

$$h_{A2} = H(p(S-3\pi+10)-10)) + 8b^3L(12d(p(S-2)+2) - 2c(p(4S-7)+7)$$

$$h_{A3} = H(p(4S-7)+7)) + 8b^4(d(-5pS+8p-8) + H(-2pS+3p-3)$$

$$h_{A4} = c(p(4S-6)+6)) + bL^3(2c(p(-4S+3\pi+8)-8) + 2d((9\pi-32)p+32) + H(p(4S-3\pi-8)+8))$$

$$h_{A5} = 6L^2p\sin^{-1}\left(\frac{b}{L}\right)\left(4b^3(2c-4d-H) + 6b^2dL + 2bL^2(2c-H) + L^3(H-2c)\right)$$

$$h_{A6} = 16bL(b-L)\left(b^2(p(S-3)+3) + bL(5p-3) - 3bLp\cos^{-1}\left(\frac{b}{L}\right) + 2L^2p(S-1)\right)$$

$$h_{Avg} = \frac{b\left(h_{A1}+h_{A2}+h_{A3}+h_{A4}\right)+h_{A5}}{h_{A6}}$$

Though physically correct, we don't use this value for h_{Avg} in our models. Instead, we approximate h_{Avg} in the following way: All triangle mass centers and inner circle centers in local mesh space are stored in memory. We imagine the vector between these two points $\vec{e}$ as a sort of eccentricity. In general, this vector approaches being perpendicular to the longest of the triangle sides. The longest of the two sides also coincides with the part of the triangle which has more area. So, if we bring back our previous definition of $\vec{h}$, projecting $\vec{e}$ onto it gives a sensible result of how far the force origin should be moved along the $\vec{h}$ axis. So, to recap we estimate h_{Avg} as $h_{Avg} = \vec{h} \cdot \vec{e}$.

2.2 Rotation

Linear movement isn't enough to achieve realistic behavior, because in some cases, the surface we want to calculate the drag of is moving edge-first. For example, imagine a cube rotating around the vertical axis. For each of the faces, their velocity will lie in the plane of the face, so in our model, we will skip these faces because they're not traveling head-first - meaning no drag would be applied to the cube. We are faced with the problem of modeling drag for a plane rotating around an arbitrary axis. Immediately we can deduce the axis of rotation is the center of mass of a specific triangle, because anything else would be counterintuitive and would also be harder to model.

2.2.1 Radial airspeed distribution

Because the triangle is rotating, it is experiencing forces that are proportional to the distance from the rotation axis. We know the triangle's rotation axis and the speed at which it is rotating. Both of these are packed into a vector $\vec{\omega}$, called the *angular velocity*. The direction of $\vec{\omega}$ is the right-hand-rule-determined axis of rotation (though in Unreal Engine, the angular velocity is the left-handed version), while the size of the vector $\omega_0 = \|\vec{\omega}\|$ is the rotation frequency in radians per second. Thus, the velocity $\vec{v}$ at point $\vec{r}$ from the center of mass is obtained by

$$\vec{v} = \vec{\omega} \times \vec{r}$$

It should be noted that for a given angle ϕ between vectors $\vec{a}$ and $\vec{b}$:

$$\left\|\vec{a} \times \vec{b}\right\| = \|\vec{a}\|\left\|\vec{b}\right\| \sin(\phi)$$

The approach used to solve the rotation problem is very similar to the previous solution for linear velocity drag. In general, there will be an axis we will integrate along, and a function that will describe how the speed changes with the distance along this axis. Because the integration axis will start from the center of mass, and we defined $\vec{r}$ as the offset from the center of mass, the distance along this axis will be $\|\vec{r}\|$ in any case. As we can tell from the above equation, the size of $\vec{v}$ changes linearly with $\|\vec{r}\|$ and ω_0 and has a multiplicative factor of $\sin(\phi)$ where ϕ is the angle between $\vec{r}$ and the axis of rotation $\vec{\omega}$. Let's define a function describing this change as $\|\vec{v}\| = f_{(r)} = \|\vec{r}\|\omega_0 \sin(\phi)$.

2.2.2 Integration axis

It makes sense to integrate along an axis that lies on the triangle itself. And further, it makes sense to choose an axis where we won't have to deal with the angle ϕ mentioned above. So, let's pick an axis $\vec{x}$ that is both on the triangle and perpendicular to the axis of rotation, so that $\sin(\phi) = 1$ and thus $f_{(r)} = \|\vec{r}\|\omega_0 = x\omega_0$. An axis like this can be found easily with

$$\vec{x} = \vec{\omega} \times \vec{N}$$

Where $\vec{N}$ is the face normal. Like in the previous solution, we can define $\vec{h}$ as the axis along which we measure the *height* of the triangle: $\vec{h} = \vec{x} \times \vec{N}$.

2.2.3 Incoming air

Along the axis $\vec{x}$ we selected, the direction of incoming air is constant, given by the original definition of $\vec{v}$. Because we know how the size of $\vec{v}$ changes with the distance along the axis $\vec{x}$, we can define a infinitesimal force dF felt at location x along the integration axis.

$$dF = \rho h_{(x)} v^2 \cos\phi \left(1 + \frac{\cos\phi}{2}\right) dx$$

Where $h_{(x)}$ is the height of the triangle along axis $\vec{h}$ at distance x along the integration axis, and ϕ is the angle between the incoming air $\vec{v}$ and the face normal - which is the same as the angle ϕ between the face normal $\vec{N}$ and the rotation axis $\vec{\omega}$, due to the way $\vec{v}$ is defined and the axis $\vec{x}$ we chose. Using the size of $\vec{v}$ we found above and (optionally) adding our falloff function $f(x)$, the differential becomes:

$$dF = \rho h_{(x)} x^2 {\omega_0}^2 \cos\phi \left(1 + \frac{\cos\phi}{2}\right) f(x) dx$$

The only problem here, is that along the integration axis, $h_{(x)}$ is not a smooth function (it is not smooth at the midpoint $x = b$ - see Fig.3). To fix this, we may break up the integration into three parts, $\int_a^b dx$, $\int_b^0 dx$ and $\int_0^c dx$, where $a < 0$, $c > 0$ and $a < b < c$. If b is more than zero, we may integrate from the opposite side to obtain a equal.

$$\int_X dx = \begin{cases} \int_a^b dx + \int_b^0 dx + \int_0^c dx, & \text{if } b < 0. \\ -\left(\int_{-c}^{-b} + \int_{-b}^{0} + \int_0^{-a}\right), & \text{if } b > 0. \end{cases}$$

In the case $b = 0$, any of the two approaches may be used. So, let's just assume $b < 0$, and if it isn't we can always say $a = -c$, $b = -b$ and $c = -a$.

2.2.4 Applying the force

Because we are talking about rotation, it makes no sense to seek a *mean force origin*, since in reality, what the triangle *feels* is actually torque, not individual forces. Following the right-hand rule, the torque felt by the body is

$$\vec{M} = \vec{F} \times \vec{x}|x|$$

And because the force F - which is in the direction of the face normal - is always perpendicular to the lever $\vec{x}|x|$, we can simplify

$$\vec{M} = \vec{h} \int_X dF_1 x dx$$

$$\vec{M} = \vec{h} \left(\int_a^b dF_1 x dx + \int_b^0 dF_1 x dx + \int_0^c dF_1 x dx\right)$$

Inserting dF_1(explained below) into the equation, while using $h_1(l)$ in $\int_a^b$ and $h_2(l)$ in the other two integrals as functions for $h_{(x)}$, we obtain a result for $\vec{M}$.

2.2.5 Tilt

Similarly to the linear problem from earlier, we have to consider that the axis we are integrating along is not necessarily an axis of symmetry for the triangle. By this we mean more of the surface could be, for example, on the upper half. This would induce a separate torque $\vec{M}_i$ around the integration axis itself. The amount of torque on this axis can be found by integrating over it and the mid-point height, expressed earlier as $h_{1m}(l)$ and $h_{2m}(l)$, but slightly different now that the integration axis starts from the mass center. $h_1(l)$ and $h_2(l)$ are also slightly different due to this.

$$h_1(x) = H\frac{x-a}{b-a} \qquad h_2(x) = H\left(1 - \frac{x-b}{c-b}\right)$$
$$h_{1m}(x) = \frac{x-a}{b-a}\left(c - \frac{H}{2}\right)$$
$$h_{2m}(x) = d + \frac{x-b}{c-b}\left(d - c + \frac{H}{2}\right)$$
$$\vec{M}_i = \vec{x}\int_X dF_2 h_m(x)dx$$

To avoid calculating the force all over again for this new orientation, we assume that $dF_2 = dF\frac{h_m(x)}{h_{(x)}}$. **Important thing to note:** We can't just conjure up forces out of nowhere. If we decide to give a portion of dF to this second torque dF_2, then we must reduce the intensity of that main torque by the complimentary factor. Let's say the original force differential was dF and the one we apply to the $\vec{h}$ axis is dF_1. In that case:

$$dF_1 = \left(1 - \frac{h_m(x)}{h_{(x)}}\right)dF$$
$$dF_2 = \frac{h_m(x)}{h_{(x)}}dF$$
$$\vec{M}_i = \vec{x}\int_X xdF\frac{h_m(x)^2}{h_{(x)}}dx$$
$$\vec{M}_i = \vec{x}\left(\int_a^b \frac{h_{1m}(x)^2}{h_1(x)}xdF_2dx + \int_b^0 \frac{h_{2m}(x)^2}{h_2(x)}xdF_2dx + \int_0^c \frac{h_{2m}(x)^2}{h_2(x)}xdF_2dx\right)$$

If we look back at the definition of dF, we can see it contains the height of the triangle $h_{(x)}$, which neatly cancels out with our definition for dF_2.

$$\vec{M}_i = \vec{x}\rho{\omega_0}^2\cos\phi\left(1 + \frac{\cos\phi}{2}\right)\left(\int_a^b h_{1m}(x)^2x^3dx + \int_b^0 h_{2m}(x)^2x^3dx + \int_0^c h_{2m}(x)^2x^3dx\right)$$

This equation shows why we can't directly integrate from b to c - the function within the integral is an odd function, so the integral of it over the zero point would start to cancel itself out. This points out another problem: one side of the integration ($x > 0$) we are basically calculating drag, since that part of the triangle is rotating *into* the air. The other side ($x < 0$) on the other hand, rotates away from the air, or *inward*, towards the center of the body. An important assumption we're making here is that the force felt by this inward-rotating part in the form of a pressure differential is equal to conventional drag at the same speeds. However, because the force on this inward-rotating part works in the opposite direction as the one on the other side, we have to negate it.

$$\vec{M}_i = \vec{x}\rho{\omega_0}^2\cos\phi\left(1 + \frac{\cos\phi}{2}\right)\left(-\int_a^b h_{1m}(x)^2x^3dx - \int_b^0 h_{2m}(x)^2x^3dx + \int_0^c h_{2m}(x)^2x^3dx\right)$$

3 Wind system

Now that we have a model for drag that takes an input velocity and applies forces to the modeled body, we can create a system that supplies part of that input. It makes sense for this to be distributed through 3D space and not just a top-down map, otherwise we could forget about any small-scale wind-creating effects (eg. air vents, the whoosh of a passing vehicle, downdraft from a helicopter, a burst pipe, etc.). The initial plan was to create a level-scale uniformly distributed grid, with spaces of 0.1m for example. Then we could add wind actors creating a certain type of disturbance around them - updating the vectors on the grid around them. This way, it would be relatively easy to extend the system to support occlusion as well - which would be really cool! However, we could never drive particle systems this way, since calculations for those are done by the GPU. The compromise was quite obvious at this point - use whatever the particle systems are using, and adapt it to get wind samples on the game thread that is running on the CPU. Luckily, Unreal Engine 4 has Global Vector Fields - actors describing an ordered collection of uniformly spaced vectors that can be placed in world space and that can affect particle systems, moving them in the direction of the closest vector (actually the average of the closest 8). Great! Now how to get that same value when it's not being calculated by dedicated GPU functions?

3.1 Trilinear interpolation

What we have to work with is the raw data of the vector field (the vectors and their coordinates), its orientation in world space and the world location at which we want to sample this field. First thing we need to do is find the points of the vector field that are closest to the sampling location $\vec{S} = (x, y, z)$, which we can define as the sampling point translated into the vector fields local space. The closest grid points to $\vec{S}$ are the corners of the cube encapsulating the sampling point $\vec{S}$. Assuming the entire field lies within $(0, 0, 0)$ and $(1, 1, 1)$, finding the extremes of the encapsulating cube as indicies (i_x, i_y, i_z as the minimum and i_X, i_Y, i_Z as the maximum) is pretty easy:

$$\begin{aligned} i_x &= \lfloor x(N_x - 1) \rfloor & i_X &= min(i_x + 1, N_x - 1) \\ i_y &= \lfloor y(N_y - 1) \rfloor & i_Y &= min(i_y + 1, N_y - 1) \\ i_z &= \lfloor z(N_z - 1) \rfloor & i_Z &= min(i_z + 1, N_z - 1) \end{aligned}$$

Where N is the number of grid points along an axis. This gives a cube with corners at indicies

$$P_1 = (i_x, i_y, i_z), P_2 = (i_X, i_y, i_z), P_3 = (i_x, i_Y, i_z), P_4 = (i_x, i_y, i_Z)$$
$$P_5 = (i_X, i_Y, i_z), P_6 = (i_X, i_y, i_Z), P_7 = (i_x, i_Y, i_Z), P_8 = (i_X, i_Y, i_Z)$$

Let's say the vectors at these points are $\vec{V}_{P_i} = \vec{V}_i$. Now lets define 3 parameters that we will use as weights: $x_d = x(N_x - 1) - i_x$, $y_d = y(N_y - 1) - i_y$ and $z_d = z(N_z - 1) - i_z$.

With those values can we find the vector $\vec{V_S}$ as

$$\vec{X}_{yz} = \vec{V}_1(1-x_d) + \vec{V}_2 x_d \qquad \vec{X}_{Yz} = \vec{V}_3(1-x_d) + \vec{V}_5 x_d$$
$$\vec{X}_{yZ} = \vec{V}_4(1-x_d) + \vec{V}_6 x_d \qquad \vec{X}_{YZ} = \vec{V}_7(1-x_d) + \vec{V}_8 x_d$$

$$\vec{Y}_z = (1-y_d)\vec{X}_{yz} + y_d\vec{X}_{Yz} \qquad \vec{Y}_Z = (1-y_d)\vec{X}_{yZ} + y_d\vec{X}_{YZ}$$

$$\vec{V_S} = (1-z_d)\vec{Y}_z + z_d\vec{Y}_Z$$

The beauty of this approach is that there is no performance penalty to the fidelity of the vector fields. A grid with a cell size of 1 centimeter does not require any more CPU resources than one with a cell size of several meters. The only cost higher fidelity brings is memory: $approx.size = 32bits \cdot 3 \cdot Res_x \cdot Res_y \cdot Res_z$.

3.2 Modular control

Because we are adding actors - vector fields - in the world separately, we can't have global wind control. For example, if we want every area in the level to feel wind from a specific direction, we need to add a vector field representing one-way wind, scale it up to cover the entire level, and orient it in the direction that we want. To make common use cases like this one easier, we developed an actor that controls a collection of vector fields. Support for animating any change, whether that be movement, rotation or scaling, was also a key feature to implement. Grouping multiple animations and playing the entire group is another important one we decided to implement. Of course, controlling the intensity (speed) of the individual fields is very important, so animations are available for that as well. Wind has very characteristic behavior we wanted to get as close as possible to. For example, when a wind gust blows through, how does the air speed change with time? We modeled this change separately after doing some research on wind gusts.

3.3 Wind gust profile

Turns out, wind gusts are an incredibly varied phenomenon. There is no *typical* wind gust profile, so from the ones that we did find, we modeled those which looked the most dramatic and have the most interesting gameplay possibilities. In particular, profiles where a low pressure zone leads the gust itself, like how water level drops before a large wave. The low pressure zone basically means that the wind speed actually changes in the negative direction at first, before the much larger, positive change comes after it. We found the simplest way to model this behavior is the following function:

$$f(t) = -\sin(2\pi x^p)\sin\left(\tfrac{\pi}{2}x\right)$$

The value of p can be used to dial in the intensity of the low pressure zone.

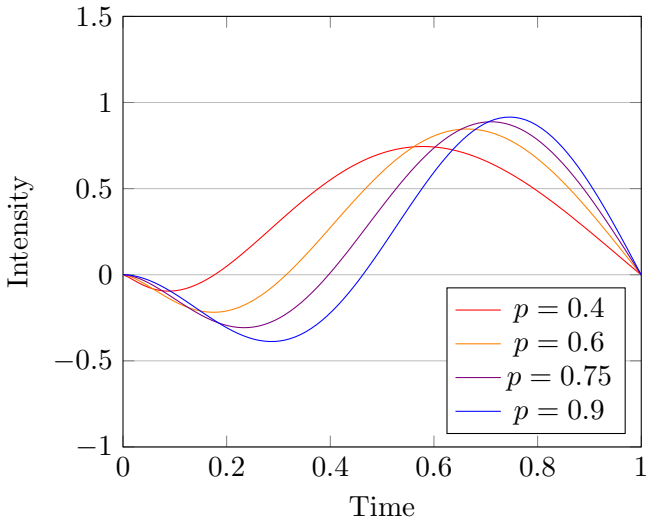

Figure 6: Plot of different wind gust profiles

3.4 Mesh-less objects

Not all objects that we would want to manipulate with wind are full-fledged physics objects or GPU particle systems. Some could be sprites, sound cues or just some non-physical objects that should react to wind. For these, we developed a component that samples wind speed using the same interface as regular physics objects, but reacts to the result in ways that can be set up. For example, movement can be locked while the component parent is rotated to face into the wind or away from it. If movement isn't locked, then the component parent moves by $\Delta\vec{r} = \alpha\vec{v}\Delta t$, where Δt is the elapsed time, $\vec{v}$ the speed of the wind and α a configurable parameter that tells the component how closely should it follow the wind (eg. $\alpha = 1$ means it travels like it was the air). This movement and the rotation described previously can both looks weird in some cases, as in reality, nothing immediately snaps to the direction of the wind. To get around this, this wind component has a configurable mass setting and an inertial vector, which basically tells it to behave like it has a mass of m and using the inertial vector, roughly approximates how it should behave when spun about which axis. This makes all acceleration and rotation look much more natural.

4 Optimization

With no optimizations, the system struggles with higher-poly objects or large numbers of objects. It quickly became clear that some large steps needed to be made to improve performance in such scenarios - no one wants frame rate drops in the midst of battle. Each system - wind and air drag - has its own solution.

4.1 Air drag

The main issue here is performing relatively expensive mathematical operations many times for a single object. Since drag on face A has nothing to do with drag on face B, it makes perfect sense to parallelize these calculations. Tasks calculating forces and torques on each face are distributed efficiently across the available processor threads thanks to Unreal Engine's ParallelFor() method. Additionally, upon first loading, a bunch of information about the mesh that is required throughout all calculations, is pre-generated, so that it need not be calculated over and over again. This includes centers of mass of each triangle, the *axis of eccentricity* $\vec{e}$, surface areas, etc.. The drag calculations themselves are written to favor RAM use over CPU cycles, even going as far as having tables of coefficient for parts where calculations of type $x(A + B \cdot C) + x^2(D + E \cdot F)$ need to be evaluated.

4.2 Wind

The wind system by itself does not require a lot of resources. Its only task is sampling vector fields and moving, rotating and scaling them as animations dictate. However, if there are many sample requests and large numbers of vector fields overlapping at sample locations, the impact is measurable. Again, since one sample is unrelated to another, the trilinear interpolation tasks are parallelized just like the air drag system.

5 Conclusion

We presented here a wind and air drag system based on simplified fluid dynamics models. The system produces realistic results in Unreal Engine 4 for use in *Project Borealis* without adding a substantial performance overhead.

There are plans in place to improve performance further, which include both software and hardware level optimizations, as well as adding a model to describe the behaviour of ellipsoids. This would be used to simulate debris or other small meshed objects, for which high simulation accuracy isn't required.

We intend to release the source code of this system, as implemented in *Project Borealis*, under an open source license at some time in the future, following further feature development and optimisation.

The *Project Borealis* team can be contacted with further comments or questions at **info@projectborealis.com**.