Simulation in Computer Graphics - Project 2

Magnus Menger, Melvin Hoogendoorn and Viljan van Raaij
Group 1

1 Introduction

In this report we describe our fluid simulation system, which is an further expansion of Jos Stam original code [1] accompanied by the *Real-Time Fluid Dynamics for Games* paper [2]. Our expansion introduces vorticity confinement, fixed and movable solid objects, collidable rigid bodies, two-way coupling and fluids with different temperatures. In section 2 we describe all features in detail and in section 3 we describe a few different scenes showing off the implemented features.

2 Implemented Features

2.1 Vorticity Confinement

A typical result of corse grid based fluid simulation systems is that you loose the small scale *swirls* that are typical of smoke due to numerical dissipation [3]. We use the approach proposed by R. Fedkiw, et al., where we apply a corrective force to the velocity field in areas where this rotational motion occurs to keep these swirls alive.

For discrete grid the vorticity $\omega_{i,j}$ for a given cell (i, j) is given by

$$\omega_{i,j} = \frac{1}{2h}[v_{i+1,j} - v_{i-1,j} - u_{i,j+1} + u_{i,j-1}] \tag{1}$$

where h is the grid spacing and $\mathbf{u} = \{u, v\}$ a 2D vector field for the velocity. Next, the normalized vector $\mathbf{N} = \{N_x, N_y\}$ that point from lower to higher vorticity is calculted:

$$\eta_x = \frac{|\omega_{i+1,j}| - |\omega_{i-1,j}|}{2h}, \quad N_x = \frac{\eta_x}{|\eta_x|} \tag{2}$$

$$\eta_y = \frac{|\omega_{i,j+1}| - |\omega_{i,j-1}|}{2h}, \quad N_y = \frac{\eta_y}{|\eta_y|} \tag{3}$$

From this we calculate the final corrective force $\mathbf{f_{conf}}$:

$$\mathbf{f_{conf}} = \epsilon h(\mathbf{N} \times \omega)$$

where ϵ is the strength of the vorticity confinement.

Our implementation has a few minor errors with the vorticity confinement which we realised after handing in the code. See subsection B.1 in the Appendix for details.

2.2 Fixed Objects

We implemented fixed rigid objects that can be placed inside the fluid, forcing the fluid to flow around them. We rasterize these objects into a grid of pointers that are null-pointers when a cell is not occupied. Otherwise they point to an occupying object.

To allow the fluid to interact with fixed objects, we introduced additional boundary conditions similar to those used for the outer borders in the original code from the Real-Time Fluid Dynamics for Games paper [1]. Specifically, for each boundary cell, we set the velocity component perpendicular to the wall, $\mathbf{u}_\perp$, to the negative of the adjacent fluid cell's value, while keeping the parallel component, $\mathbf{u}_\parallel$, unchanged. For density ρ, we copy the value from the neighboring fluid cell to the boundary cell to prevent diffusion into the wall. These conditions can be summarized as follows:

$$\mathbf{u}_\perp^{\text{wall}} = -\mathbf{u}_\perp^{\text{fluid}} \tag{4}$$

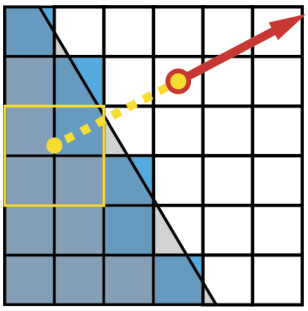
$$\mathbf{u}_\parallel^{\text{wall}} = \mathbf{u}_\parallel^{\text{fluid}} \tag{5}$$

$$\rho^{\text{wall}} = \rho^{\text{fluid}} \tag{6}$$

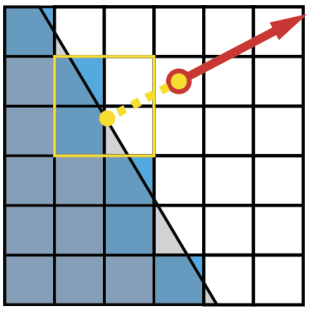
In the case of a boundary cell neighboring multiple fluids cells, the individual contributions of the different fluid cells are first calculated induvidually, then averaged together.

A downside with this ghost cell based approach is that we are limited to define walls with a minimum width of 2 grid cells, as mentioned in Real-Time Fluid Dynamics for Games [2].

We also adjusted the advection step to support solid objects. In the original code, the change of a cells values based on advection are computed by taking a backwards step along the velocity field and interpolating between the values of cells arond the position where you end up. When the position is outside of the fluid grid, it is clipped to the border. With this logic, the position could still end up inside of a solid object. To prevent this we iterate over all obstacles to detect if taking a backwards step would move us through the face of a solid obstacle. In this case we clip the backwards step to the closest such face. Figure 1 shows an example of this clipping.

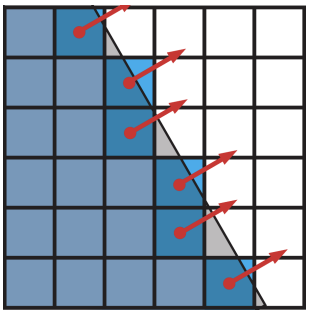


(a) Without clipping.

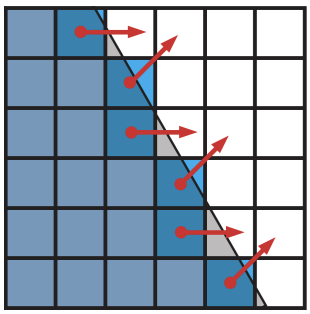


(b) With clipping.

Fig. 1: Visualization of advection backward clipping.



(a) Actual normal of the solid object.



(b) Normal determined from the adjacent fluid cells.

Fig. 2: Comparison of different normals used by the ghost cell to calculate the velocity. Blue cells are the rasterized cells and white cells are the fluid cells.

2.3 Moving Solid Objects

We have implemented solid objects with which the user can interact. For simplicity, we have focused on rectangles, but extending the code to support other shapes should be straightforward. When moving solid objects around it influences the surrounding fluid – pushing it forward and drawing it in from behind. The velocity of the fluid adjacent to the wall is given by

$$\mathbf{u} \cdot \mathbf{n}_{\text{solid}} = \mathbf{u}_{\text{solid}} \cdot \mathbf{n}_{\text{solid}}$$

where $\mathbf{u}$ is the velocity of the fluid, $\mathbf{u}_{\text{solid}}$ the velocity of the solid wall, and $\mathbf{n}_{\text{solid}}$ the normal of the solid wall. We extend this problem into a grid by rasterizing our object and assuming that the boundary is in between a ghost cell and a fluid, which gives us:

$$\frac{\mathbf{u}_{\text{ghost}} + \mathbf{u}_{\text{fluid}}}{2} \cdot \mathbf{n}_{\text{solid}} = \mathbf{u}_{\text{solid}} \cdot \mathbf{n}_{\text{solid}}$$

We rewrite this into:

$$\mathbf{u}_{\text{ghost}} \cdot \mathbf{n}_{\text{solid}} = 2 \cdot \mathbf{u}_{\text{solid}} \cdot \mathbf{n}_{\text{solid}} - \mathbf{u}_{\text{fluid}} \cdot \mathbf{n}_{\text{solid}}$$

Finally, we can represent the perpendicular and parallel component of the velocity with the following two formulas:

$$\mathbf{u}_{\perp}^{\text{ghost}} = 2 \cdot \mathbf{u}_{\perp}^{\text{solid}} - \mathbf{u}_{\perp}^{\text{fluid}} \tag{7}$$

$$\mathbf{u}_{\parallel}^{\text{ghost}} = \mathbf{u}_{\parallel}^{\text{fluid}} \tag{8}$$

As for the wall normals used to determine the perpendicular and parallel components in Equation 7 and Equation 8, we approximate them by averaging the direction of the adjacent fluid cells, which can be seen in Figure 2b. This is less accurate than using the real normals of the solid object, as seen in Figure 2a, but the idea is that the final result will be similar when we average the contribution of the different ghost cells. This way we simplify the code as we don't need to explicitly define normals for all solid objects.

2.4 Moving Rigid bodies

In addition to solid objects, we have implemented rigid bodies that can rotate. Beyond the variables we store for solid objects, we now also track each object's current orientation θ, angular velocity ω, and inertia I. Let $\mathbf{p}_{\mathbf{obj}}$ be an arbitrary point in object space, and let $\mathbf{x}$ be the position of the center of mass of the object in world space. Then we can compute the world space coordinates of p by first rotating around the origin with the current orientation of the object, and then translating with $\mathbf{x}$:

$$\mathbf{p}_{\mathbf{world}} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \mathbf{p}_{\mathbf{obj}} + \mathbf{x}$$

We compute the vertices to be rendered in world space by applying the above equation to all vertices of the object defined in object space.

Since our simulation is in 2D, we have that rotation only happens around the z-axis. Therefore the inertia tensor is just a scalar value. Our simulation only contains rectangles, We compute this inertia scalar of a rectangle as follows, where we assume that the rectangle is uniformly dense:

$$I = \frac{m}{12} * (w^2 + h^2)$$

To rasterize the object, we need to check which cells are in and outside of the object. To decrease the amount of cells that we have to check, we first compute the axis-aligned bounding box. We do this by finding both the maximum and minimum x and y values over all four vertices in world space. Then, for all cells in the bounding box, we check if it is inside our object by converting the cell coordinates to object space and checking if it is located within the vertices. If it is, we store the pointer to the rectangle in the cell.

The user can interact with the rectangle by applying a force $\mathbf{f}$ at some position $\mathbf{p}$ in the rectangle. The torque resulting from this force can then be computed by:

$$\tau = \mathbf{p} \times \mathbf{f}$$

. The change in linear force is just computed by adding vector f to the force state vector of the rectangle.

In each timestep, next to the standard velocity and position update, the angular velocity and orientation are updated as follows:

$$\omega_{new} = \omega_{old} + \frac{\tau}{I} * dt \tag{9}$$

$$\theta_{new} = \theta_{old} + \omega * dt \bmod 360 \tag{10}$$

2.5 Colliding contact between rigid bodies

We detect collisions between objects in each timestep by iterating over all unique pairs of objects. For each pair, we check whether any vertices of one object lie inside the other. For every such vertex, we apply a collision impulse. Let us call the object containing the colliding vertex the **vertex object**, and the other the **edge object**.

To compute the collision impulse, we first determine the collision normal $\hat{\mathbf{n}}$ on the edge object. This is done by identifying the edge of the edge object that is closest to the colliding vertex. The collision normal $\hat{\mathbf{n}}$ is then computed by rotating that colliding edge vector by 90 degrees.

We can calculate the velocity at a certain position p on an object in world space using

$$\dot{\mathbf{p}} = \mathbf{v} + \omega \times (\mathbf{p} - \mathbf{x}),$$

where $\mathbf{v}$ is the linear velocity, ω is the angular velocity, and $\mathbf{x}$ the center of mass of the object. Let $\mathbf{v_a}$, $\mathbf{v_b}$ be the velocities of both objects at the position of the colliding vertex. We then can calculate the relative velocity at that position along the collision normal as:

$$\mathbf{v_{rel}} = \hat{\mathbf{n}} \cdot (\mathbf{v_a} - \mathbf{v_b})$$

We apply a collision impulse only if the vertex is moving toward the surface or resting against it, i.e., when $\mathbf{v_{rel}} \leq 0$. Let ϵ be the coefficient of restitution (bounciness), m_i the mass of object i, and $\mathbf{r_i}$ the position of the collision vertex relative to the center of mass of object i. The impulse magnitude j for two objects a and b is then given by:

$$j = \frac{-(1+\epsilon)\mathbf{v_{rel}}}{\frac{1}{m_a} + \frac{1}{m_b} + \frac{(\mathbf{r_a} \times \hat{\mathbf{n}})^2}{I_a} + \frac{(\mathbf{r_b} \times \hat{\mathbf{n}})^2}{I_b}} \tag{11}$$

This equation was derived from equation 8-18 in [4], with simplifications due to our 2D setting. Specifically, the torque terms in the denominator simplify as follows. Let $S = \mathbf{r} \times \hat{\mathbf{n}}$. Then

$$\begin{aligned}
\hat{\mathbf{n}} \cdot \left(\frac{S}{I} \times \mathbf{r} \right) &= \hat{\mathbf{n}} \cdot \left(\frac{S}{I} \cdot \begin{bmatrix} -\mathbf{r}_y \\ \mathbf{r}_x \end{bmatrix} \right) \\
&= \frac{S}{I} \cdot \left(\hat{\mathbf{n}} \cdot \begin{bmatrix} -\mathbf{r}_y \\ \mathbf{r}_x \end{bmatrix} \right) \\
&= \frac{S}{I} \cdot (-\hat{\mathbf{n}}_x \mathbf{r}_y + \hat{\mathbf{n}}_y \mathbf{r}_x) \\
&= \frac{S}{I} \cdot (\mathbf{r} \times \hat{\mathbf{n}}) = \frac{S^2}{I}
\end{aligned}$$

After we have obtained impulse magnitude j, we can compute the final impulse as $\mathbf{J} = \hat{\mathbf{n}} j$. To apply the impulse to the objects, we apply a force at the collision vertex position: $\mathbf{J}/dt$ for the vertex object, $-\mathbf{J}/dt$ for the edge object.

For object-wall collisions, we set $m_{wall} = \infty$ and $I_{wall} = \infty$, as the wall is immovable and non-rotating. This causes the two terms for the wall to disappear in the denominator of Equation 11, making the impulse depend only on the colliding rectangle.

Initially, we also backtracked the vertices to the point of contact using bisection before computing and applying the impulse. Doing this leads to a more accurate collision response, especially if objects are penetrating at very high speeds. We noticed however that leaving this backtracking out did not worsen any visible quality in our simulation. This is also due to the fact that objects in our simulation do not collide at high velocities. To simplify our logic and to save computation, we decided to leave backtracking out.

2.6 Two-way Coupling

We implemented two-way coupling between rigid bodies and the fluid. This means that the fluid can also exert forces on the objects in the fluid. To compute the resulting forces, we make use of the pressure field that is computed during the projection step.

To calculate the forces that need to be applied to an object, we iterate over all faces of its boundary grid cells. The resulting force f at a single face between a cell i inside the object and cell o outside is computed based on the face normal n, face length $h = 1/N$, the pressure at the center of the face $\rho = 0.5 \cdot (\rho_i + \rho_o)$ and a coupling strength constant δ as follows:

$$f = -\rho \cdot n \cdot h \cdot \delta$$

This force is applied to the object at the boundary cell for which it is computed. Applying these forces for each boundary face of a rigid body results in a realistic change in overall force and torque according to the fluid flow around the object.

Note that this approach does not take into account viscous drag forces that would occur in real life, when fluid flows tangential to an objects surface without exerting pressure on it.

2.7 Temperature

We implemented the simulation of different water temperatures and buoyancy forces resulting from the water temperature.

The temperature is modeled as a scalar field t. Similar to the density field, it changes based on external changes to the temperature, diffusion and advection along the velocity field. It has a separate diffusion coefficient. This means that the rate at which temperature diffuses through the fluid is independent of the diffusion rate of density and velocity.

The boundary conditions for solid objects are implemented in the same way as those for density. This means that the boundary cells of solid objects will be set to the same temperature as the neighboring fluid cells to prevent any diffusion of temperature into or out of the objects. The boundary conditions for the grid walls are implemented differently: The boundary cell temperatures are set to constant values, which can differ between cells. This enables us to have hot spots on the boundaries, where temperature is diffused into the fluid and cold areas, where temperature is diffused out cooling down fluid next to the border.

For each fluid cell c, we apply a buoyancy force pointing straight up that scales linearly with the temperature t_c and can be controlled by a constant coefficient α.

$$f_c = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \cdot t_c \cdot \alpha$$

This buoyancy force results in warm water flowing upwards while pushing cold water downwards. Figure 3 shows how we visualize the temperature with hue while showing the density with the brightness of the rendered cells.

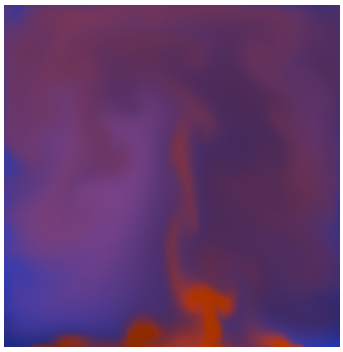


Fig. 3: Visualization of temperature and density.

2.8 Linear Solver

One issue we encountered with our Gauss-Seidel linear solver was that it required a large number of iterations to prevent the simulation from becoming unstable. This was particularly evident during the projection step, where the velocity field is computed. We often needed more than 30 iterations for acceptable results. To address this, we tried switching to a Conjugate Gradient solver, which has better convergence properties [2]. Thus we expected it to reduce the required number of iterations and therefore potentially improve performance.

The problem we aim to solve is a linear system of unknown x, given an initial $x^{(0)}$:

$$x_{i,j}^{(0)} = c \cdot x_{i,j} - a\left(x_{i-1,j} + x_{i+1,j} + x_{i,j-1} + x_{i,j+1}\right)$$

The Gauss-Seidel method approximates the solution by iteratively updating x as follows:

$$x_{i,j} = \frac{x_{i,j}^{(0)} + a\left(x_{i-1,j} + x_{i+1,j} + x_{i,j-1} + x_{i,j+1}\right)}{c}$$

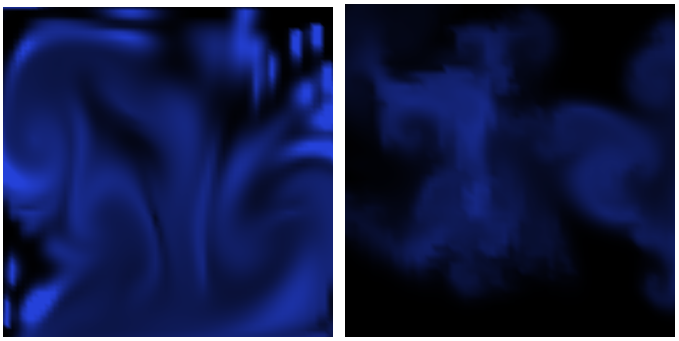
This system can also be written in matrix form as $\mathbf{A}x = x^{(0)}$, where $\mathbf{A}$ is a sparse matrix with c on the diagonal, and four off-diagonals with $-a$. In practise, we don't explicitly define this matrix in code. Rather, we've implemented an *implicit matrix* interface with a method that computes $\mathbf{A}x$, for a given vector x. The Conjugate Gradient solver we use is the one provided in Project 1.

Unfortunately, we never managed to get this solver working correctly, even though we implemented everything as mentioned. The solver consistently blows up to infinity, due to a bug that remains unresolved.

3 Simulation Scenes

In this section, we showcase some scenes which demonstrate features of our simulation.

In figure Figure 4, a comparison is made between vorticity confinement disabled and enabled. Note that no small scale swirls can be seen in Figure 4a, while Figure 4b does show this off.



(a) Without confinement (b) With confinement

Fig. 4: Comparison between none vs. high vorticity confinement.

We have implemented a direction-encoded coloring of the velocity field, which can be seen in Figure 5. North is encoded with red, east with yellow, south with green, and west with blue. The velocity field also shows the vorticity confinement in action, note the different small vortices that occur in the field.

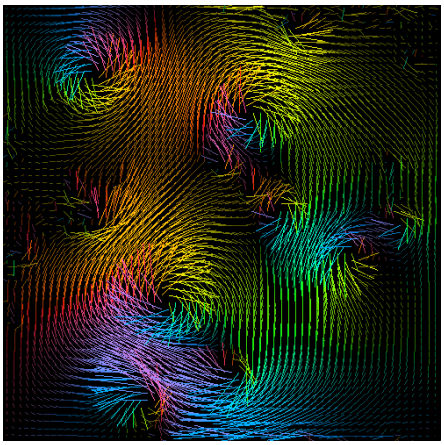


Fig. 5: Direction-colored velocity field

Figure 6 shows off the result of fluid colliding with a square. Note how most of the fluid is pushed to the side, while only a small part moves further downward along the square.

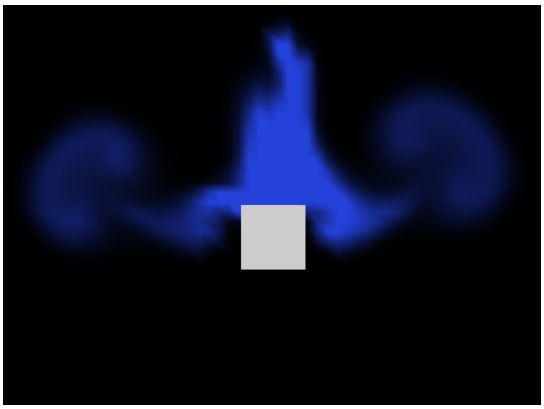


Fig. 6: Fluid pushed downward on a square

Figure 7 shows the result of a fluid-object collision the other way around, where a rectangle was pushed upward against fluid. Note the two swirls resulting from the interaction.

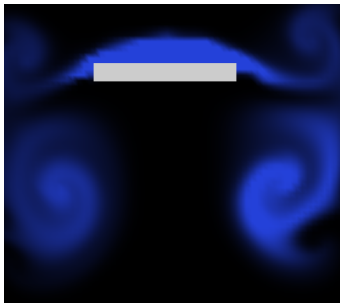


Fig. 7: Rectangle pushed upwards against fluid

When putting all features of our simulation together, we get to our final scene shown below in figure Figure 8. Hot spots along the bottom boundary causes the water to heat up and rise, before diffusing some heat along the other borders and sinking again. This results in constant swirl like movements in the water that move the rigid bodies (two-way coupling). This movement causes them to collide with each other and the border.

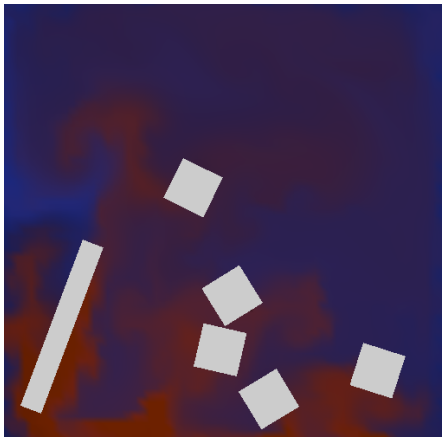


Fig. 8: All features put together in one scene

4 Conclusion

In this project, we developed a fluid simulation system capable of capturing realistic interactions between fluid and rigid bodies. Our additions include vorticity confinement, solid and movable obstacles, collidable rigid bodies with rotational dynamics, two-way coupling between fluids and obstacles, and temperature-dependent buoyancy effects. Although we encountered some performance challenges, in the end we managed to achieve stable and visually convincing simulations. Future improvements could target enhancing performance by implementing a more efficient linear solver, and incorporating a particle system in the simulation. With the implemented features and performance optimizations, our system serves as a robust platform for future development and experimentation.

APPENDIX A
INDIVIDUAL CONTRIBUTIONS

Note: During this group project all team members participated very actively and with similar effort and with high collaboration on almost all task. It is therefore difficult to differentiate which part of the work should be contributed to whom. In the list below the features and tasks are listed below the team members who worked most actively on them but it does not mean that they were only implemented by a single person. For most of the features all team members were actively involved in either conception, implementation, bugfixing or optimization.

Magnus Menger:

1) Vorticity confinement
2) Fixed objects
3) Two-way coupling
4) Temperature
5) Conjugate gradient solver

Melvin Hoogendoorn:

1) Vorticity confinement
2) Rigid bodies
3) Collisions
4) Conjugate gradient solver

Viljan van Raaj:

1) Vorticity confinement
2) Moving solid objects
3) Performance optimization
4) Conjugate gradient solver
5) Demo video

APPENDIX B
KNOWN ERRORS IN CODE

B.1 Vorticity Confinement

In our implementation of the vorticity confinement a small error in the formulas resulted in us having to use large ϵ values to get a visible result. This was simply due to us dividing Equation 1, Equation 2 and Equation 3 by the grid size $N = \frac{1}{h}$, instead of multiplying with $N = \frac{1}{h}$.

REFERENCES

[1] J. Stam, "solver code." [Online]. Available: http://www.dgp.toronto.edu/people/stam/reality/Research/zip/CDROM_GDC03.zip
[2] ——, "Real-time fluid dynamics for games," Proceedings of the Game Developer Conference, March 2003.
[3] R. Fedkiw, J. Stam, and H. W. Jensen, "Visual simulation of smoke," in *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '01. New York, NY, USA: Association for Computing Machinery, 2001, p. 15–22. [Online]. Available: https://doi.org/10.1145/383259.383260
[4] D. Baraff and Pixar Animation Studios, "Physically based modeling," SIGGRAPH '99 Course Notes. [Online]. Available: https://graphics.stanford.edu/courses/cs448b-00-winter/papers/phys_model.pdf