Bubbles Simulation in Blender and OpenGL

Emily Pedersen
University of California, Berkeley
Berkeley, CA, USA
epedersen@berkeley.edu

George Zhang
University of California, Berkeley
Berkeley, CA, USA
george_zhang@berkeley.edu

Isabella Chavira
University of California, Berkeley
Berkeley, CA, USA
ichavira@berkeley.edu

ABSTRACT

The beauty of soap bubbles has a timeless appeal, captivating people of all ages. Alongside children's fascination with bubbles, artists, such as Chardin and Monet, have been amazed, capturing the beauty of soap bubbles through the ages [4]. Maybe the soap bubbles' complex physical structures motivated the artists into attempting to replicate their beauty. Soap bubbles are created when air is trapped inside soap films, sandwiching a thin layer of water between two layers of film [6]. Soap bubbles have a shifting iridescence, and perfect geometries [4]. Soap bubbles' complicated anatomies makes them challenging to simulate in computer graphics. Researchers have developed complicated systems to model soap bubbles, but we provide a more straight forward framework. We wanted to model bubbles as we saw them, not focusing on making the simulations physically correct, but plausible and pleasing to the eye. We accomplished this through building bubbles simulations in Blender and in OpenGL. In Blender, we modelled a bubble popping, and in OpenGL we modelled bubbles floating in the air, as well as wobbling. In terms of our technical contribution, we implemented a method to interpolate keyframes to simulate a bubble wobbling through the air.

KEYWORDS

Bubble simulation, Blender, OpenGL, popping bubbles, physics of bubbles, keyframe interpolation

ACM Reference Format:
Emily Pedersen, George Zhang, and Isabella Chavira. 2019. Bubbles Simulation in Blender and OpenGL. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

For this project we created a bubble simulator using both Blender and OpenGL. We chose bubbles as the subject of our simulation because we found them to be a beautiful, yet complex physical phenomena. We were especially interested in the rainbow effect caused by the soapiness of the bubble, and also the physics of a bubble's shape and how this changes given an environment. The interesting physical aspects of bubbles led to us wanting to create an interactive bubble simulation, in which a player can move the bubble around an environment, and interact with other objects to pop the bubble.

We wanted our bubble model to be as realistic as possible, so we researched the physics behind a bubble's geometry, topology, and coloring. We knew that creating a realistic bubble model alone would be a difficult task, so we used the Navier-Stokes and Thin-film interference equations to guide us in creating a realistic bubble model. The Navier-Stokes equation describes the motion of a viscous fluid substance [9]:

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\frac{1}{\rho}\nabla p + \gamma \nabla^2 \mathbf{u} + \frac{1}{\rho}\mathbf{F} \tag{1}$$

$$\nabla \mathbf{u} = 0 \tag{2}$$

In the above equations, u is the velocity field, ρ is density, p is pressure, γ is viscosity, and F represents any additional forces, such as gravity, buoyancy, and surface tension.

Thin-film interference is a natural phenomenon, in which light waves that interfere with each other, either enhance or reduce the reflected light. In the case of soap bubbles, light travels through the air and hits the soap bubble's film. Air has a refractive index of 1, and film has a refractive index larger than 1. The conditions for interference with soap bubbles are [11]:

$$2 * n_{film} * d * \cos\theta_2 = (m - \frac{1}{2}) * \lambda \tag{3}$$

$$2 * n_{film} * d * \cos\theta_2 = m * \lambda \tag{4}$$

(3) Refers to the constructive interference of reflected light.
(4) Refers destructive interference of reflected light.
d is the film thickness, n_{film} is the refractive index of the film, θ_2 is the angle of incidence of the wave on the lower boundary, m is a constant, and λ is the wavelength of light.

We did not directly implement these equations, but used them to influence how we modelled our bubbles through GLSL shaders, and GLSL reflect and refract functions as we will discuss in the Implementation and Results sections.

2 RELATED WORKS

We rely on one major area of prior research: modeling soap bubbles in computer graphics.

2.1 Modeling Soap Bubbles

2.1.1 Popping Soap Bubbles. Two UC Berkeley researchers mathematically described the successive stages of the disappearance of foamy bubbles [8]. Their work has applications in the mixing of foams, industrial processes for making plastic foams, and modeling cell cluster growth. Their techniques relied on a set of linked partial differential equations that tracked the motion of connected interfaces, where physics and chemistry define the surface dynamics. Their novel framework was a scale-separated approach that

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

identified the physics happening in each distinct scale, which are then coupled in a consistent manner Specifically, they developed four sets of equations:

(1) Described the gravitational drain of liquid from the bubble walls
(2) Dealt with the flow of liquid inside the junctions between bubble membranes
(3) Dealt with wobbly rearrangement of bubbles after one bubble pops
(4) Solved the physics of a sunset reflected in the bubbles, creating rainbow hues similar to slick on wet pavement. Solving these four equations took 5 days at the LBNL's national energy research scientific computing center (NERSC)

Solving these four equations took 5 days at the LBNL's national energy research scientific computing center (NERSC).

2.1.2 Modeling the Light Inference in Soap Bubbles. Researchers from the University of Tokyo proposed a fast rendering method for soap bubbles considering light interference and dynamics [5]. The reflectivities of the thin film are calculated in advance and stored as texture, which made it possible to render soap bubbles in real time.

2.1.3 Animation of Air Bubbles with Smoothed Particles Hydrodynamics. Three researchers from the University of Freiburg, Germany developed a new physically-based multiphase model to simulate air and water bubbles [7]. Their new model takes into account the physical phenomena such as drag and buoyancy.

All of these research findings are advanced, so we used them as guides us as we built our bubble models in Blender and OpenGL.

3 TECHNICAL CONTRIBUTIONS

During our project, we discovered that translating animations from Blender into OpenGL is difficult, especially when the animation destroys the object's mesh such as bubble popping. We still wanted to have an animation in OpenGL, so we investigated how to make the soap bubbles looks like they are wobbling. In terms of modelling a natural phenomenon, large bubbles wobble because their internal pressure and surface tensions are inadequate to keep the sphericity of large bubbles [2]. To simulate bubbles wobbling, we interpolated between key frames to make the animation smooth. In our research, we didn't see much prior research on this topic, keyframe interpolation in OpenGL, so we present our solution as our technical contribution.

3.1 Interpolation Between Key Frames

We noticed that Blender would automatically interpolate between key frames to give smooth animation. We wanted to achieve this result in C++ OpenGL. We deformed our original sphere bubble mesh in Blender, giving us two models: bubble as sphere, and bubble as deformed (to produce the wobbly effect) shown in Figure 8 and Figure 9, respectively. We created a LerpEntity class for Entitys that have two models to interpolate between. When constructing a LerpEntity, we use our OBJLoader class to load both models and store their respective data (position coordinates, texture coordinates, normal vectors, and indices). Assuming that indices will be the same between the two models, we disregard the indices. Also because we are using a skybox, we can neglect the texture coordinates. Thus, we need to linearly interpolate between position coordinates and normal vectors. We can find the differential amount to update each position coordinate and normal vector by dividing the difference between the initial and final values by m_NumSteps, the number of interpolation steps needed to transition between the two models. The larger m_NumSteps is, the smoother the transition.

4 IMPLEMENTATION

4.1 Modeling Soap Bubbles in Blender

We first started by creating a bubble model in Blender. In order to create this we decided to go with a method similar to how Bui Tuong Phong created his shading model. In this way, we created an empirical model using observations rather than mathematical equations and relationships. Using the node editor in blender we were able to aggregate these factors to create a realistic bubble appearance. Specifically, we put together transparency, glossy specular lighting, and rainbow color effects, making a realistic looking bubble. In addition to the lighting effects of bubbles, we also noticed that bubbles have a slight thickness. In order to implement this, we created two UV spheres; one on the outside with its normal vectors pointing, and another on the inside with its normal vectors pointing inwards. The final step to creating our bubble model was to give the bubble a slight 'wobbly' effect. Bubbles are not perfect spheres, so in order to model this, we used the displacement modifier on our bubbles model, and tweaked the parameters to slightly offset the outer vertices of the bubble.

4.1.1 Animating the Bubble Model. We not only wanted to recreate the physical appearance of bubbles, but also how bubbles interact with their surrounding environments. This led us to be interested in simulating bubbles popping. In the physical world, bubbles pop for many reasons. Evaporation of its water contents, air turbulence, and most commonly coming in contact with a dry surface or dry air [12]. Bubbles can also pop if punctured by a sharp, dry object [12]. To simulate a bubble popping, we wanted to animate it in OpenGL, which led to us again recreating the simulation based on observation. We watched several videos of bubbles popping in slow motion, each with varying angles and popping mechanisms. Using slow motion videos to observe the bubble popping made it much easier to recognize the key frames needed to animate and make the popping look realistic. From these videos we decided to mimic the bubble popping as if it were stabbed by a thin object. We created several key frames within Blender and then used interpolation to create the moments in between these key frames.

Our original intention was to create the popping animation in blender and then import that into OpenGL and have it occur on the event of a collision. However, this proved to be extremely difficult, as animating in OpenGL is the equivalent of destroying the given mesh object and replacing it for each frame of animation.

4.2 Modeling Soap Bubbles in OpenGL

We implemented a barebones OpenGL rendering pipeline with the following functionalities: (1) .obj 3D model loading, (2) image texture loading, (3) Blinn-Phong shading, (4) keyboard control, (5) third-person camera, (6) cubemap loading, (7) Fresnel reflections

and refractions, (8) chromatic aberration, and (9) interpolation between keyframes. (7) and (8) are discussed in the Results section, and (9) is discussed in the Technical Contributions section. We referred to Java and C++ tutorials by ThinMatrix [11] and TheChernoProject [10] for an idea of how to organize our OpenGL implementation into classes.

4.2.1 Loading .obj Models. In general, .obj files are organized as follows:

v	$x_1\ y_1\ z_1$		(1)
v	$x_2\ y_2\ z_2$		(2)
v	$x_3\ y_3\ z_3$		(3)
...			
v	$x_M\ y_M\ z_M$		(M)
vt	$u_1\ v_1$		(1)
vt	$u_2\ v_2$		(2)
vt	$u_3\ v_3$		(3)
...			
vt	$u_N\ v_N$		(N)
vn	$x_1\ y_1\ z_1$		(1)
vn	$x_2\ y_2\ z_2$		(2)
vn	$x_3\ y_3\ z_3$		(3)
...			
vn	$x_P\ y_P\ z_P$		(P)
...			
f	$v_a/vt_a/vn_a$	$v_b/vt_b/vn_b$	$v_c/vt_c/vn_c$
f	$v_d/vt_d/vn_d$	$v_e/vt_e/vn_e$	$v_f/vt_f/vn_f$
f	$v_g/vt_g/vn_g$	$v_h/vt_h/vn_h$	$v_i/vt_i/vn_i$

The first M rows starting with v contain the (x, y, z) position coordinates of vertices. The N rows starting with vt contain the (u, v) texture coordinates of vertices. The P rows starting with v_n contain the (x, y, z) normal vectors of vertices. Note that the indexing of these variables starts at 1 instead of 0, which is standard for .obj model files exported from Blender. The rows starting with f tells us which vertices each position coordinates, texture coordinates, and normal vector corresponds to. For example, for $v_a/v_{ta}/v_{na}$, $v_a \in [1, M]$, $v_{ta} \in [1, N]$, $v_{na} \in [1, P]$, which indicates that there is a vertex with the v_a^{th} position coordinates, v_{ta}^{th} texture coordinates, and v_{na}^{th} normal vector. Note that the f row contains information for three vertices, which form a triangle. The model's mesh is divided into triangle faces, and our OpenGL rendering implementation draws these triangles when displaying the model to the screen. Note that the .obj models were exported from Blender with the setting "Triangulate Faces" selected. We referred to TheChernoProject's YouTube tutorials [10] in implementing an OBJLoader class that string parses the .obj file and stores its data into four arrays: positions, texCoords, normals, and indices.

The data in positions, texCoords, and normal are stored in Vertex Buffer Objects (VBOs). The data in indices is stored in an Index Buffer (IB), a special type of VBO. These VBOs (and IB) are added to a Vertex Array Object (VAO). First, a VAO must be created, then binded. Then, one by one, each VBO is binded, added to the bound VAO, and unbinded (optional). Finally, the VAO is unbinded (optional). Binding a VAO then calling glDrawElements(...) will render the data in the bound VAO.

4.2.2 Textures. Our Texture class serves three main purposes: (1) loading the image texture file (.jpg, .png, etc...), (2) allowing binding and unbinding, and (3) defining ambient, diffuse, and specular reflection constants and a shininess constant and loading them into shaders as uniforms (to be discussed in 3.2 Shaders). For loading the image texture file, we used the stb_image API [1]. Binding a texture (into a slot, e.g. GL_TEXTURE0, GL_TEXTURE1, GL_TEXTURE2, ...) and loading an integer representing the slot (e.g. 0, 1, 2, ...) into the uniform sampler2D allows us to render that texture. Unbinding that texture allows us to bind another texture for rendering. In our Texture class, we also define ambient, diffuse, and specular reflection constants and a shininess constant. These variables can be loaded as uniform variables into a shader to influence the Blinn-Phong shading of the model. An example of a Blinn-Phong shading model is shown in Figure 1.

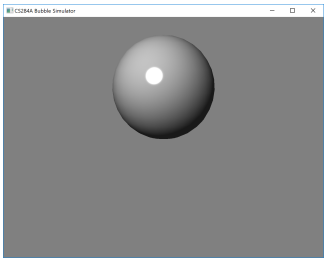

Figure 1: Sphere (Blinn-Phong shading).

4.2.3 Shaders. Our Shader class serves three main purposes: (1) compiling the GLSL .shader files, (2) allowing binding and unbinding, and (3) loading uniforms to the GLSL .shader files. Our GLSL .shader files are located in the res/shaders folder. We only used vertex and fragment shaders. A vertex shader is called once per vertex. The inputs to a vertex shader come from the VBOs in the bound VAO. In the vertex shader, we define gl_Position, which determines where on the screen the vertex is rendered. Outputs of the vertex shader are inputs to the fragment shader, which is called once per pixel inside the triangle defined by three vertices. In the fragment shader, we define gl_FragColor, which determines the color of the pixel. Interpolation between the three vertices is done to determine the colors of pixels within the triangle.

4.2.4 Keyboard Control. In the open() method of our DisplayManager class, we call glfwSetKeyCallback(...), glfwSetCursorPosCallback(...), glfwSetMouseButtonCallBack(...), and glfwSetScrollCallBack(...) to activate responses for keyboard keys, mouse movements, clicks, and scrolls. In DisplayManager's finishLoop() method, we call glfwPollEvents(), which polls for these events. In this method, we also calculate and store the time taken for each loop in m_DeltaTime. This time is useful for actions that should depend on real time rather than frames per second. For instance, player movement speed should be uniform across all users regardless of how fast their machine can run through frames.

4.2.5 Cubemaps. A cubemap (or skybox or environment map) is essentially six images that are stitched together to form a cube. One side of the cubemap is shown in Figure 2. Viewing inside this

cubemap gives the illusion that there is a smooth, continuous environment. For loading cubemaps, we referred to Learn OpenGL [3]. Again, we use the stb_image API to load all six images. When calling glTexImage2D(...), we input GL_TEXTURE_CUBE_MAP_POSITIVE_X $+i$, where $i \in [0, 5]$ for each face of the cubemap. This cubemap has its own VAO, which only has one VBO listing the position coordinates of a cube. The cubemap also has its own shader, which is incredibly simple. In its vertex shader, gl_Position is set to projection matrix * view matrix $*$ input position (Note that there is no model matrix in this case, since the sky box won't be translated, rotated, or scaled). In its fragment shader, gl_FragColor is set to texture(u_SkyBox, position), where u_SkyBox is a samplerCube and position is just the input position to the vertex shader.

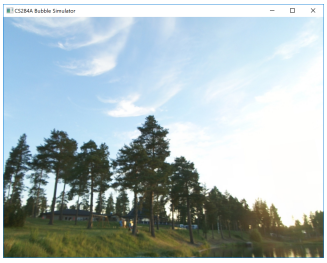

Figure 2: One side of the Lycksele cubemap

5 RESULTS

5.1 Blender Models

5.1.1 Modeling the Bubble. In order to make our bubble model as realistic as possible we knew that there were several appearance factors that we would need to recreate. The first was the rainbow lighting effect caused by the soapiness of the bubble material. Next, we also wanted to make sure that the bubble appeared to be transparent, and include glossy specular lighting. We aggregated all of these effects to create our bubble material and placed this material onto a sphere with its vertices slightly displaced to give the model a slight wobble.

Figure 3: The completed Blender bubble model

Another important aspect of our bubble model was the thickness of the soapy film of a bubble. We noticed from closely observing several pictures of real life bubbles that it is apparent that there is a thickness to the walls of a bubble. We created two transparent spheres, one that acts as the outside wall of the bubble and the other on the acting as the inside wall.

Figure 4: Zoomed in photo of the bubble model to highlight the thickness

5.1.2 Popping Animation. In order to animate a bubble popping in blender, we created key frames based off of watching several videos of bubbles popping in slow motion. We decided that the key frames to be modeled were the initial impact of the popping object, the bubble halfway through its pop, and the end of the pop, making sure to show that the dispersing of the bubble continues for a few frames after the bubble has fully popped. These key frame models are displayed below.

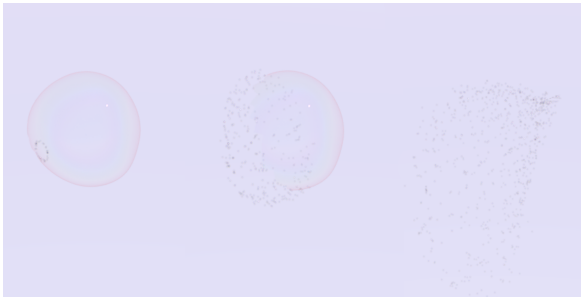

Figure 5: Key frames of bubble popping

5.2 OpenGL Bubble Models

5.2.1 Fresnel Reflections and Refractions and Chromatic Aberrations. To simulate Fresnel reflections and refractions, we used GLSL reflect(...) and refract(...) functions, and mixed the two results by a ratio of Schlick's approximation. However, this resulted in a glassy look rather than a bubble look, which has chromatic aberration. To simulate a simple chromatic aberration, we set slightly different indices of refraction for red, green, and blue light (we set the ratio of indices of refraction for the two mediums as $\eta_R = 0.65$, $\eta_G = 0.67$, and $\eta_B = 0.69$). We also mixed this result with Blinn-Phong shading to give a more realistic shine on the bubble. The results are shown in Figure 6 and Figure 7.

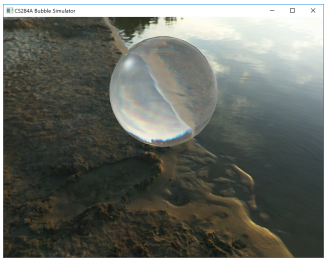

Figure 6: Bubble facing ground (Fresnel reflections/refractions and chromatic aberration).

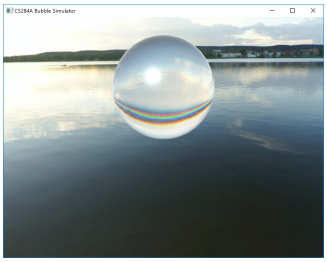

Figure 7: Bubble facing horizon (Fresnel reflections/refractions and chromatic aberration).

5.2.2 Modeling Wobbly Bubbles. Figures 8 and 9 are the rendered results from before and after interpolating between key frames to produce a wobbly bubble effect, respectively.

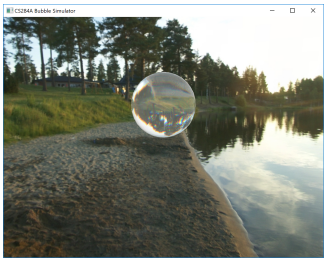

Figure 8: Bubble as sphere

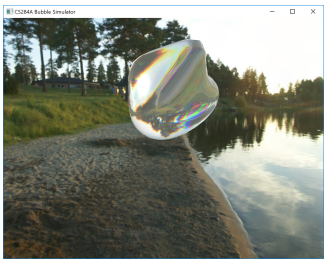

Figure 9: Bubble as wobbly

6 FUTURE WORK

In the future we are looking to create a model using the mathematical relationships pertaining to bubbles, or perhaps a combination of this and observations. In this way we would be able to more accurately simulate the physical properties of bubbles. While we were able to make the lighting and shape of the bubbles look semi-realistic in our empirical model, the use of mathematical equations could further improve this realism. Additionally, using these relationships would greatly aid us in simulating a bubble's interaction with its environment. We were able to create a realistic popping animation for a bubble, however, if we were to have the proper equations that make up a bubble, we could make our simulation as realistic as possible. Furthermore, this would allow us to be able to simulate more physical phenomena surrounding bubbles, like bubble merging. Given the mathematical and physical makeup of soap bubbles, we would also be able to simulate how bubbles interact with each other.

7 CONCLUSION

Due to soap bubbles' beauty, we wanted to model them for our final project. However, soap bubbles have a complicated structure and physical properties, making it a challenge to accurately model. We used the Navier-Stokes and the thin-film interference equations as guides and empirically modelled bubbles in Blender and OpenGL. In Blender, we built a realistic looking bubble through adding transparency, glossy specular lighting, and rainbow color effects. We added a second UV Sphere on top of our original UV sphere to create the effect of a bubble's thickness. We used the displacement modifier to give our bubble model a wobbly effect. In Blender, we also created several key frames to mimic a bubble popping. We also implemented a barebones OpenGL rendering pipeline that can (1) load .obj models, (2) load image textures, (3) render Blinn-Phong shading, (4) control the model through keyboard controls, (5) load a cubemap, (6) navigate through the cubemap with a third-person camera, (7) render Fresnel reflections and refractions, (8) render chromatic aberrations, and (9) interpolate between key frames. (3), (7) and (8) were instrumental in imitating the rainbow effect of a soap bubble's film. (9), interpolating between key frames in OpenGL, is our project's novel contribution.

ACKNOWLEDGMENTS

To get started with OpenGL, we watched YouTube tutorials by ThinMatrix and TheChernoProject to learn about the OpenGL pipeline. The tutorials were in Java, and we translated it into C++. ThinMatrix's tutorials can be found here: https://www.youtube.com/watch?v=VS8wlS9hF8E&list=PLRIWtICgwaX0u7Rf9zkZhLoLuZVfUksDP TheChernoProject's tutorials can be found here: https://www.youtube.com/watch?v=W3gAzLwfIP0&list=PLlrATfBNZ98foTJPJ_Ev03o2oq3-GGOS2

We'd also like to thank Professor Ren Ng, and Professor Jonathan Ragan-Kelley for teaching such an interesting and fun class. Finally, we'd like to thank the TA staff for all your support during the semester.

REFERENCES

[1] Sean Barrett. *nothings/stb*. GitHub. 2019. URL: https://github.com/nothings/stb.
[2] *Bubble Optics*. Website. Last accessed 14 May 2019. OPOD. URL: https://www.atoptics.co.uk/fz618.htm.

[3] *Cubemaps*. Website. Last accessed 14 May 2019. LearnOpenGL. URL: https://learnopengl.com/Advanced-OpenGL/Cubemaps.

[4] Cyril Isenberg. *The Science of Soap Films and Soap Bubbles*. Website. Last accessed 14 May 2019. URL: https://books.google.com/books?hl=en&lr=&id=PdsVME_LXTYC&oi=fnd&pg=PA1&dq=soap+bubbles+in+air+&ots=p0nVgW54Aj&sig=ixPBvKq3mHVJ8uTrGXOdE8jEDTY#v=onepage&q=soap%20bubbles%20in%20air&f=false.

[5] Keichi Matsuzawa Kei Iwasaki and Tomoyuki Nishita. *Real-time Rendering of Soap Bubbles Taking into Account Light Interference*. Website. Last accessed 14 May 2019. University of Tokyo. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.81.9952&rep=rep1&type=pdf.

[6] David Devraj Kumar. *SOAP BUBBLES: NOT JUST KIDSfffdfffdfffd STUFF!* Website. Last accessed 14 May 2019. URL: http://www.theaic.org/pub_thechemist_journals/Vol-88-No-2/Vol-88-No2-Article-8.pdf.

[7] Gizem Akinci Markus Ihmsen Julian Bader and Matthias Teschner. *ANIMATION OF AIR BUBBLES WITH SPH*. Website. Last accessed 14 May 2019. University of Freiburg, Germany. URL: https://pdfs.semanticscholar.org/43ff/7c78cbd323c8829e97bc44bee83e3e8e3fa9.pdf.

[8] Robert Sanders. *Heady mathematics: Describing popping bubbles in a foam*. Website. Last accessed 13 May 2019. May 2013. URL: https://news.berkeley.edu/2013/05/09/heady-mathematics-describing-popping-bubbles-in-a-foam/.

[9] Jos Stam. *Stable Fluids*. Website. Last accessed 14 May 2019. URL: http://www.dgp.toronto.edu/people/stam/reality/Research/pdf/ns.pdf.

[10] TheChernoProject. *Welcome to OpenGL*. Youtube. 2017. URL: https://www.youtube.com/watch?v=W3gAzLwfIP0&list=PLlrATfBNZ98foTJPJ_Ev03o2oq3-GGOS2.

[11] *Thin Film Interference*. Website. Last accessed 14 May 2019. Lumen Physics. URL: https://courses.lumenlearning.com/physics/chapter/27-7-thin-film-interference/.

[12] *Why do bubbles pop?* Website. Last accessed 14 May 2019. Bubbles.org. URL: http://www.bubbles.org/html/questions/pop.htm.