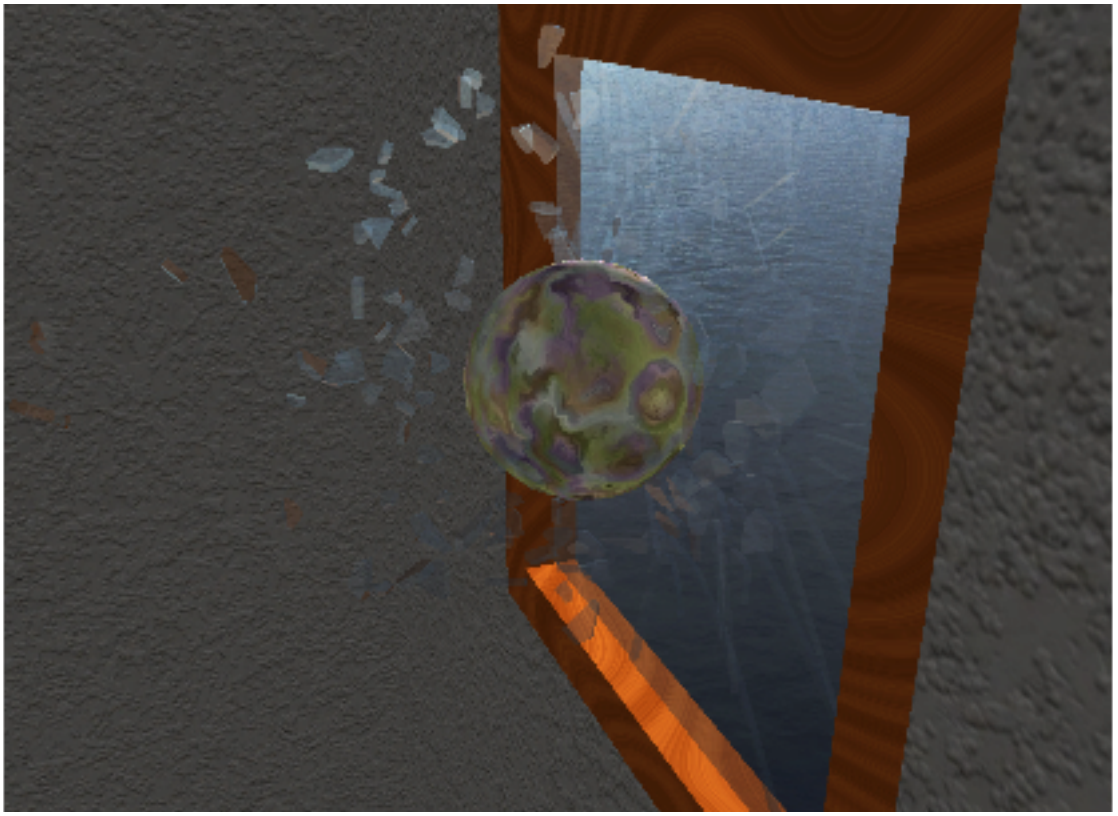
Computer Graphics for Games 2019

Shattered Glass

Procedural generation of glass shards according to a point of impact, with accurate physics behaviour and fresnel reflection/refraction



Afonso Vieira
86458
MEIC
jose.a.vieira@tecnico.ulisboa.pt

João Patrício
97046
MEIC
joaommpatricio@tecnico.ulisboa.pt

Jorge Nunes
87677
MEIC
jorge.costa.nunes@tecnico.ulisboa.pt

Taíssa Ribeiro
86514
MEIC
taissa.ribeiro@tecnico.ulisboa.pt

Abstract

With this project we tried to simulate the breaking of a glass window on impact from a ball. To do so, we implemented the Fortune's Algorithm to split the glass into smaller shards depending on the point of impact, making them different every run. We thought of this algorithm as Voronoi Diagrams are one of the easiest ways to make the appearance of a shattered glass and the Fortune's variant not only because it is one of the fastest ways to achieve the final look (uses O(n log n) time) but it is also memory cheap (O(n) space). Additionally, we integrated a physics engine into our application to simulate how the glass shards would behave when broken from the window upon the collision of the ball and its interaction with the latter. We've also fabricated a glass material to achieve a more realistic look, and developed interesting procedurally generated materials for the ball and window frame.

1. Concept

Initially we thought of replicating what happens in the game *Smash Hit* a game who's co-creator is Dennis Gustafsson, a man (worth mentioning as one of the current intellect in computer graphics) who at the time made the procedurally generated shattering of glass structures when thrown a metallic ball at.

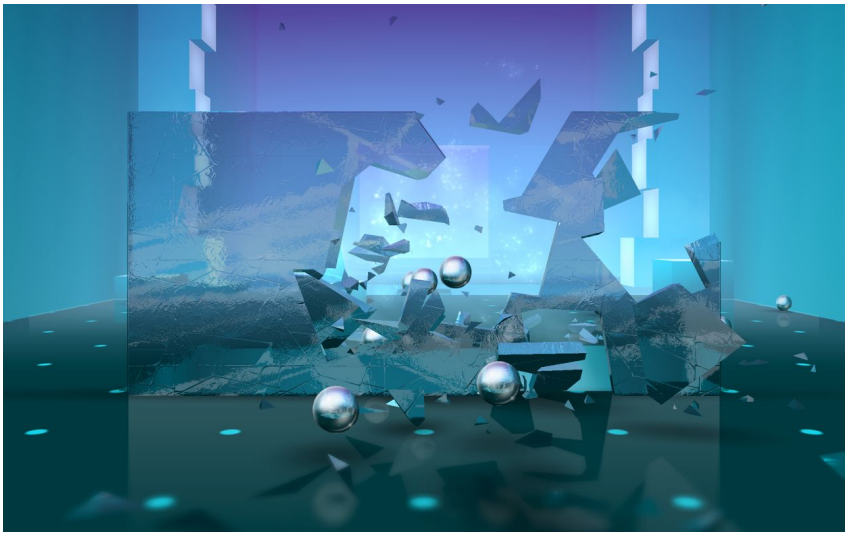


Fig 1 - Smash Hit (Play Store's Snapshot)

The idea was simple but challenging: create an interactive simulation of a ball colliding with a window. Whilst being inside the window's housing, it should be possible to see the collision happening. Following the collision, the window would break into smaller shards/fragments and those would fall, bouncing on the floor along with the ball. In addition, we would try and simulate the blowing of the wind on the curtains while making the scene rendered 100% in real time and being as realistic as possible, taking into account the time given for the project.

For achieving the most true to life appearance, we would try to simulate every material used: the glass, using Fresnel Reflection/Refraction; the wood for the window frame and marble from the ball, making use of noise functions in both of them (simplex and perlin, respectively). Furthermore, the glass shards would be calculated according to a voronoi diagram - a simple but very visually effective way to represent events which happen naturally: from giraffe's spots and dragonfly wings to drylands, leaves' cuticles and (not surprisingly) glass shattering. Moreover, integrating a physics engine in our application would ensure cohesion, providing a smooth and believable scene.

This concept was, indeed, changed throughout the semester.

On one hand, the idea for the curtains was completely removed. We observed that the physics engine we settled on would not replicate objects with a cloth-like behaviour, making it difficult to single-handedly approach this reproduction.

On the other hand, while implementing the shattering, our team came up with the idea to leave the shattered fragments that belonged to the border attached to the wooden frame. Through this, there was a significant addition in realism.

2. Technical Challenges

2.1. **Glass** [Taíssa Ribeiro]

The first technical challenge was making the glass realistic with all it's reflections and refractions using the Fresnel model. One of the main issues was doing the transparencies between each fragment and at the same time making the fresnel model work the right way. The changing environment between the inside and the outside of the room was also a problem as the shards inside the housing would reflect/refract the cube map outside the house, which was not intended.

Figure 2 shows how the glass should look like with the transparencies between the fragments and also the reflections and refractions.

Fig 2 - Example of a shattered glass with some reflections and refractions

2.2. **Wood** [Jorge Nunes]

One of the challenges we faced was making a wood shader with a realistic lighting and look. In order to achieve a similar appearance of what a wooden frame would look like we needed to resort to a noise algorithm that could provide a grainy look and both curly lines and uneven circles, as seen in Figure 3.

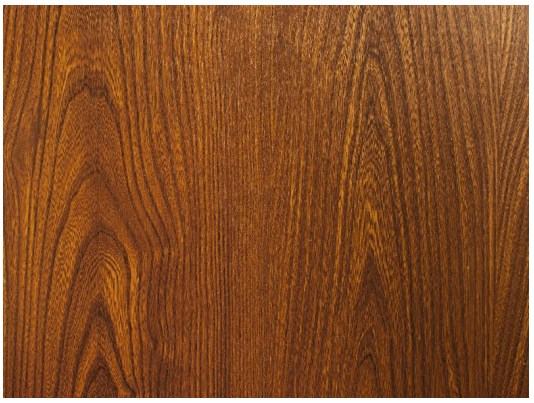


Fig 3 - Example of realistic wood

2.3. **Marble** [Jorge Nunes]

Just as we did for the wood shader, we needed a way to create realistic marble. In this case we needed a noise algorithm which could replicate the abstract grainy look of a marble stone, as seen in Figure 4.

Since marble has such a huge palette of colours, we also wanted to play with a mix of all three RGB colours, while trying to avoid a rainbowy look.

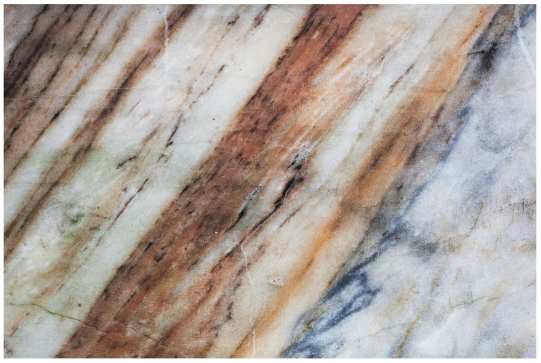


Fig 4 - Example of realistic marble stone

2.4. **Procedurally Generated Glass Shattering** [Afonso Vieira]

Upon some research, we had come up with some issues regarding how to approach the making of the voronoi algorithm. At first glance, it seemed like a reading through how to write the algorithm would be difficult as it required a deep understanding of 3D algebra and calculus. Further on, we learned these problems were tackled slightly differently in computer graphics than from the literature.

At a certain point, we discovered that the algorithm was not calculating the cells on the border, making the application run at a very low frame rate and/or crash.

Also, in the final part of the algorithm, the resulting polygons were not taking into account collinear edges which then made the physics behave poorly.

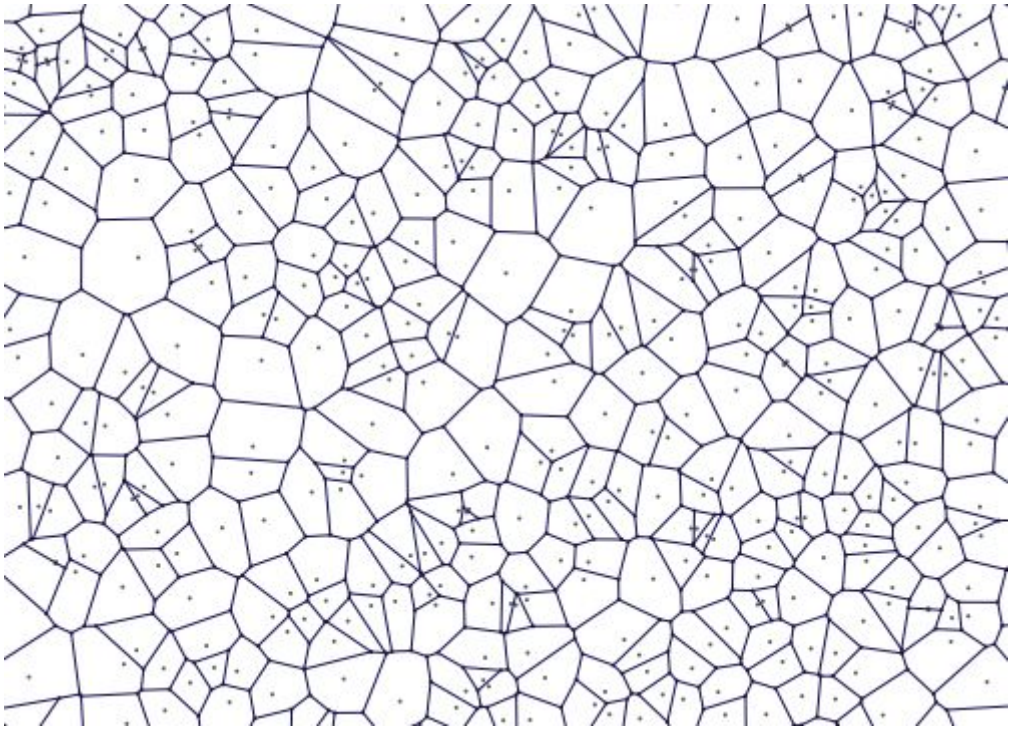


Fig 5 - Example of a Voronoi Diagram

2.5. **Physics Engine Integration** [João Patrício]

While integrating *ReactPhysics3D* (from now on referred to as “rp3d”) into our application, we found several challenges.

The first challenge was synchronizing the scenegraph with the world managed by *rp3d*. This would require that we update the *rp3d* world and update each scenegraph node’s transform to match the physical one.

Next we had to convert the polygon data received from the fragment generation and 3D model files into a format that *rp3d* can understand and work with. This required some refactoring of the way in which meshes are loaded into our applications, since in order for *rp3d* to recognize them as valid they must be “closed” meshes, meaning that the faces must share vertices.

Previously, as it was the simplest approach and also the one provided during the labs, we were simply replicating the shared vertices for each face within memory. This would not work, as rp3d would interpret each face of a mesh as a mesh on it’s own.

When trying to work with concave meshes we were presented with 2 options, we could either use *rp3d*’s ConcaveMeshShape class or try to decompose them into groups of convex meshes, with each approach having its drawbacks.

We also ran into an issue with some objects that rp3d was not calculating the center of mass of the meshes correctly, leading to odd collision errors and positioning of shards in the world.

3. Proposed Solutions

3.1. **Glass**

3.1.1. Explored approaches

When a dielectric material, such as glass, is looked at along a grazing angle, it is more reflective. To accomplish this, one approach was to use a simplification of the Fresnel equation:

$$F = f_\lambda + (1 - f_\lambda)(1 - v \bullet h)^5$$

Fig 6 - Schlick's approximation of the Fresnel formula

In Figure 7, we can see that, depending on the angle, there's more reflection or refraction, even though it's representing water the effect it gives to glass is practically the same.

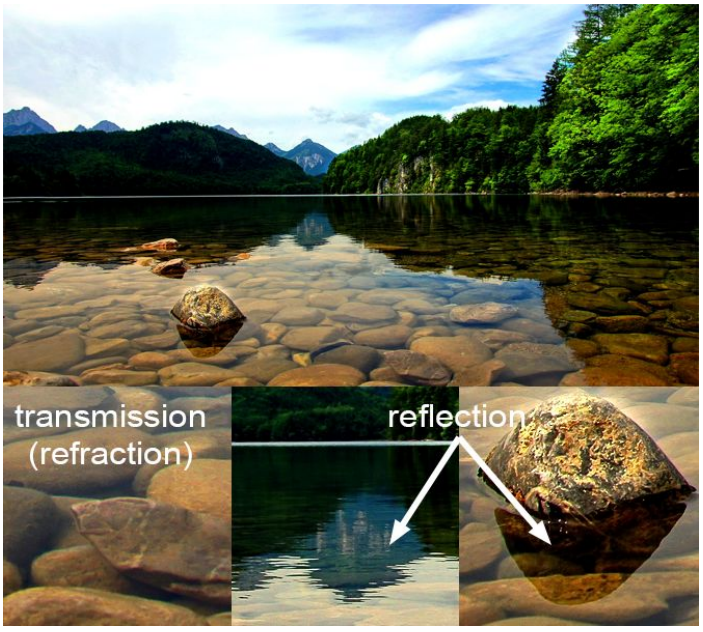


Fig 7 - Fresnel Reflectance

To use the Fresnel model, first we need to assess the environment around the glass, to do that we tried a simple cubemap as we can see in Figure 8.

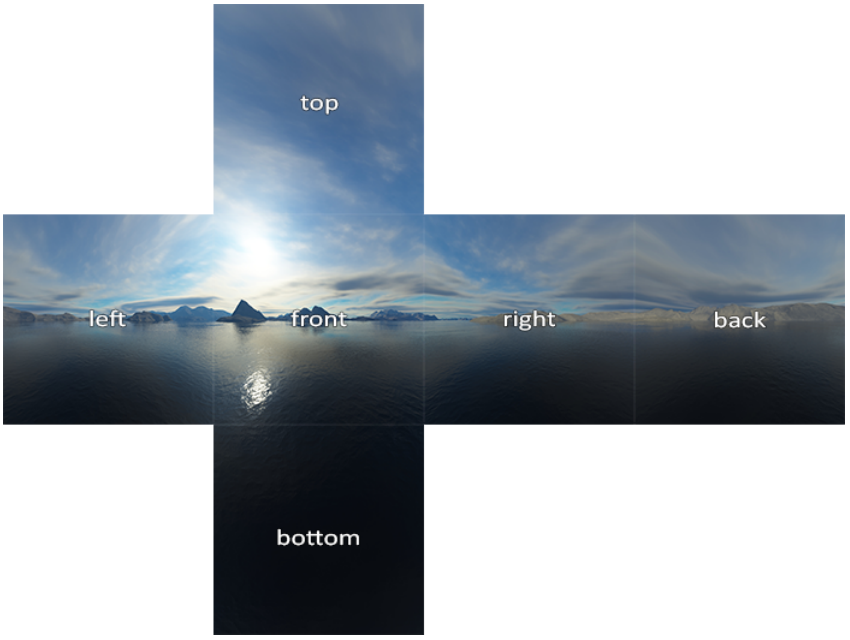


Fig 8 - Example of the 6 faces of a cubemap

We needed two ShaderPrograms: one for doing the skybox and the other for the fragments. In the fragment shader of the glass shards, we tried to use the Fresnel equation. Firstly, we calculate the reflection and refraction vectors using the functions given by glsl which we then use to sample from the skybox cubemap. Lastly, depending on the result given by the equation **(F)** we did a mix between both the reflection and refraction.

For the transparencies between each shard, we experimented with Blending. One problem we had was the order in which the shards were drawn. To fix this we had to sort them depending on the camera position.

In the blending function we used the *alpha* for the source and *1 - alpha* for the destination: glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA).

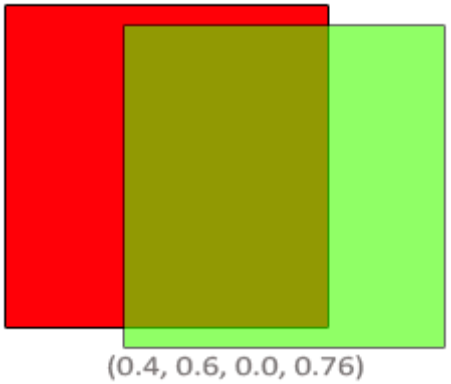


Fig 9 - Example of Blending with an alpha of 0.76

Because we wanted to have our own room as the cubemap we had to get the data for it from the scene. We experimented with Render Target Textures, placing the camera at the position where we wanted our cubemap to be rendered from. We rotated the camera to each of the 6 directions of the cubemap and rendered the scene graph without the glass shards to a secondary framebuffer that is associated to a texture. That is made in order to extract the pixel data from this texture and copy it into each of the cubemap's faces.

When testing this approach we noticed that, since the glass was not static and could be in many different places, a single cubemap wouldn't be enough to show reflections and refractions in all positions. We could instead use multiple cubemaps, created from different points of the scene, and use each one according to the position of each glass shard.

3.1.2. Final implementation

To make the glass we ended up using the Render Target Textures to make the cubemap as explained above. It was also taken into account the fact that the shards would fall into different places. To simplify, we only used two cubemaps one centered in the inside of the room and the other one in the perspective of the window.

Depending on the position of the fragments we would give a different cubemap texture, for the ones inside and the ones in the window frame. We used those cubemaps for the fragment shader of the glass only so that the vectors of the refraction and reflection could sample from it.

The Blending was something we used but we had to mix it with the Fresnel model. For that, the alpha value had to suffer some changes, the more reflective it was the higher the alpha value had to be so that it would not look so transparent. Figure 10 shows a bit of this changes.

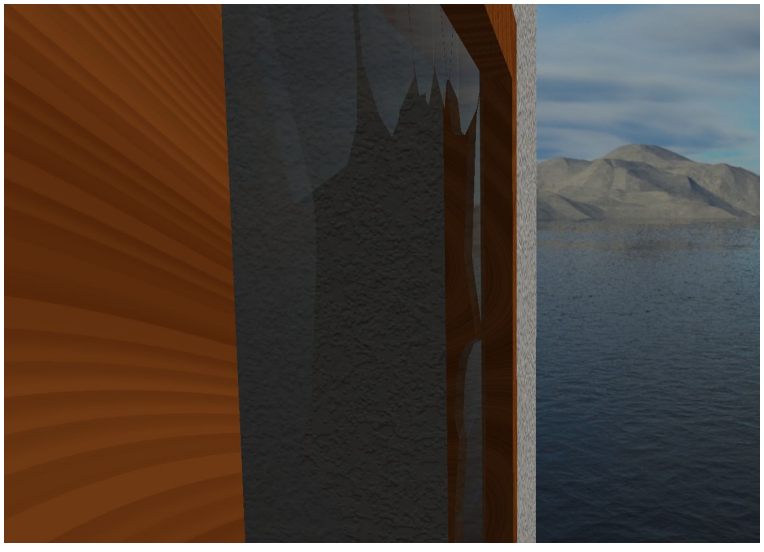


Fig 10 - Reflection of the mountains and the sky in the window

3.2. **Wood**

3.2.1. Explored approaches

In order to implement the shader for the wood we had to create one more set of vertex and fragment shaders and a new shader program to handle them.

On a first approach, we tried to implement a 2D version of the noise algorithm, but soon found out in an early version of the window frame (a cube at that point) that the object was not seamless, i.e., the noise algorithm would not continue from face to face, making it look like an object whose faces were glued together instead of a solid cube.

The most suitable algorithm we found to create realistic wood was a 3D chained simplex noise, as seen in Figure 11. This algorithm implementation was done in the wood fragment shader.

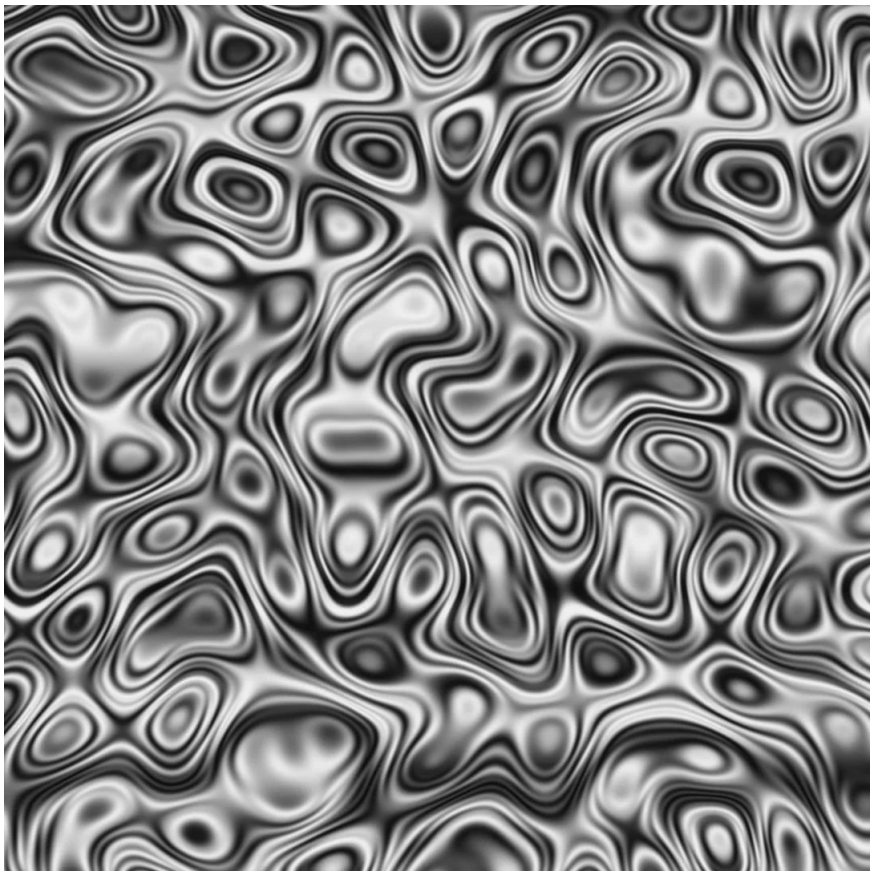


Fig 11 - Image of a chained simplex noise algorithm implementation

By recursively applying this algorithm we were able to interpolate between two colours, light and darker brown, in such a way that the rings get darker or lighter more naturally.

In order to get a more realistic lighting we made it so only the lighter browns would reflect the specular component of the light, unlike the dark ones which would absorb it.

3.2.2. Final implementation

The final wood shader implementation was based on the explored approaches mentioned above with extra attention to what colors most suited the replication of real wood and which best reflected the lights in the world of the created virtual room.

We also departed from the cube to a more lookalike window frame, in which the 3D simplex noise algorithm worked just as fine, as seen in Figure 12.

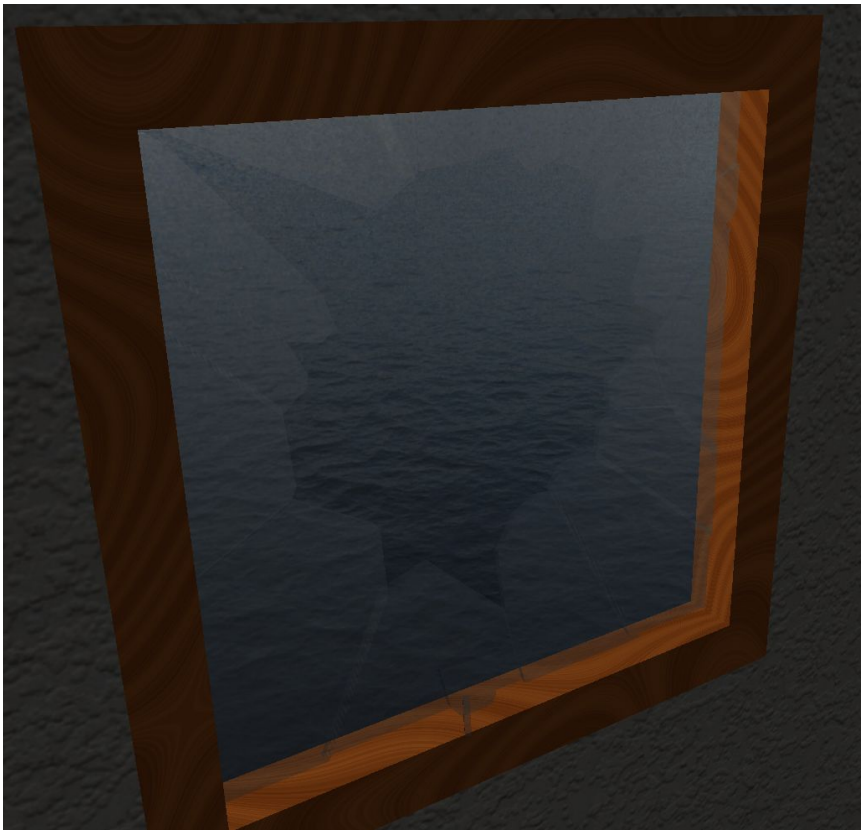


Fig. 12 - Final look of realistic wood frame

3.3. **Marble**

3.3.1. Explored approaches

Once again, to implement the shader for the marble we had to create another set of vertex and fragment shaders and a new shader program to handle this new set.

As in the wood shader we had to implement a 3D noise algorithm, otherwise the sphere was not seamless, resembling a wrapped sphere, as seen in Figure 13.

The most suitable algorithm we found to create realistic marble was 3D perlin noise, as seen in Figure XXX. This algorithm implementation was done in the wood fragment shader.

By applying perlin, “double perlin” or “double double perlin” we would get an increasingly granulated marble.

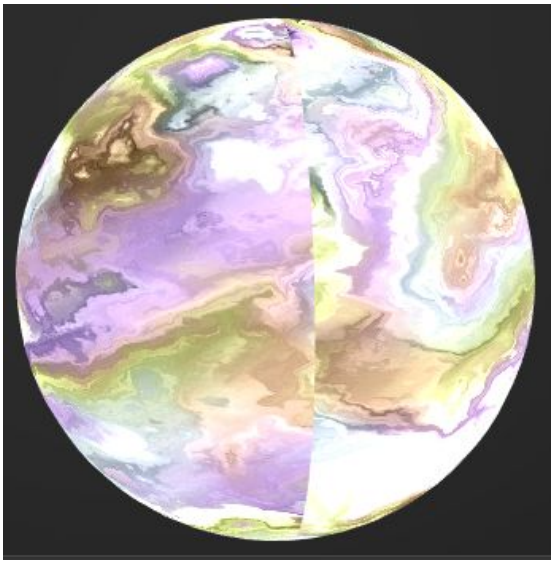


Fig 13 - Marble sphere with 2D noise algorithm implementation

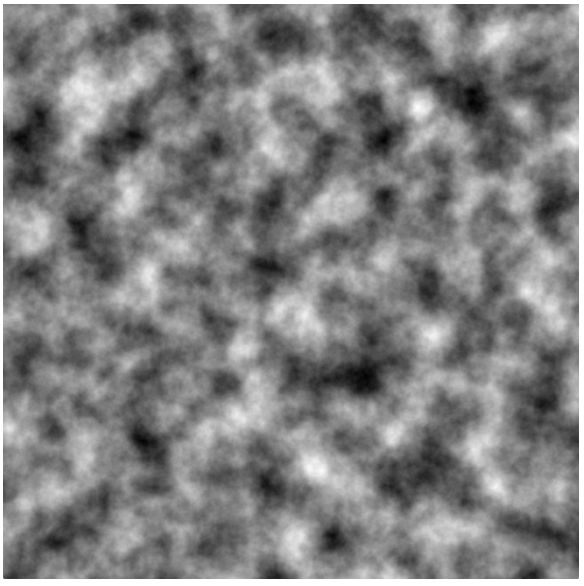


Fig. 14 - Image of a perlin noise algorithm implementation

3.3.2. Final implementation

The final implementation of the multicolored marble sphere was based on the explored approaches mentioned above with caution of what colors best fit without creating a rainbowy image.

We also paid attention to its lighting, focusing on accomplishing a realistic stone reflection of the light.

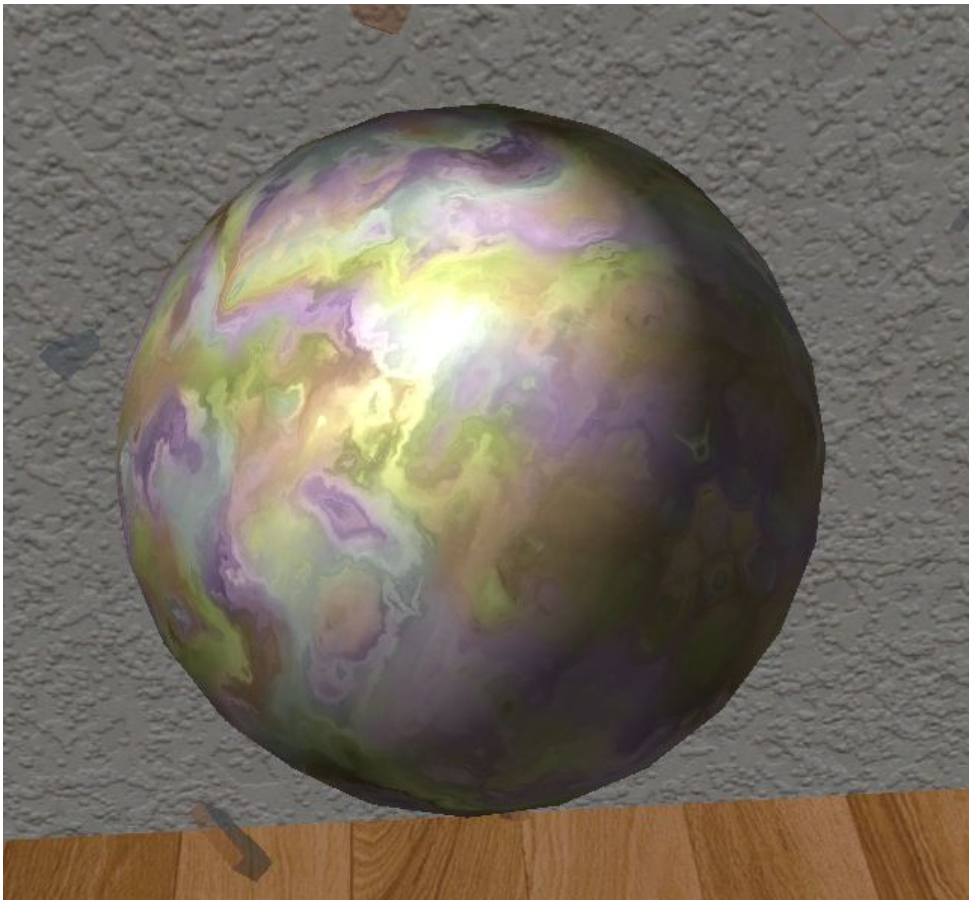


Fig. 15 - Final look of realistic multicolored marble sphere

3.4. **Procedurally Generated Glass Shattering**

3.4.1. Explored approaches

Mathematically, a Voronoi Diagram of a discrete point set in general position corresponds to the dual graph of the Delaunay Triangulation for **P**. This means that in order to compute the voronoi diagram we could scale down the problem to only perform a Delaunay Triangulation on a random set of points of the glass plane and then find its dual.

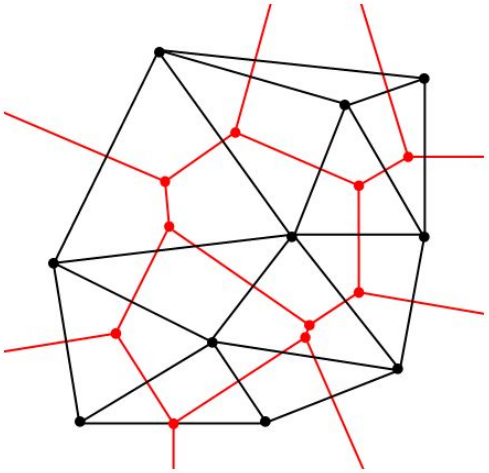


Fig. 16 - Delaunay Triangulation (black) and its dual, a Voronoi Diagram (red)

On a naïve approach, in order to perform a delaunay triangulation one would, at every new point inserted , connect it with the 2 nearest points that were already part of the set. This is a problem in itself as the 2 nearest points could make the new connections overlap the ones previously made or it could make the triangle too thin (called a non-delaunay triangulation). Both of these, when performing a dual calculation, would result in voronoi cells which would not tessellate. One solution to this problem is to detect if a new point lies in the circumcircle of a triangle formed with a subset of 3 points already in the set.

To do so, there is a matrix determinant we must calculate in order to understand if a new point D belongs to the circumcircle defined by the points A, B and C, as follows:

$$\begin{vmatrix} A_x & A_y & A_x^2+A_y^2 & 1 \\ B_x & B_y & B_x^2+B_y^2 & 1 \\ C_x & C_y & C_x^2+C_y^2 & 1 \\ D_x & D_y & D_x^2+D_y^2 & 1 \end{vmatrix} = \begin{vmatrix} A_x-D_x & A_y-D_y & (A_x^2-D_x^2)+(A_y^2-D_y^2) \\ B_x-D_x & B_y-D_y & (B_x^2-D_x^2)+(B_y^2-D_y^2) \\ C_x-D_x & C_y-D_y & (C_x^2-D_x^2)+(C_y^2-D_y^2) \end{vmatrix}$$

$$= \begin{vmatrix} A_x-D_x & A_y-D_y & (A_x-D_x)^2+(A_y-D_y)^2 \\ B_x-D_x & B_y-D_y & (B_x-D_x)^2+(B_y-D_y)^2 \\ C_x-D_x & C_y-D_y & (C_x-D_x)^2+(C_y-D_y)^2 \end{vmatrix} > 0$$

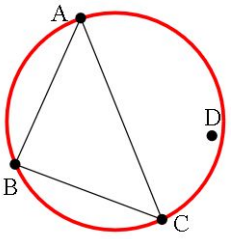


Fig. 17 - Determinant calculation

When A, B and C are sorted in a counterclockwise order, this determinant is positive if and only if D lies inside the circumference. If it does then we must do a flipping of the edges as visually explained:

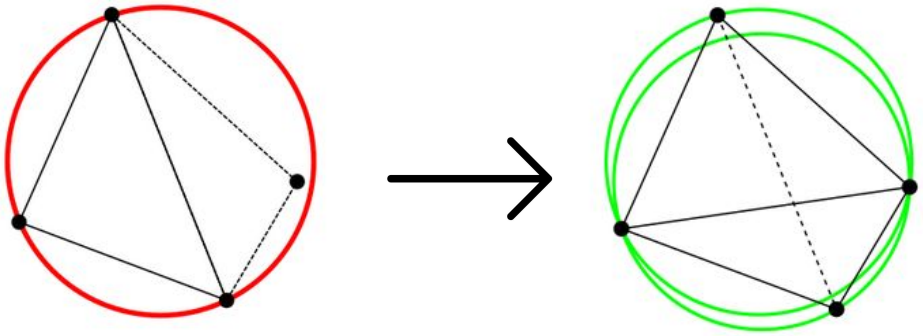


Fig. 18 - Flipping Algorithm Demonstration

This leads to a straightforward algorithm: construct any triangulation of the points, and then flip edges until no triangle is non-Delaunay. When a new vertex D is added, we split in three the triangle that contains D, then we apply the flip algorithm. Done naïvely, this will take O(n) time: we search through all the triangles to find the one that contains D , then we potentially flip away every triangle. Then the overall runtime is O(n^2) in the worst case scenario.

As one can see, when the goal is to ensure fast calculations in real time, this could be a burden. A new and faster way to calculate the diagram was needed.

3.4.2. Final implementation

While making some research, the algorithm which seemed both fast and doable was Fortune's Algorithm. We came across a third party code implementation of this algorithm which returned the voronoi edges. Using O(n log n) time to compute, this algorithm maintains both a sweep line and a beach line, which both move through the plane as the algorithm progresses making the voronoi cells in one go.

The beach line is a piecewise curve to the left of the sweep line, composed of pieces of parabolas; it divides the portion of the plane within which the Voronoi diagram can be known, regardless of what other points might be right of the sweep line, from the rest of the plane. For each point left of the sweep line, one can define a parabola of points equidistant from that point and from the sweep line; the beach line is the boundary of the union of these parabolas. When a new parabola is added to the beach line we call that a site event.

As the sweep line progresses, the vertices of the beach line, at which two parabolas cross, trace out the edges of the Voronoi diagram, while the ones at which three parabolas cross make a voronoi point. We call this a circle event. The beach line progresses by keeping each parabola base exactly halfway between the points initially swept over with the sweep line, and the new position of the sweep line. Mathematically, this means each parabola is formed by using the sweep line as the directrix and the input point as the focus.

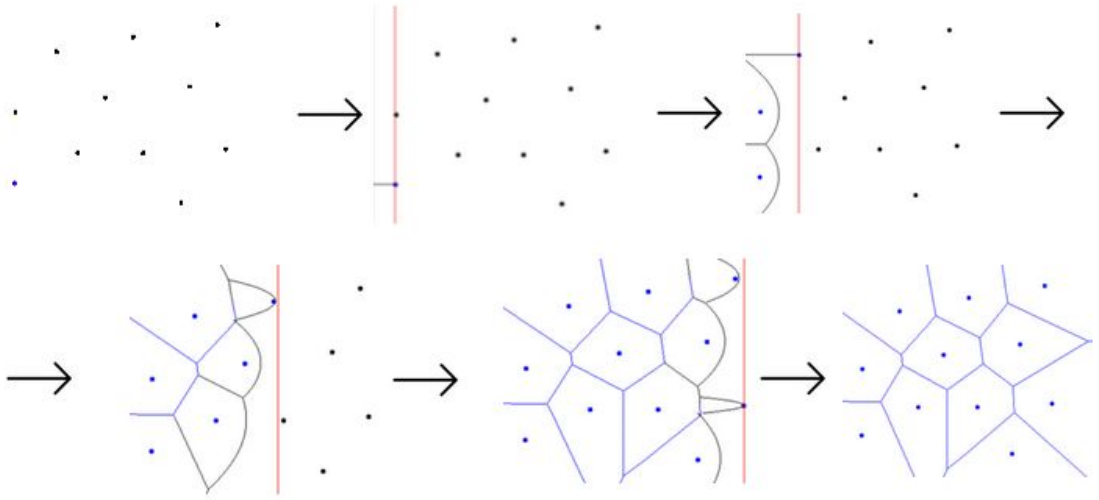


Fig. 19 - Fortune's Algorithm Visually Explained

From that point on, we used the returned edges (which were composed of an initial vertex and a final vertex) to identify to which polygon it belonged to through euclidean distance squared. After, we identified the polygons' points by traversing its edges and adding them sequentially. Polygons which belonged to some border of the plane and corner had to be worked around so that the adding of a new (corner) point would be possible. Polygons which had 2 corner points were not taken into account as it would not be possible for the amount of shard to compute one with 2 corners.

The third party code had an issue: one edge would sometimes be split in 2, resulting in a new (superfluous) point which had then to be removed to avoid conflicts with the physics engine's behaviour.

Lastly, the polygons were given thickness and converted into meshes.

3.5. **Physics Engine Integration**

3.5.1. Explored approaches

To keep the scenegraph nodes synchronized with the physics world, each node keeps a reference to a rigidbody that is part of *rp3d*'s dynamic world. Before each frame, we update the scene's time accumulator, and perform as many physics steps as is allowed with current value of the time accumulator. We follow this asynchronous approach so that the physics world always gets updated with the same time step every time, which provides a more accurate simulation. We then update each node's transform to match that of it's associated rigidbody. Here is a simplified diagram on how this update process works:

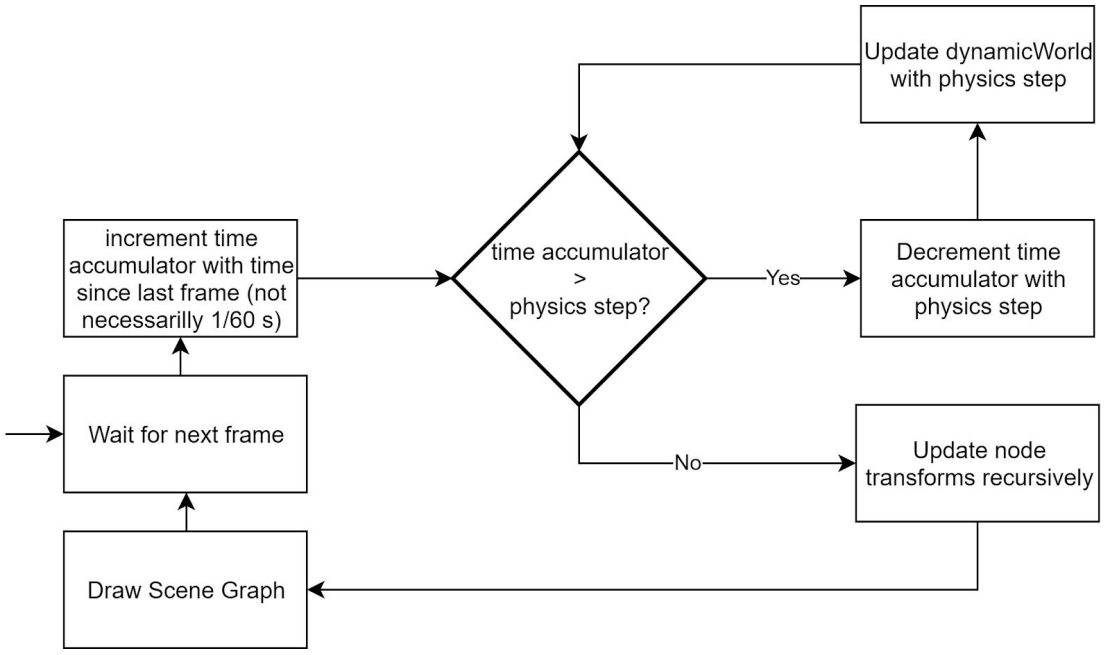


Fig 20 - Update process

In order for *rp3d* to recognize meshes as actual objects with volume, they must be provided as "closed" meshes. What this means is that the faces of each mesh must share vertices to indicate to *rp3d* which edges they have in common to allow for collision detection. The code we were given in the labs to load 3D models simply replicated the vertices for each face they were a part of and our Mesh classes worked around that principle as well, so that had to be changed to instead work with a vertex/index system.

To create physics objects with custom meshes, *rp3d* requires that the vertex and polygon data be provided in a specific format and be available in such format for as long as the simulation is running. The vertices must be provided concatenated in a float array, and the polygons must be provided as an array of indices and a corresponding array of PolygonFace, with each PolygonFace containing the start index in the index array and the number of vertices that face has (note that not all physics object types may contain faces other than triangles, only convex meshes are allowed under these conditions). This means we must save these data arrays in a way that they are available to rp3d while we are running the simulation. The most obvious and straightforward way to go about this is to save them in the respective node.

To implement physics objects with concave meshes, we first tried using the ConcaveMeshShape class, given that it was what was recommended in *rp3d*'s documentation. However we noticed severe collision detection problems, where some objects colliding with concave meshes would simply go through, get stuck inside them or have jerky movements while standing atop them. After thoroughly analysing the way we were creating the rigidbodies and ensuring we were following the documentation correctly, we concluded that *rp3d*'s concave mesh performance was not sufficient for the purpose of our application. We then explored the other option which was to divide the meshes into simpler convex components, which had already proven to be significantly less prone to the previously mentioned errors. This however implied that we had to export our meshes already split into these simple components, as developing an algorithm to split the meshes procedurally in runtime would be too heavy an undertaking for the scope of this project.

When testing the behaviour of the glass shards, we noticed some unexpected behaviour, in which some shards would intersect each other and the response from colliding with other shards and objects in the scene was not what was expected, with more extreme cases like the shards appearing to float around the floor, after hitting it, as if they were anchored to it. After some analysis, we concluded the rigidbodies's center of mass was being miscalculated. To fix this we needed to provide a more accurate center of mass for each object. Several options were considered, such as the geometric center, which was very accurate but not fast or straightforward to calculate given the various shapes the shards could have, and the average of the vertices, which while less accurate is very easy and performant to obtain.

3.5.2. Final implementation

To keep the scene graph and physics world synchronized we ended up using the system illustrated in Fig. 4, as it was simple enough to implement and proved to work without problems while also being scalable as we were developing the rest of the application's systems.

To allow for proper collision detection, we implemented a vertex/index system for the faces of meshes, in which all unique vertices in a mesh are kept in a vector array and all faces are a list of indices of that array. We also changed the way we draw polygons to use the GL_TRIANGLE_FAN primitive, to allow for n size polygons to be drawn without having to be triangulated beforehand.

Saving the vertex and index arrays in each node proved to be an adequate solution to maintaining the data available for *rp3d* to access during the simulation.

To represent concave objects in our scene we opted for a hybrid approach between the two explored options, subdividing complex objects into convex meshes as much as possible and using ConcaveMeshShape when either not many collisions will occur with that object or it's too hard to subdivide it.

To fix the center of mass of objects in the scene we ended up using the average of the objects vertices, as it proved sufficient for our objective and didn't present any detectable inconsistencies in the behaviour of the rigidbodies.

4. Post-Mortem

4.1. What went well?

One aspect of the development process that went really well was the merging of all our features. Since we all used a common code base when developing each individual's features, joining them was relatively simple, without very little code refactoring being needed. We also followed code conventions and developed the code in a very modular way to allow for it to be quickly integrated and improved as the features began to interact with each other. This resulted in us being able to merge all of our features in what was essentially a day.

4.2. What did not go so well?

Our implementation of the glass material using static cubemaps rendered at the start of the application runtime proved not to be the best for our objectives, since the glass shards are very much not static objects and could be in very different positions. Overall we like the results of our multiple cubemap solution, but given more time we would, without a doubt, revisit this problem to find a better solution.

Another huge source of stress during this project's development was the unexpected bugs due to the physics engine. Due to the lack of time and perhaps a mismanagement of resources, we did not devote enough time to experiment with ReactPhysics3D. This resulted in many issues that took a really long time to fix due to our inexperience with the engine, and bugs which we could not decide whether to blame on our usage of it or itself.

4.3. Lessons learned

From the development of this project we took away a few lessons:

- When using 3rd party libraries, specially really auto contained ones like ReactPhysics3D, save some time to thoroughly explore the inner workings of the library's functionalities. Find out the weaknesses and strengths of the systems and figure out beforehand what might be the best way to use it to your advantage.
- When implementing dynamic materials like glass, metal or water, test them in the conditions you intend to use them in (moving around in a room in our case), to make sure they behave the way you expect them to.
- The realm of computational mathematics is a very theoretical one, but there are papers and interactive websites that help explain what is behind the theory and, which makes it more tangible. If we could, we would start-off immediately implementing the Fortune's Algorithm without spending so much time reading all the papers necessary to fully understand a concept which then was not done, it only served to expand our horizons on how to approach similar problems. Exploring computational mathematics proved to be challenging but very rewarding.
- We learned how to work with multiple shaders.

References

Dennis Gustafsson's Homepage
http://www.tuxedolabs.com/

Smash Hit (Fig. 1)
https://play.google.com/store/apps/details?id=com.mediocre.smashhit&hl=en

ReactPhysics3D
https://www.reactphysics3d.com/

Glass Shatter (Fig. 2)
https://static.wixstatic.com/media/cea698_915107eed8f94efd9ae3ae79dc55b723~mv2_d_4032_3024_s_4_2.jpg/v1/fill/w_4032,h_3024,al_c,q_85/cea698_915107eed8f94efd9ae3ae79dc55b723~mv2_d_4032_3024_s_4_2.jpg

Wood Image (Fig. 3)
http://www.textures4photoshop.com/tex/wood/free-wood-texture-with-high-resolution.aspx

Marble Image (Fig. 4)
https://www.wildtextures.com/free-textures/multicolor-marble-stone-texture/

Voronoi Example (Fig. 5)
https://philogb.github.io/blog/2010/02/12/voronoi-tessellation/

Fresnel (Fig. 6)
https://fenix.tecnico.ulisboa.pt/downloadFile/1689468335628587/Fresnel2019_handouts.pdf

Fresnel Reflectance (Fig. 7)
https://www.scratchapixel.com/images/upload/shading-intro/shad-fresnel2.png?

CubeMap (Fig. 8)
https://learnopengl.com/img/advanced/cubemaps_skybox.png

Blending (Fig. 9)
https://learnopengl.com/img/advanced/blending_equation_mixed.png

Not Seamless (Fig. 11)
https://shaderfrog.com/app/

Perlin (Fig. 14)
https://medium.com/@yvanscher/playing-with-perlin-noise-generating-realistic-archipelagos-b59f004d8401

Delaunay Triangulation and Voronoi Diagram (Fig. 16)
https://hpaulkeeler.com/voronoi-dirichlet-tessellations/

Delaunay Triangulation Explanation
https://en.wikipedia.org/wiki/Delaunay_triangulation

Fortune's Algorithm based on
https://www.cs.hmc.edu/~mbrubeck/voronoi.html

Voronoi Interactive
http://alexbeutel.com/webgl/voronoi.html

Fortune's Algorithm Interactive
https://www.desmos.com/calculator/ejatebvup4

Shaders
https://thebookofshaders.com/

https://learnopengl.com/

ReactPhysics3D
https://www.reactphysics3d.com/