Computer Graphics for Games 2020

The Enchanted Rose

Recreating the magic of the original enchanted rose from the animated movie “Beauty and the Beast” using OpenGL

Diogo Silva	**Daniel Gonçalves**	**Henrique Gaspar**	**Guilherme Menezes**
98776	98845	98879	76494
MEIC-T	MEIC-T	MEIC-T	MEIC-T
diogo.goncalves.silva@tecnico.ulisboa.pt	daniel.f.g.goncalves@tecnico.ulisboa.pt	henrique.a.p.gaspar@tecnico.ulisboa.pt	guilherme.menezes@ist.utl.pt

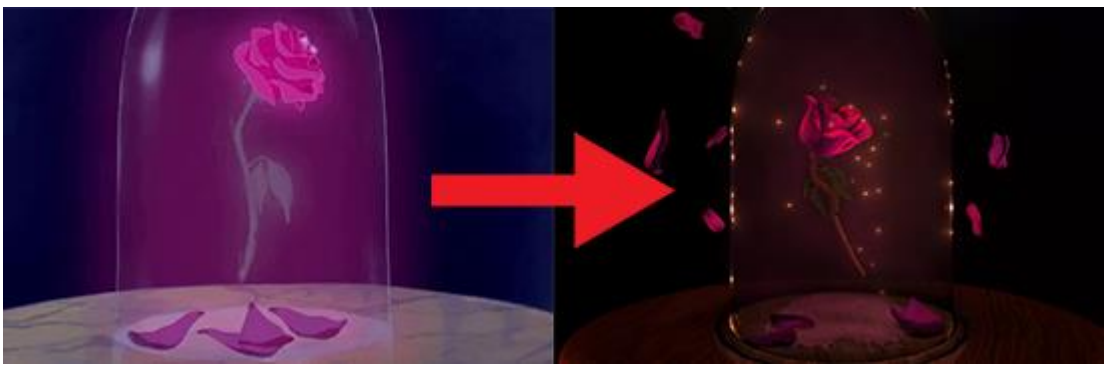
Abstract

The concept presented is called “The Enchanted Rose” and consists of a 3D representation of the iconic magical rose from Walt Disney’s 1991 movie “Beauty and the Beast”. It was decided to have not only custom design models for the Rose, the Stem, the base plate and the handle, in order to achieve exactly what was intended, but to have as well custom materials for each one of them, using different kinds of texture generation (for consistency reasons, only the handle and the base plate share the same material). On top of this, and to bring some magical and faithful 3D related elements to the scene, such as Realistic Lighting (Blinn-Phong) to improve the general mood of the ambient, Shadows to complement the lighting, Reflexions for a true glass look, a custom skybox for ambient setting and two different particle systems to increase on the magic (one as an instanced based solution and the other using a geometry shader).

1. Concept

The original concept for this project was to make a three dimensional recreation of one of the most well known objects(?) in Disney history: The Beast’s rose, using an OpenGL engine developed by the team to achieve it.

For that, the team searched online for 3D representations of the rose to get some inspiration. It was quickly decided that the focus would be on the original 1991 movie representation of the rose, instead of the 2017 remake one.



For this recreation to be faithful, it was needed to make 3D models for the rose itself, the marble base and the glass dome. Each of these would need to have different types of materials to fully realise the look of the scene. An objective of the team would be to also try and implement different ways of creating materials, using image textures and procedural noise textures as well.

The team also decided to implement a particle system, mirroring the one that emanates from the flower in the original movie, since that would help bring the magic of the scene to our 3D representation. Another particle system that would look like petals falling from the rose was also implemented.

To bring the whole scene together, the team decided on a realistic lighting model that would enable the dome to cast reflections and the rose to cast shadow, on itself and on the marble underneath. All of this would be complemented with a skybox that would try and mirror the one seen in the movie, to create the ambience needed for the scene.

An objective for the application was to also have it be interactable, in order for the user to be able to see the rose in different ways and angles, previously unseen because of the static nature of a drawn animated movie.

After the basic concept was completed, it was time to move on to the actual implementation of the application.

2. Technical Challenges

2.1. Creating the object's models [Guilherme Menezes]

The team decided that they didn't want to rely on external resources, wanting every model's mesh to be of their own authorship. As such, the team had to create a **Rose**, **Stem**, **Base**, a **Dome** and a **handle** model. that would mimic the original animated film's models' looks.

2.2. Generic scene graph handling hierarchical drawing [Everyone]

The entire scene had to be managed in a hierarchical fashion in order to allow for the entire rose/dome to be moved in tandem (and for it to function as a congruent object). The hardships came from both adapting code from our previous assignments, but also certain changes that had to be made in order to improve the correctness of some of our classes (such as the SceneNode).

2.3. Creating realistic looking glass [Diogo Silva]

One of the first challenges the team tackled was the creation of glass in order to represent the dome that envelops our rose. The prompt asked for a material with transparencies but we wanted to aim for a "realistic" look, meaning we had to deal with both reflections and refractions.

2.4. Creating a Particle System [Daniel Gonçalves, Diogo Silva]

One of the toughest challenges was to create an instanced rendering based particle system in order to represent the flakes of light that surround the rose and add to the whole "magic" of the scene. So basically, we wanted to have several particles surrounding the rose, gently falling down around it and reappearing.

2.5. Implementing Blinn-Phong Lighting [Guilherme Menezes, Diogo Silva]

At the time of establishing which challenges were to be tackled, the team was torn between picking a cartoony CEL-Shaded look, or a more realistic one. In the end, it was decided to go with a more realistic look using **Blinn-Phong Lighting**, in order to achieve a mix of the classic scene's magic and the remake's realism.

2.6. Creating good looking materials [Henrique Gaspar]

Apart from the glass dome, the other objects, like the stem, rose, base and handle, would also need realistic, distinct, good looking **materials**.

2.7. Implementing shadows [Diogo Silva, Guilherme Menezes]

In order for the scene to not look as bland or flat, the team decided that shadows were needed (especially to go along with the realism provided by the *Blinn-Phong* Lighting model). These shadows would allow certain objects in our scene to pop off, such as the rose's folds and the stem's curves.

2.8. Creating a Skybox [Henrique Gaspar, Diogo Silva]

The team didn't want the rose to just be floating in the middle of a void, and as such a skybox had to be implemented in order to enrich our scene. Besides the challenge of rendering the skybox itself, however, we also faced the problem of actually creating the skybox's textures to reflect the room where the scene takes place in the original movie.

3. Proposed Solutions

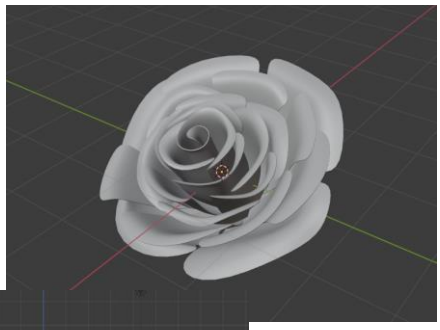
3.1. Modelling the Objects

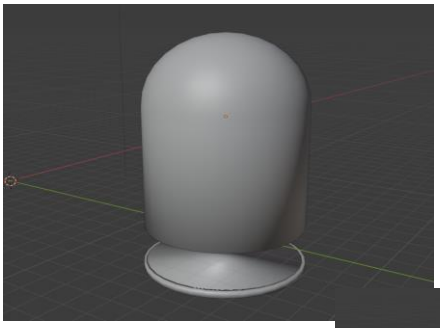
3.1.1. Explored Approaches

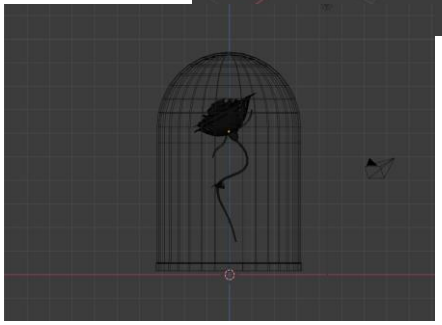
In order to create every model present in the scene, a free open source 3D creation software called Blender was used, and multiple techniques were applied. One of the problems found with this method was the incorrect normals associated with each model. The team had to return to Blender and correct this issue on a later date.

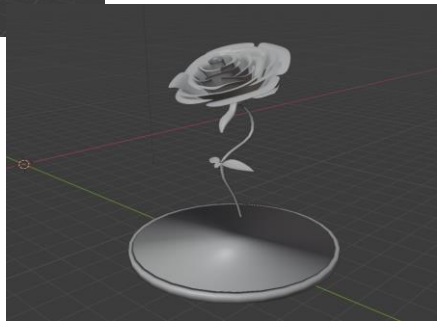
3.1.2. Final Implementation

Besides designing every model from scratch, the UV maps were exported together with the objects so that the team could apply the correct textures to every model.









3.2. Generic scene graph handling hierarchical drawing

3.2.1. Explored Approaches

This challenge had already been worked on in the previous 4th assignment. Initially, each of our scene nodes had a "local transform" matrix and a "world transform" matrix. Whilst the first would be initialized as an identity matrix upon the creation of the scene node and updated by multiplying with transformation matrices, the second held the node's parent's local transform (and so on, recursively). So, for example, to translate a node we multiplied the node's local transform with a translation matrix, and updated all of that node's children's world transforms to reflect this new local transform. This, however, was not optimal, and was altered for our final implementation.

3.2.2. Final implementation

In the end, each of our scene nodes held three parameters, Position, Scale and Rotation, with the first two being Vector3's and the last being a Quaternion. Besides this, we only fetched the node's parent's parameters when it was time to draw the node, rather than at every update (gaining some optimization). We would also like to mention that several other improvements were done such as the way our camera was controlled changing to a drag and rotate rather than just a rotate, but these didn't warrant a full "Challenge" section of their own.

3.3. Creating realistic looking glass

3.3.1. Explored approaches

As the team had never worked with transparency, we decided to start off by doing just that using **alpha blending**. First, we created a new *Glass Material* using our material system and, using our basic shader, gave it a colour with an alpha value lower than one (0.15). In order for this to work, we had to enable blending and make sure that the glass was the **last** model rendered into the scene (since the depth buffer doesn't automatically order the objects, meaning if non-transparent models were rendered after the dome, the dome wouldn't be transparent towards them). For our blending function, we used `glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA).` The result achieved is shown in the following image.

This was decent for a first implementation, but there were more things we could add to the glass to give it a more "realistic" look. We decided to achieve this by implementing *reflection* and *refraction*. At this point, we didn't want to actually implement any frame-buffer based reflections and instead concluded that simply doing some reflection and refraction effects of the skybox would achieve our desired look. For this, we created a new *Glass.shader*. Using the "normal" air and glass *refraction indices* we were able to compute the glass' reflection and refraction values, as well as Schlick's Approximation of the *Fresnel Factor* value (which describes the reflection and transmission of light).

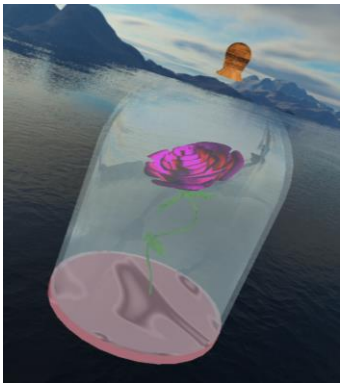
Using these values, on our *Glass.shader*'s fragment shader we were then able to compute the refraction and reflection "colour" (achieved by first using the *texture* function with our skybox's texture and our previously computed refraction and reflection values, respectively) and then setting the strength of these effects by dampening them with "anti-reflection" and "anti-refraction" values. Finally, the glass's colour was computed by mixing the refraction and reflection colours using our computed Fresnel Factor. Following is the result of this operation.

As can be seen, there were some issues to be corrected, notably those "triangle artefacts" and the fact that we could only see an "outline" of things that were inside the dome.

We took a while to realize this, but the source of our problems was the fact that we had Cull_Faces disabled, and since the dome "loops in on itself" (i.e we can see the back of the dome from the front), there was an issue in the blending. This was actually also noticeable before adding our reflection and refraction effects, but not as much (only by zooming in really close to the glass, would it be noticeable). So all we did was enable face culling and the result was the following.

3.3.2. Final implementation

For our final implementation the team basically just played around with values for our glass' reflection and refraction strength. Per the teacher's recommendation, we also added a "second dome", slightly smaller and placed inside the former (and while the outer dome would have its backface culled, the inner dome would do front face culling instead). This gave our dome some girth and overall improved its quality, and as such, this was our final result.



3.4. Creating a Particle System

3.4.1. Explored Approaches

For the Particle System, the team ended up with a far more complex state of analysis and implementation. After a long research period, one approach was selected and implemented: An **Instance based particle system**.
Later, during a discussion with the teacher, it was indicated that the development team should use a more shader based approach to fulfill the project requirement.
This led to another period of discussion and research and another solution was found: a **geometry shader based particle system**.
This second approach was far more easily implemented than the previous one, but was also deprived of the flexibility available for the previous system.
Here is a small comparison between both methods:

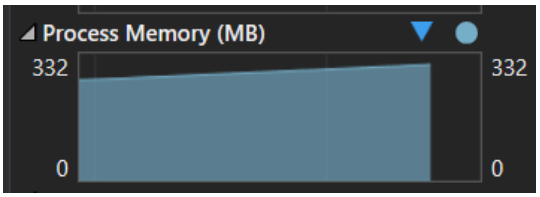
	Instance based	**Geometry shader based**
Implementation	- Longer and more complex	- Easier and smaller - The direction of each particle needs to be calculated and it's not trivial to do so.
Flexibility	- Fully configurable (first color, last color, first size, last size, speed, size variation, LifeTime, speed variation, position, etc.)	- Few changeable configurations (only size, color, gravity and number of particles per second and total, position and life). - At a root level, the particles

	- Each particle can even have a mesh that is created by loading a .obj file directly at the code.	are just points that are transformed into cubes at the geometry shader. If a new geometry is needed, it was to be hard coded on the shader.
Demand of resources	- Much more complex and susceptible to memory leaks	- Far simpler and lighter

- **Instance based particle system**

As said before, this was the first approach to be followed and implemented. Aesthetically, it was pretty pleasing and it's currently simulating falling petals, an effect that was made just by testing this feature with a random .obj file (the one used for the rose) that ended up to be perfect for the look that the team was going for.

This approach demanded an update of the project code structure and made a single frame draw not five objects, but a few hundreds. This emphasized a clear flaw on our project, a big memory leak that was previously so small that went unnoticed.



This memory leak was present at every Matrix4 creation along the code, just because our constructor for that class demanded a float array and we were creating it at the constructor invocation itself, something like `Matrix4 m4 (new float[16]{})`. The team quickly understood, by doing some research, that this way of creating a float array was keeping its memory reserved for the full extent of the project execution. This meant that if a Matrix had to be created at each frame, and its memory was never released, it would always ask for more and more MB of memory. In other words, a memory leak.

The solution was pretty simple, though. By creating the array with its designated values and then passing it to the constructor, the problem was solved.

- **Geometry-Shader based Particle System**

For our second solution, the process was the same as before: search online for a way to implement exactly what we wanted and then try and create an implemented version that would fit the previously defined needs. But this way of establishing a particle system doesn't seem to be a very usual one, and this is sustained by the fact that there were very few implementations using a geometry shader and a lot of discussions that indicated the Instance based approach was a very much clearer and obvious choice.

After some time and some code restructuration, there was only a small problem left standing: The particles were not receiving the view and projection matrices.

This was quickly solved with just a line by line review of the uniform related part of the code.

Some small tweaks were also made to the configurable options of the particle system and the function that generated a unit vector inside a cone for the direction of each particle was replaced by one that does the some but for a sphere instead.

At the end, a final change was made to both particle systems in order to change the particles' spawn point when moving the rose itself.

3.4.2. Final implementation

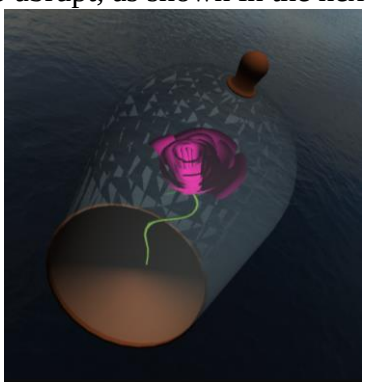
For the final implementation, and for demo purposes, we decided to integrate both Particle systems, allowing the user to change between them by hitting a keyboard key (V).

3.5. Implementing Blinn-Phong Lighting

3.5.1. Explored Approaches

In order to implement the Blinn-Phong Lighting model, the team started by applying the Phong lighting model to the project. This included three different types of lighting: Ambient lighting, represented by the shade of colour always present in every object so that even in the dark, the objects still had some kind of colour. Diffuse lighting, where a lighter colour was applied to the faces turned toward the light source and finally, Specular lighting represented by an even lighter spot in shining objects highlighting the light source.

This method was applied to every object's shader so that the team had the option of changing some specific object's lighting parameters if they wished to. Even though the results were satisfactory, there were areas where the change from a lit area and an area being in shadow was too abrupt, as shown in the next image.



3.5.2. Final implementation

To fix this issue, the team implemented the Blinn-Phong lighting model where instead of using the relationship between the reflection vector and the view vector to implement Specular lighting, a "halfway vector", that is precisely halfway between the reflection vector and the view vector, was used instead of the reflection vector. Doing this increased the specular highlight on a reflective surface the closer the player was to the light source.

3.6. Creating good looking materials

3.6.1. Explored Approaches

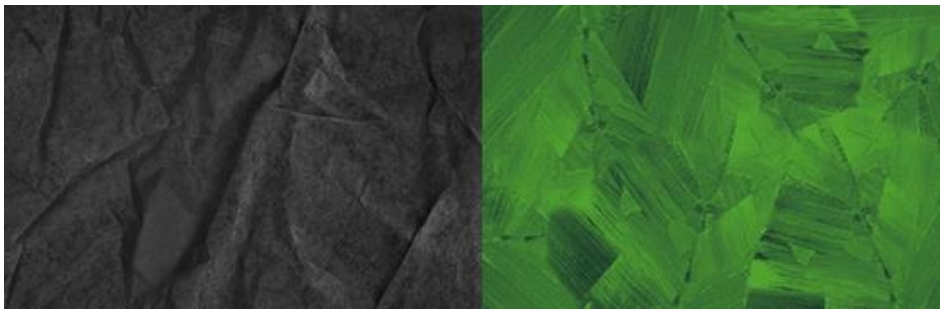
- Taking into account the different types of materials the scene would need, the team decided it would be best to implement different ways of applying textures to each object, in order to better understand the options that OpenGL provides when doing so.
-
- After consideration, it was decided that the application would use:

- Image texturing, loading textures into OpenGL for the rose and the stem;
- Procedural generated texturing, using noise functions for the marble base.

The first one to be developed was the rose, since it was the focal point of the scene, it needed the most attention, because if it wouldn't look good the whole application would fall apart.

The image texture was decided for this, since the team felt the margin for error would be smaller using this method, because an image texture could be easily replaced if it didn't look good.

For that, the application needed a way of importing image textures using code, and the team decided to use an image-loading library called stb_image.h, because it supported the most used formats, like .jpg and .png that would be used in this case. The following images would be the ones used to texture the rose and the stem respectively:



For both, the base texture is repeated using the default OpenGL behaviour for textures, so the model is completely filled with the image using it's texture coordinates.

Adding to that, as can be seen, the rose texture is in greyscale. This would allow for the addition of a base color that could be parameterizable allowing the team to tinker with it in order to achieve the perfect look for the rose.

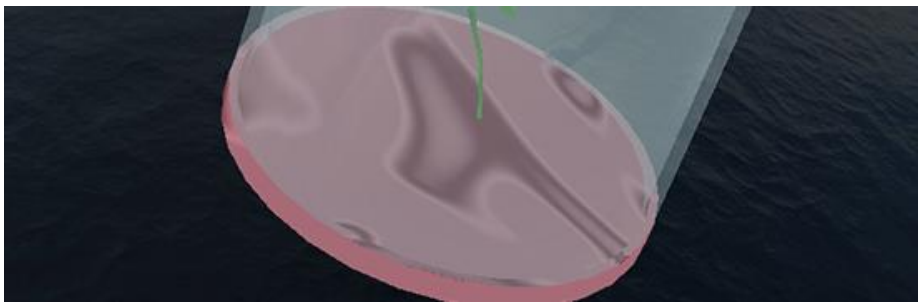
With the look of the rose being in an acceptable state, work on the marble base could be started. Upon some investigation, this material had some specific qualities that the team needed to take into account: marble's surface is very shiny and reflective with little to no bumps all the while containing very specific patterns that are only present in this type of material.

To achieve the shiny and reflective look, the ambient, diffuse and specular properties of the material were changed to better fit these characteristics.
For the physical properties of the surface, since marble's surface is always very flat and smooth, there was no need to implement things like normal mapping, and the team went with using simply the basic properties for the material.

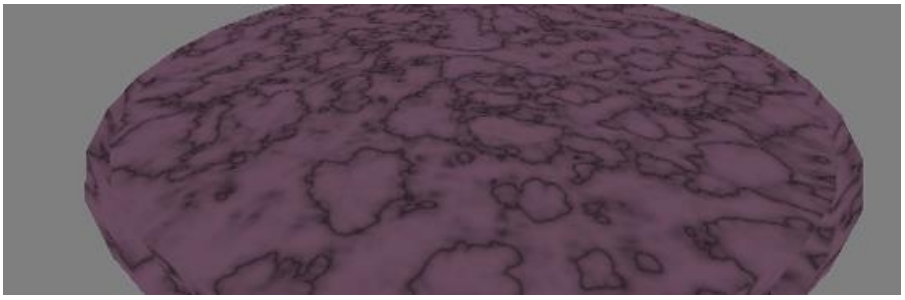
The biggest challenge was getting the very recognizable pattern present in marble.

The first iteration used a very simple implementation of the perlin noise algorithm, and since there was no need for bumps, this noise was also implemented in 2D. But the noise was generating a blocky texture that could work for a stone material for example, but looking at the reference material, the pattern consists of lines that are randomly created. To achieve this, the algorithm had to be changed to one of **Gradient Noise**, treating it like a distance field, resulting in the following texture:

While this was far from what was intended, this texture already contained some of the features necessary, like the swirly continuous lines and the random nature of their positioning. So, and like what was done with the greyscale texture for the rose, a base color could be added to this generated texture in order to achieve a look resembling the marble texture from the movie:



The marble looked close to what was intended, but overall it wasn't satisfactory, so search began on a new way of achieving the look of the pattern, finding a solution (reference in "References" section) that looked much closer to what was expected. This solution basically created black lines using the model-space vertex coordinates and a sine solution of the x-coordinate and then applying perlin noise to those lines. This created a black and white texture that looked much better when compared to the reference photo. After the texture look was satisfactory, it only needed the color added like it was done previously. Reaching the following look:



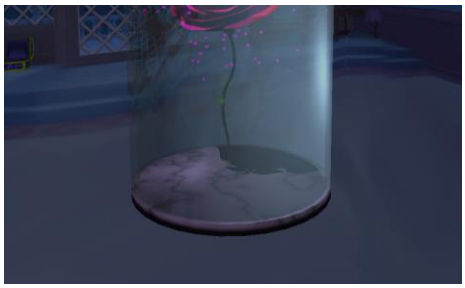
3.6.2. Final implementation

For our final implementation, one adjustment was made to each of the materials.

For the rose material, as previously stated, a parameterizable color was added, and as such, a changing color was implemented to simulate a glowing effect, to further the magical feeling the rose provides.

For the marble material, the strength of the blacks on the noise texture was too high, so after bringing that value down a bit, the look mirrored a marble surface much better.

Achieving a final look that the team felt capture the essence of the rose from the movie.



3.7. Implementing shadows

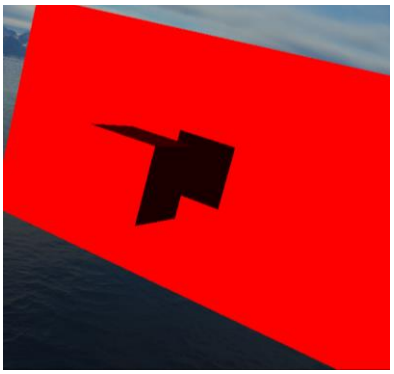
3.7.1. Explored Approaches

Following the challenge's prompt, the team realized that to implement shadows they would need to utilize the *frame buffer*. The technique we ended up settling on, dubbed **Shadow Mapping** basically consists in rendering our scene from our light's point of view in order to create a **Depth Map** (a texture corresponding to the depth values of our scene, as seen from the light).

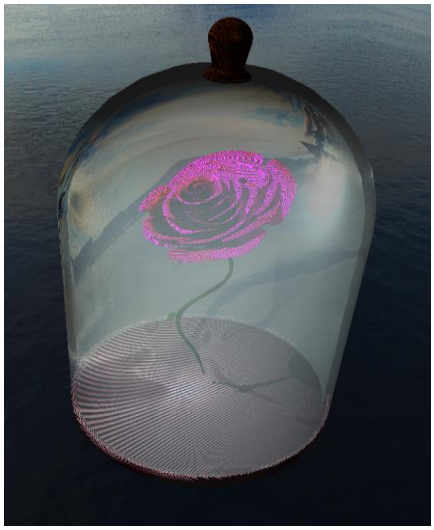
We started off by simplifying our scene to be just a single cube, a wall (the cube but stretched) and a light source in front of the cube. With this we would be able to more easily verify the correct functioning of our implementation and (hopefully) make debugging easier. To generate our Depth Map, we begin by defining a *frame buffer* object to render our scene from the light's perspective and then creating a 2D Texture to use as that buffer's *depth buffer* (note that, as we don't need a color buffer for this rendering, we set both the read and draw buffer to *GL_None* using *glDrawBuffer* and *glReadBuffer* in order to explicitly tell OpenGL that no color data will be rendered). After this configuration step, we can then render our scene, by binding to our created frame buffer (and setting our viewport to be equal to our Depth Map texture's dimensions). We render our entire scene using a simplified shader - *LightDepth.shader* - which besides containing an empty main for its fragment shader part, takes, in the vertex shader, each of our scene's models and transforms them into "light space" (in a similar process as the one done to render our scene into our camera's space). In the end we have a texture that corresponds to our frame buffer's depth buffer, which we can then showcase by rendering onto a quad (occupying the entire scene space).

In the image we can basically see what the light "sees" in terms of depth. There's the big wall that we mentioned before, and that darker part corresponds to the section of the wall that is obscured due to a cube being in front of it (hence the depth value being higher). After having this texture, we can now use it to actually render the shadows onto our scene. This is done by passing the Depth Map to our shaders as a texture and

using the light's position to compute, in the fragment shaders, a shadow value (ranging from 0 to 1, with 1 corresponding to the section being completely in shadow). This shadow value is calculated by taking the closest depth value (using our depth map) from the light's perspective, getting the current fragment from the lights perspective and checking whether or not that fragment's position is or is not in shadow. We then add this value to our Blinn-Phong's calculations and end up with something as follows.



There were still improvements to be done, however. Reverting back to our original scene, we noticed some artefacts - dubbed *Shadow Acne* - happening, and as such these had to be fixed for our final implementation (note the stripes).



3.7.2. Final implementation

For our final implementation two improvements were applied. First we fixed Shadow Acne by using *Shadow Bias*. Shadow Acne is caused by the our Depth Map's resolution limitation. The difference between our scene's resolution and our depth map may cause multiple fragments to sample the same value from the depth map even when they're far away from the light source. The issue with this is that, when the light hits our surfaces at an angle, the depth map is (obviously) rendered at an angle, and several fragments will access the same tilted depth texel despite being above or below

the surface leading to a *shadow discrepancy*. Using *Shadow Bias* we offset the depth of our surface by a small value such that fragments stop being incorrectly considered as being below the surface. A downside to this fix is that it causes a different artefact called *Peter Panning* - since we're applying an offset to our object's depth we run the risk of objects appearing detached from their shadows. This is easily fixed, however, by activating front-face culling when generating the depth map. With these two fixes (and by making sure a new depth map is only generated when either the scene, or light moves, rather than every single frame), we end with the result below.

3.8. Creating a Skybox

3.8.1. Explored Approaches

The team implemented a skybox using a *Cubemap* - A texture containing 6 individual 2D textures that each form one side of a textured cube. In order to do this, in our Skybox class, we just generate a texture and bind it, then, calling *glTexImage2D* for each of our cube's faces we load the corresponding image (using stb for loading). To display the skybox we load our cubemap's textures into a 3D cube centered on the origin and spanning to the borders of our scene using our *Skybox.shader*. The end result was the following.

3.8.2. Final Implementation

For our final implementation, however, we couldn't just use a generic skybox texture. While serviceable, what we wanted was to recreate the room from the classic scene, a search online provided no good usable results, considering the original movie had limited views of the room itself, and as such, it was apparent that the group would have to make its own skybox.
Thankfully, one member of the team remembered that a videogame called "Kingdom Hearts 2" had the whole room rendered in 3D, so a total of 52 screenshoots were taken of the room, and then stitched together in Photoshop to achieve the final skybox:

4. Post-Mortem

4.1. What went well?

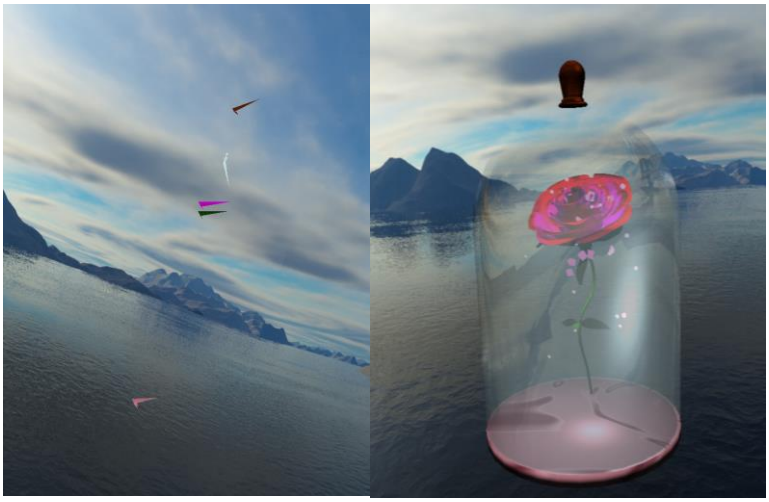
All in all the team was ecstatic with the final look of the project. We believe we managed to produce something we were truly proud of, having been able to strike a good balance between the classic animated movie's cartoon look alongside the more realistic aesthetic of the live-action remake. The team worked well together, and a sense of help and comradeship was felt throughout the entire process of development. Pulling from each of the member's strengths and preferences we managed to organize our work ahead of time and follow it thoroughly, with the assurance that if any of us hit a blockade, someone would come to help and brainstorm solutions. Our initial roadmap and project status tracking allowed us to work continuously, always knowing

what had to be done, and when it had to be done. Our management inclusively allowed us to enjoy a (much needed) week of rest during the Christmas time and still have everything done in time.

4.2. **What did not go so well?**

With that being said, obviously not everything went well. Although we managed to accomplish our main challenges, there were additional things the team wanted to but did not have time (due to other projects and exams) to implement, such as a volumetric fog around the rose, or light emanating from the particles. Other than that, the fact that we had to restart our particle system and redo our scene nodes also hindered our development by adding some unexpected overhead, and we did face some problems due to having used our own Blender models due to issues with the Normals and Faces generations (UV Mapping).

Besides this, one of the teammates, Henrique Gaspar, had to deal with a problem that impaired his development of this application. The problem was related to the use of an AMD video graphics card, instead of an Nvidia one, that led to the drawing of models only outputting triangles instead of the models themselves:



Upon showing the problem to the professor, all given possible fixes were tried, but none was successful, making the member have to develop his part on an external solution and then apply it to this current project.

4.3. **Lessons learned**

The main lesson we learned from this entire process was probably the desperation one feels when programming in OpenGL, making changes, and not seeing anything happening (whilst at the same time, nothing raising any errors in terms of compilation, linkage, or execution). Truly, bug spotting and fixing in OpenGL (both in terms of code and shader programming), is a dreadful task. When everything works, the visual output produced can be one of the most rewarding feelings, but when it doesn’t, it's easy to feel frustrated and want to just give up on everything then and there. A lot of

patience had to be learned. Other than that, we also found some really good OpenGL learning resources that supplemented the theory we learned during class.

- References

LearnOpenGL - Blending, https://learnopengl.com/Advanced-OpenGL/Blending

Norbet Nopper - Glass shader example, https://github.com/McNopper/OpenGL/tree/master/Example11/shader

Wikipedia - Refractive Index, https://en.wikipedia.org/wiki/Refractive_index

Wikipedia - Schlick's Approximation, https://en.wikipedia.org/wiki/Schlick%27s_approximation

Wikipedia - Fresnel Equations, https://en.wikipedia.org/wiki/Fresnel_equations

LearnOpenGL - Shadow Mapping, https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping

Yan " The Cherno" Chernikov - Making a Particle System in ONE HOUR! (No Engine), https://www.youtube.com/watch?v=GK0jHlv3e3w

LearnOpenGL - CubeMaps, https://learnopengl.com/Advanced-OpenGL/Cubemaps