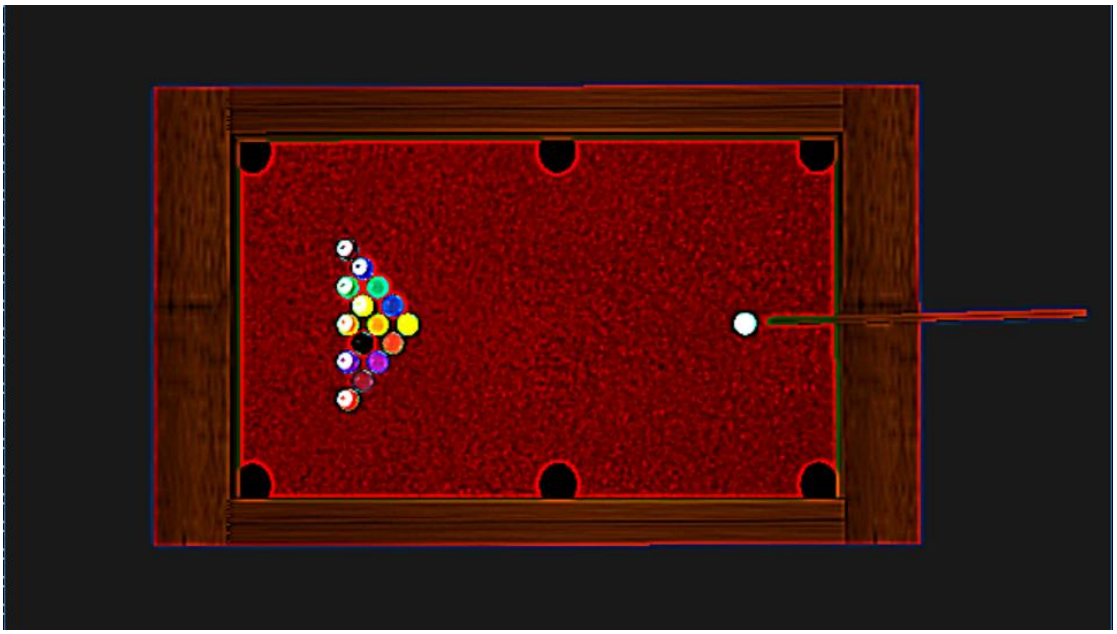
Computer Graphics for Games 2016

Snooker on Fire

Using a basic physics engine, lighting model, animations, and textures, the main concept of this project is to provide a semi-realistic simulation of a snooker game, with a table, balls and a cue.

It is a game of snooker, where everything is on **FIRE**! Well… maybe not *everything*. More like nothing... They always say, it's the thought that counts. Right?
(obs: the physics of the pool balls falling into the holes was not implemented, so technically is not really a "game" per say)



Vinicius Bruch Zuchi	**Ricardo Pereira**	**Marcelo Silva**	**Ricardo Silva**
86055	78051	87899	65937
MEIC-A	MEIC-T	MEIC-T	MEIC-T
vinicius.b.zuchi@gmail.com	ricardo.g.pereira@ist.utl.pt	marc.silva90@gmail.com	ricardo.filipe.f.silva@gmail.com

Abstract

The main concept is to simulate in **real-time**, a **game of snooker** (without the scoring part).

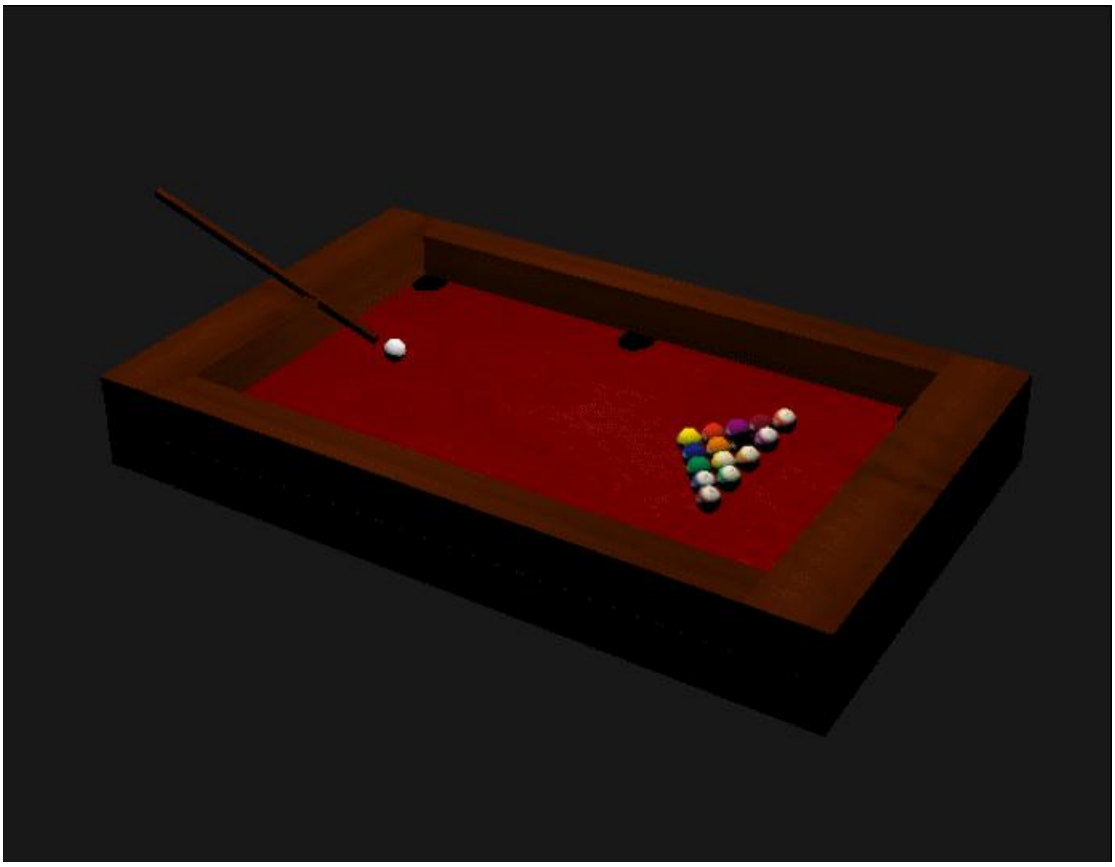
In order to do so, we need a scene. Since snooker needs collisions, there's a division between the physics and the visual engine. Physics engine performs every computation needed to move and collide the balls with each other and the table, while Visual engine renders the scene.

The scene itself is organized in a graph with multiple nodes (objects to be rendered), each one loaded from a **obj** file and with a **bmp** texture attached (only wood differs a little, since noise it's applied to give a realistic look).

It's also possible to manipulate the cue (rotate around the white ball) and performing a shot that will apply force to the white ball (visible via an animation).

There's also light in the scene (**Blinn-Phong**) to make it more realistic (observable since everything below the table is dark).

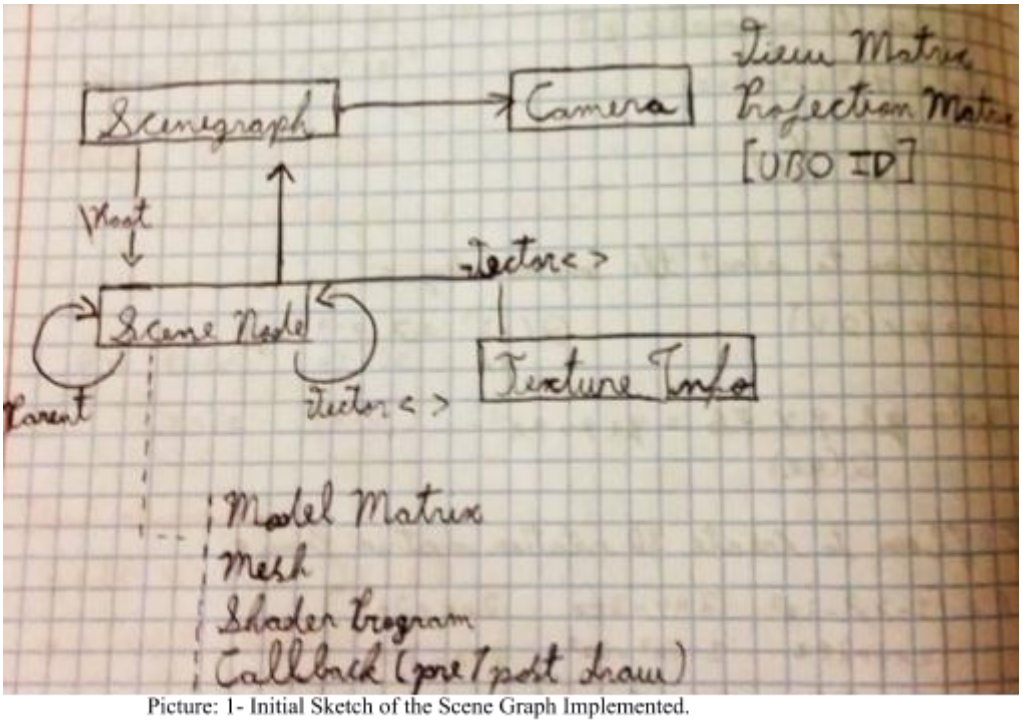
Finally, it's possible to add a **sepia** effect to the scene and take **snapshots** to save the best moments.



1. Concept

The initial concept was to provide a realistic simulation of one shot of a snooker game. This mean that our initial intention was to focus on only doing one shot with the cue and after that, restart the simulation. This part of the development went better than expected though, we managed to extend the single shot to a continuous simulation, the only part of the snooker simulation that was not implemented was the pocketing of the balls on the holes, thus all of them remain on the table.

In order to provide the simulation, we decided that the project had to have the following features implemented: a generic scene graph to handle the objects of the simulation; a physics model, to simulate the collision between the balls and the table walls, between the balls itselves, and between the white ball and the cue; a photorealistic lighting/shading using light over the pool table; a system for loading textures into the engine; interaction with the cue using the keyboard, to rotate it around the white ball in order to choose a desired angle to hit the ball from; the use of shading language to generate noise in order to provide a more realistic looking for the wood part of the table; and additionally, the possibility of taking a snapshot of any moment of the simulation with an option of sepia post processing effect.



Picture: 1- Initial Sketch of the Scene Graph Implemented.

2. Technical Challenges

2.1. Scene Graph [Vinícius]

Technically everyone did their own scene graph in their individual assignment but the one chosen to be used in the group project was Vinícius'.
In the conceptual phase of the implementation of the scene graph, the following goals were defined beforehand:

- Every scene graph has to have one reference to a camera object and a root node (responsible for initializing the drawing method)
- The nodes are independent of any shader program, except the root node.
- The nodes are independent of any mesh
- Each node has: its own Model matrix, a reference to its parent, a vector with references to its children, additionally, it can have a reference to a shader program, a reference to a mesh, and a reference to a texture.

With these features in mind, the scene graph was implemented. The biggest challenge with the scene graph was to keep it as generic as possible, and this was not entirely possible. To simplify it a little bit, some conventions were used. For instance, whichever shader program used for the nodes has to have uniform variables for a model matrix, texture, normals and vertex coordinates with the same name conventions used in the creation of the nodes. Also, whichever mesh that is used, the external file must be in OBJ extension.

Regarding the behaviour of the scene graph, a new node can be added to the graph both by another node, situation which the new added node automatically becomes a child of the node that has added it, or it can be added directly in the scene graph, in this case the new node becomes a child of the root node.

The graph is drawn recursively, starting by the root, it draws itself, and after that, it calls the draw method for each children it has. The model matrix is also calculated recursively, starting by the current node being draw, the model matrices are accumulated by getting the parent's model matrix and multiplying it by the current node's model matrix, the method is then called for the node's parent until the parent is the root.

The nodes have the option of not having a mesh, in this case their draw method will not be called. In order to provide proper manipulation of the scene graph, the nodes have methods for changing the model matrix, the shader program, the mesh, texture information, and the scene graph has a method for changing the camera object reference.

2.2. **Load External Meshes and Materials** [**Marcelo**]

The mesh loader is a simple parser for external files in OBJ extension. The technical challenge here was not the implementation of the mesh loader itself, but its integration of the rest of the engine. In order for the loader to work properly with the other parts of the engine, it was needed, as mentioned above, to have some name conventions to the meshes variables, and this names had to be used on the scene graph and on the shader programs as well.

For the textures the technical problem was to read the texture and send it to the fragment shader. To solve this issue it was created a texture loader on BMP image format where we create a texture id that can be used to set a Sampler2D uniform on the fragment shader.

2.3.**Physics and Collisions** [**Everyone**]

The main issue in this technical challenge was the abstraction of a physics engine and how to have it working with the visual engine (e.g how to update position, what

information should be used by each engine and how to pass it between them, how to calculate collisions correctly and apply ball rotation).

The first challenge was to create a virtual representation of the pool balls, so that the physics engine could update their positions as time progressed, and so that they could be visually represented in the correct locations by the visual engine.

The second challenge involved all the physics that each of the balls undergoes as time moves on. The adjustment of their directions, accelerations and positions, as well as checking whether any two balls where colliding with one another at any given time, or whether a balls was colliding with any of the 4 table walls.

These challenges will be properly explored further down the line, in the solutions section of these report.

2.4. **Object Manipulation** [**Ricardo S., Ricardo P.**]

For this challenge, we chose to focus on manipulating the pool cue in our scene.

More specifically, we focused on how to freely rotate the cue around the white ball (just like one would see a pool player do before taking the shot) and how to update the pool cue's position after each shot.

2.5. **Realistic/Stylized Solid Materials** [**Vinicius, Ricardo P.**]

At first we thought it was only needed a good texture in order to create a wood material. We later discovered that we had to approach it differently and implement noise into the texture to give it a different look. The main challenge was to create a shader that would use this noise texture to create the realistic look.

The main issue of this challenge was to test if the noise functions were actually doing anything, because most of them were not having any impact on the final output color of the objects, and this was the reason why we decided to try different approaches until finally finding something that was not completely intended, but it was giving results.

In order to make it work, we used a different fragment shader for all the objects that use wood which combines the texture, the lighting model and the noise function to give final output color.

The fragment shader was based on the implementation of the David Wolff's shader cookbook, which uses a function that defines a 2D region of the texture using the texture coordinates and defines a slice of the region. It then proceeds to perturb this surface and calculates an interpolation of two colors that represent the lightest wood color and the darkest wood color, both of this values can be determined.

2.6. **Scene Post Processing** [**Marcelo, Ricardo S.**]

The technical problem with post-processing was to create a framebuffer that would hold the relevant information of a frame, so that one could then take this information and translate it onto a texture, which could then be altered, if desired, and finally mapped onto a square that could be drawn on the screen.

The secondary aspect of this challenge revolved around the altering of said texture, in order to simulate a visual effect chosen by us (we settled on an approximation of the "sepia" tone filter).

2.7. **Photorealistic Lighting/Shading (Blinn Phong)** [**Marcelo**]

The biggest challenge of doing lightning was how to propagate a simple point of light between all the objects that are on the screen making the faces of the object that doesn't receive any light darker and having some fading depending of the object distance.

2.8. **Animation** [**Ricardo S.**]

This challenge involved adding some sort of animation to the scene.

To fulfill this requirement, we decided that simulating the animation of the pool cue being pulled back, and then having it strike the white ball, would be sufficient for this purpose, and add some charm to our snooker game.

2.9. **Snapshot** [**Ricardo P.**]

The most technical problem with this challenge was the fact that we had to save the scene into a known format. This implies (other than writing the information in a file) to create the corresponding headers, necessary to each format, which could be complex.

3. **Proposed Solutions**

3.1. **Scene Graph**

Regarding the development process of the scene graph, there wasn't any drawbacks, the first solution thought for it was the only one tried, and the one that worked.

As mentioned on the previous sections, there was a conceptual phase where the elements of each node and each scene graph were thought. The first phase to do was to implement the mesh loader, since the rest of the code would depend on it.

The mesh loader parses an OBJ file, with vertices, normals and textures coordinates, it then parses the faces of the OBJ file, counts how many of them exists, and creates three vectors (for textures, normals and vertices positions) that have the same size as the number of faces of the mesh, these vectors are used to send the data to the GPU to be used by the shaders. The mesh object has a draw method which is a wrapper to the "draw vertices" method call from openGL.

Next, the scene node class was implemented. Most of the methods of the class are simple setters and getters, except the method for getting the Model Matrix, which calculates it recursively by combining it with the model matrices from parent to parent until it reaches the root node.

The references to the mesh and shader program are pointers to the respectives classes of this object types created for the engine. If there is no reference to a mesh

object then a flag “drawable” is set to false and the draw method for the node is skipped when drawing the scene graph, this flag can also be set to false even if the node has a reference to a mesh, this was useful, for instance, to make the cue do not appear around the whiteball when the shot was already made.

If the node doesn’t have a reference to a shader program, on the first call to the draw method, it will try to get the first shader program from a parent that it finds, this means, that the only node required to have a shader program is the root, because is the last node that the method searches for a program reference. If the root does not have a reference to a shader program, neither all of the parents does, the draw method will throw an exception.

On the draw method of each node, it will set the the referenced shader program to be used, set its model matrix and texture to the correspondents uniform variables, it will call its mesh’s draw method, if the drawable flag is set to true, and recursively draws its children by calling the draw method for each one of them.

The node object have a method for creating a new node of the graph, this method will automatically set the new node to be the child of the node which called it and add the new node the children vector of the calling node.

The scene graph class is relatively simple as most of the work is done by the nodes objects. It has a reference to a camera object and a root node, the latter being responsible for initializing the draw method of the scene. The graph has methods for changing the camera object, and most of the other methods are wrappers for the root node, such as the “create node” and the “draw” methods, which simply call the correspondent method of the root node.

3.2. **Physics and Collisions**

The physics engine was one of the first implementations made as a group.

To start, we created a structure to model each ball. We modelled the necessary attributes (radius, direction, acceleration, mass, etc) so that our physics engine could properly update the position of each ball as time progresses, and handle the situations where one ball would collide with another (or with the table). This “ball” object is also accessed by our graphical engine, after all the calculations have been complete, so that it can draw each ball in the updated position. These objects are also stored in a manager object, that allows us to more easily handle updating their positions. Our typical “physics” update cycle goes as follows:

1. Update each ball’s position based on its direction an acceleration
2. Calculate which balls are colliding with one another, and resolve those collisions.
3. Calculate if any ball is colliding with one of the four walls of the pool table, and resolve those collisions.

After the cycle finishes *(a physics cycle is considered finished when no ball is colliding with another, or when no ball is colliding with any of the table’s walls)*, the ball objects are then passed on to our visual engine, which draws them in their newly updated positions.

Moving on to discussing the actual collisions themselves, to determine whether two balls are colliding, we first check if the distance between them is smaller than or equal

to the sum of their radius. If it is, then the balls are, technically, inside each other, which means they have collided. The collision itself is handled making use of the *"1-dimensional elastic collision equations"*. Lastly, and taking into account the new direction and acceleration of each ball, the balls are displaced, so that they are no longer colliding with one another.

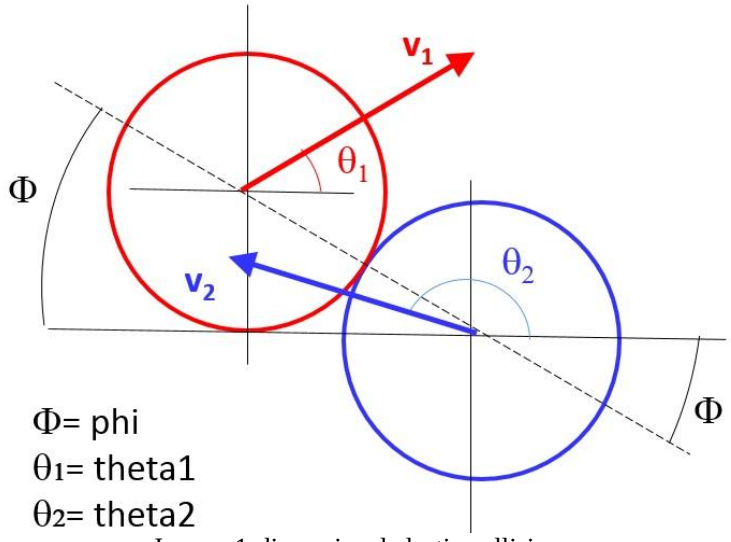


Image: 1-dimensional elastic collisions

As far as the collisions with the table walls is concerned, we opted to simplify the problem, and merely check whether a ball is past the "edge" position of each wall. If it is, and depending on its direction, the ball's position is updated, and the ball is, just as before, displaced so that it is no longer considered to be colliding with the wall.

3.3. **Object Manipulation**

The first order of business involved coming up with a solution of where exactly the pool cue would reside at each moment. We agreed that the simple solution would have the cue "follow" the white ball, since they were so closely related. This was accomplished by having the cue as a node of the white ball with the addition of a simple translation. However, this would look very odd when a shot was taking place, so we decided to hide the pool cue in between shots (very similarly to what happens in snooker/pool video games).

The next course of action was to do implement the actual manipulation of the cue. Taking the lessons learned early on in the course, we first positioned the tip at the origin. This meant that any transformations would have said tip as the reference point. From here on, it was fairly simple to add some pitch and yaw information to the cue, that would change according to the user input. The pitch would be controlled via the mouse, while the yaw would be relegated to the keyboard. There isn't any particular reason to have it not all be either mouse or keyboard controlled. The dual peripheral control was just a convention that we established at the moment, and never got around to review.

Picture: 2- Cue rotated

We also added some hard min. and max. caps on the pitch values, mostly to prevent the pool cue from going through the table.
Lastly, we took care to save the yaw angle so that it could be used when determining the direction in which the white ball would be launched. This links directly to the manipulation of the object feeding into the physics calculations.

3.4. **Realistic/Stylized Solid Materials**

We've tried multiple shader implementations and as mentioned on the section 2, the biggest problem we had was to debug the noise function since most of them apparently didn't have any impact whatsoever on the final output color of the vertices.

One of our first tries was to create a fragment shader that would generate random noise with a function and would use the result on the wood objects. This approach didn't work out because we didn't manage to make the noise function to work properly, regardless of the parameters used. The picture below demonstrates the expected behaviour of this tried approach.

We stopped on the only one that created actual visible noise. The one we ended up using is based on the David Wolff's shader cookbook as stated before.

The main idea is:

1. Have a noise texture (we used a normal texture instead)
2. Create a new shader to be used only by the walls (wood)

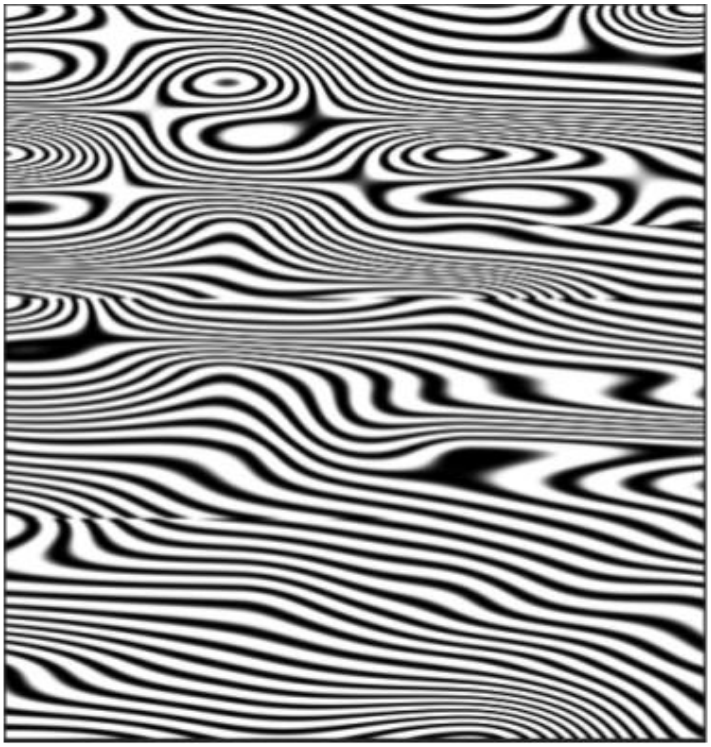
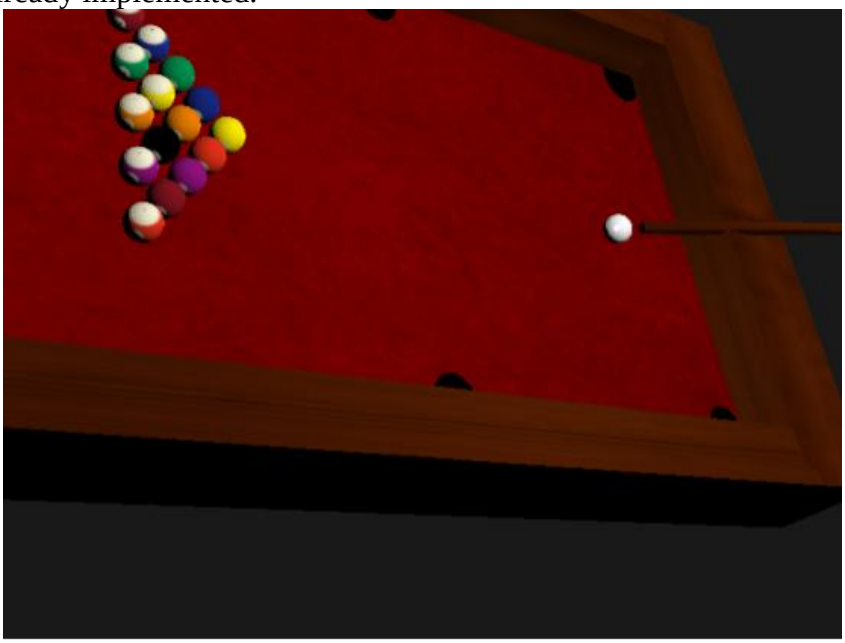


Image: 2- The original intended noise texture that was we have tried to implement

3. Implement the shader that generates noise based on light wood and dark wood colors

The shader itself works, as previously explained, by interpolating between those two colors within a certain distance defined by a slice of the texture.
By using a slice and calculating this distance, we then proceed to give a new color based on the position (a mix between the color of the texture, the light wood and the dark wood), which will be the final color. This was combined with the lighting factor that was already implemented.



Picture: 3- Final result (wood getting darked on the left)

3.5. **Scene Post Processing**

To achieve the desired “sepia” effect, a first approach was to simply introduce a flag that would tell the engine to change the shader being used to process the scene according to the situation. However, we soon realised that this had little to do with post-processing, so we chose to abandon this avenue and look for a more appropriate solution. This lead us to the use of framebuffers and rendering the whole scene into a texture that could then be mapped onto a quad.

We first created a framebuffer object that would be used as the render target for our scene. We aso took care to create an allocate space for a 2d texture, to be used as the color buffer of our newly created framebuffer.

Picture: 4- Sepia effect

After processing the scene, we took the resulting texture from the FBO and mapped it into a quad. This meant that we now had a quad that could be rendered to fill the whole screen. Without any further steps, the end result would be the same has just rendering the scene directly onto the screen. However, having the quad at our disposal means that now we can render it with a different shader applied to it.

With this in mind, and to finalise the process, we created a simple fragment shader that replicated the “sepia” tone filter effect (to the best of our ability). This shader was then used to render the quad. The final result was our snooker game with a “sepia” filter on top.

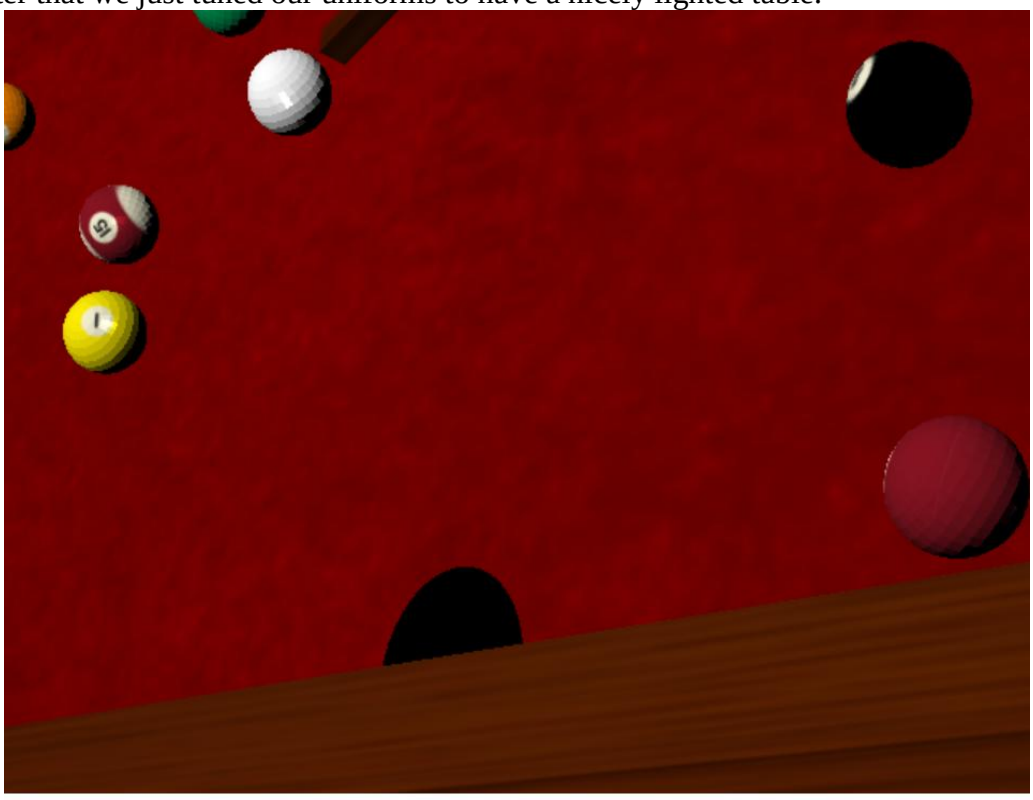
3.6. **Photorealistic Lighting/Shading (Blinn Phong)**

The first approach at lightning was trying the follow the blinn-phong reflection model, so it was first created 4 vectors that would be defined as a uniform in our shaders:

LightPosition - The position of the light in the world.
AmbientProduct - The color of the ambient light, it was defined as a pure black, so in our case there was really no ambient lightning.
SpecularProduct - The highlight color from from the specular reflection.
DiffuseProduct - The diffuse color that will be used for the diffuse reflection.

This uniforms are defined and set at each frame before the drawing of all the meshes to be able to calculate the lightning accordingly to blinn-phong function.
After the definition on the fragment shader and when trying to draw the scene we faced a problem, the snooker table was not illuminating correctly, half of the table was with the correct lightning and the other was pitch black. We found out that our meshes had defective normals and we had to create new ones.

After that we just tuned our uniforms to have a nicely lighted table.



Picture : Specular lightning effect and fading.

3.7. **Animation**

As a first approach, we tried to have the animation play out as the user initiated the shot via the keyboard input. However, this made it difficult to properly align the animation with the shot. There was also the issue with having the white ball immediately begin its movement when the corresponding input key was released,

which then meant that the pool cue would vanish, since the shot was considered to be "in progress".
To solve this issue, we decided to delay the shot until the animation of the cue had finished. This guaranteed that the ball would start moving at the proper time, and that the full animation would play out before the pool cue vanished.
Lastly, to further perfect the effect, we took into account the "speed" at which the pool cue was "pulled back" by the user, and then sped up the inverse animation. This gave us an effect that was fairly close to what one can witness during an actual pool shot.

3.8. **Snapshot**

Snapshot works for two file formats (**tga** and **bmp**). Both work by reading the buffer (**readPixels**) and writing a file with the respective headers and the scene information.
The only thing that differs between the two are the headers needed (each format has a specific structure and metadata).
During the process, tga snapshot remained almost the same while bmp had 2 versions (mostly equal) because there was a small bug that was only later corrected.
The only difference between them, is that the chosen one uses c++ libraries to create bmp headers (**BITMAPFILEHEADER** and **BITMAPINFOHEADER**),
The version uploaded, mistakenly, had the first version in it (has a bug that causes the snapshot not to work with some window sizes). After the submission date, we detected the mistake and corrected it (only for bmp, tga still has a variation of this bug).
When it comes to naming the file, the class has a counter (starting at 1) for the pictures taken. Each file is named with the following logic: **"snapshot_x"** with x being the counter. There are more interesting approaches to name the files (so it wouldn't reset and rewrite them each time the application starts) but the best way would be to use an external library to count the number of files in the folder and use that number has the starting counter (counters if we could count by format). We rejected this solution and kept it simple to reduce dependencies.

4. **Post-Mortem**

4.1. **What went well?**

When it comes to management, instead of following a schedule and dividing the work, we went for a different approach. When somebody had the time, they would choose a challenge and work on it (mostly following the order needed). This happened to work out great since everyone ended up focusing on a task they thought they could do and organized their time better (each element had more work to do aside from this project, so there were peaks of work for each person). In other words, everyone had a little bit of freedom to what, when and how they were working.

Getting a little more technical now, there were a few things that went better than expecting.
Vinícius' individual engine was well structured and abstract enough for us to work on without any problems. Everything needed was easy to find and use, so everyone managed to work with it without having to read the code or taking to long to

understand it.
This was visible since the second checkpoint, when we were asked to implement textures and we only had to add it to the sceneNode without changing the abstraction already made.

Finally, our collisions ended up being more polished than we could've thought. Our physics changed a lot throughout the implementation, and every new try was having its own problems (the final being the separation of balls that were stuck together). But we eventually got it right and everything works smoothly.

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

One of the most problematic things that happened, was the fact that every member had work to finish for other projects, around the same time as this deadline. Because of that, we had few days to come up with solutions for the challenges missing. We managed to end it in time, but we didn't have time to explore multiple solutions and test which one was better (first one that worked, stayed).
There were also challenges that we found out that had to be changed or reworked because it wasn't the right approach, which combined with the final days could've gone wrong.

One of the final challenges that we worked on was the wood, since for a long time we thought it was only needed a realistic texture. Because of that, as said before, we didn't have time to explore alternatives or improve the look. Due to the lack of time we ended up using a simple texture and a shader to generate noise above it, instead of using a noise texture, as it should be.
There was also a bug with the post-processing implementation. In essence, there was an oversight while implementing the framebuffer, and its size was not updated after a viewport rescale. This oversight was only caught and corrected after the submission of the final version.

4.3. **Lessons learned**

If it was possible to redo the team project from the very beginning, we would probably decide to take a more balanced approach to the problem.

Initially we focused quite heavily on the physics aspect, trying to get the interaction between the objects to be as smooth as it could be, while, sadly, postponing the work on the other aspects of the project, such as the lighting model, realistic textures and other minor challenges. We realise that that it would have been more beneficial to have began working on all the challenges, in order to have something that would allow us to gather a more thorough and balanced feedback on checkpoint 2.

In addition, should we have known how heavy the workload each and all team members would have been under during the more crucial portions of the assignment, we might have tried to negotiate, or at least seeked better advice, as to what challenges would have been more beneficial to chase after, given every team member's tight schedule.

5. References

Wood Noise:
Wolff, David. (2013). OpenGL 4 Shading Language Cookbook. pp.275-279.
https://thebookofshaders.com/edit.php#11/wood.frag

Snapshots:
http://www.david-amador.com/2012/09/how-to-take-screenshot-in-opengl/
https://www.gamedev.net/topic/203986-opengl-screenshot/

Post-Processing:
https://learnopengl.com/#!Advanced-OpenGL/Framebuffers

Blinn-Phong:
https://en.wikipedia.org/wiki/Blinn%E2%80%93Phong_shading_model