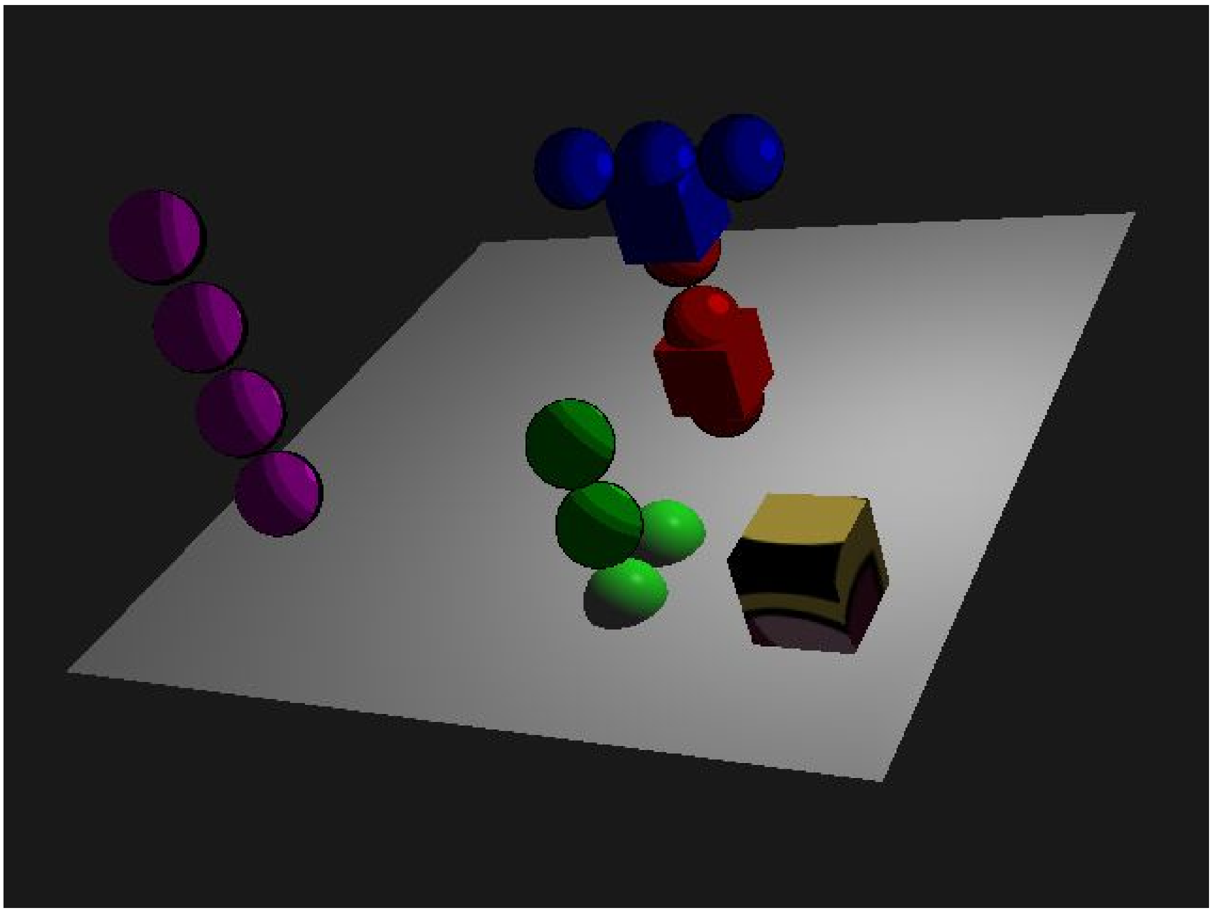
Computer Graphics for Games 2020

OpenGL Photo Mode

An interactive photo mode where you can set objects around, photograph them, and try out new filters.



Francisco Mendes, 87719 MEIC, francisco.andrade.mendes@tecnico.ulisboa.pt

Lucas Silva, 96962 MEIC, lucaslsilva@tecnico.ulisboa.pt

Luís Vital 97934 MEIC, luis.vital@tecnico.ulisboa.pt

Carlos Marques, 98639 MEIC, carlos.a.marques@tecnico.ulisboa.pt

Abstract

An interactive photography tool using OpenGL. A software where one can import models, build a set, set the blocking of the scene, and take snapshots. A variety of image filters are also available, to give the images an extra flair. Also, progress is saved in between sections, with users being able to reload their scene and easily resume their work.

1. Concept

We were inspired by video game photo modes, more specifically like in Kingdom Hearts III, where the player is given a scene, in which he can place various characters and objects, move and aim the camera and then take a snapshot they can save to the console. There are also various filters and stickers the player can add to the image.

Our project is similar. There is a scene with various objects the user can rearrange at will. When he closes the program, it will save where the objects were and reload them correctly the next time it is booted up. The player can also apply various filters to the render, providing different color schemes or other types of fun image distortions to the scene.

The user can move the camera around, aiming it and zooming it in on points of interest, and press a button to save a snapshot directly to their file system.

These objects, in addition to being movable as well, will be lit and cast shadows. The user will even be able to toggle between a photorealistic Phong shading style and a cartoony Cel shading style for the scene.

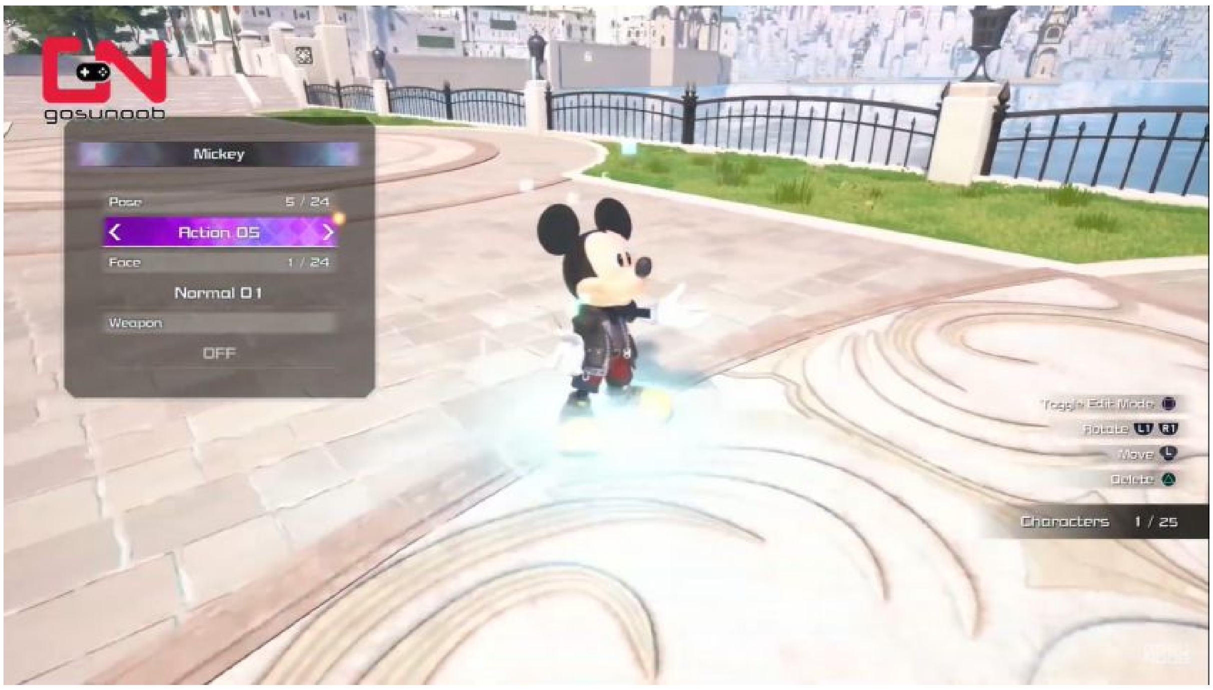


Figure 1: The player places a Mickey Mouse model in the Kingdom Hearts 3 photo mode. The user will be able to place and move objects similarly as well. (GosuNoob 2020)

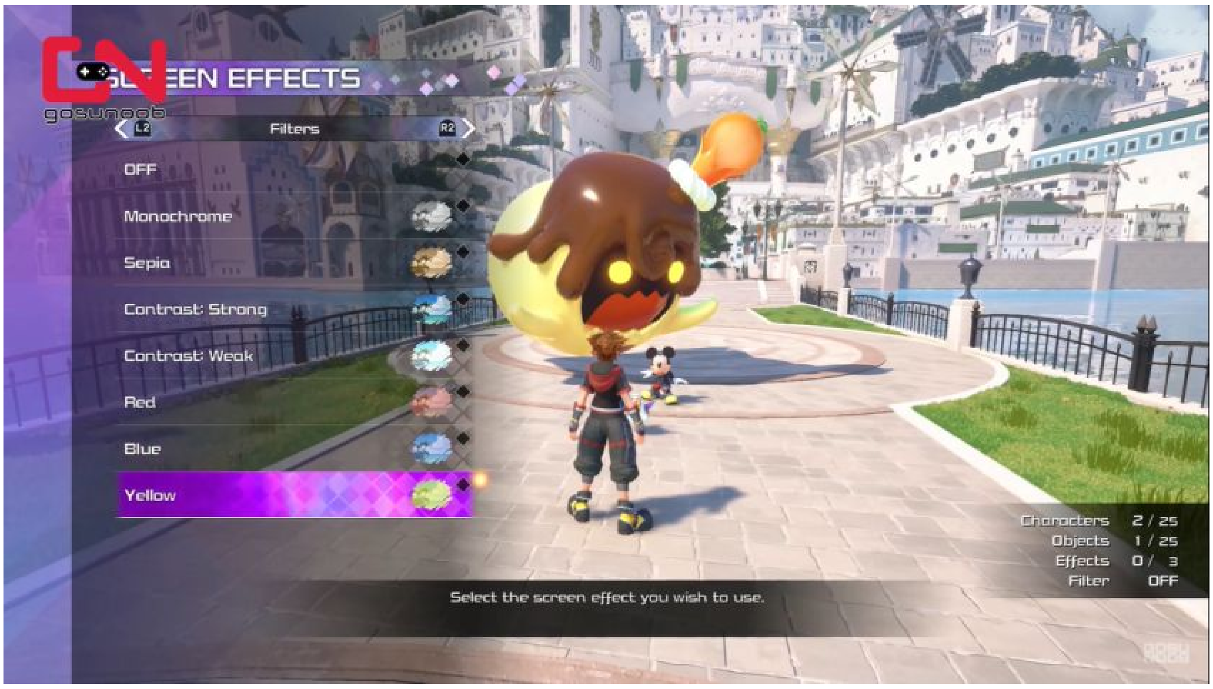


Figure 2: The player picks a filter to apply to the screen. Our project will also have a variety of filters for the player to try out. (GosuNoob 2020)

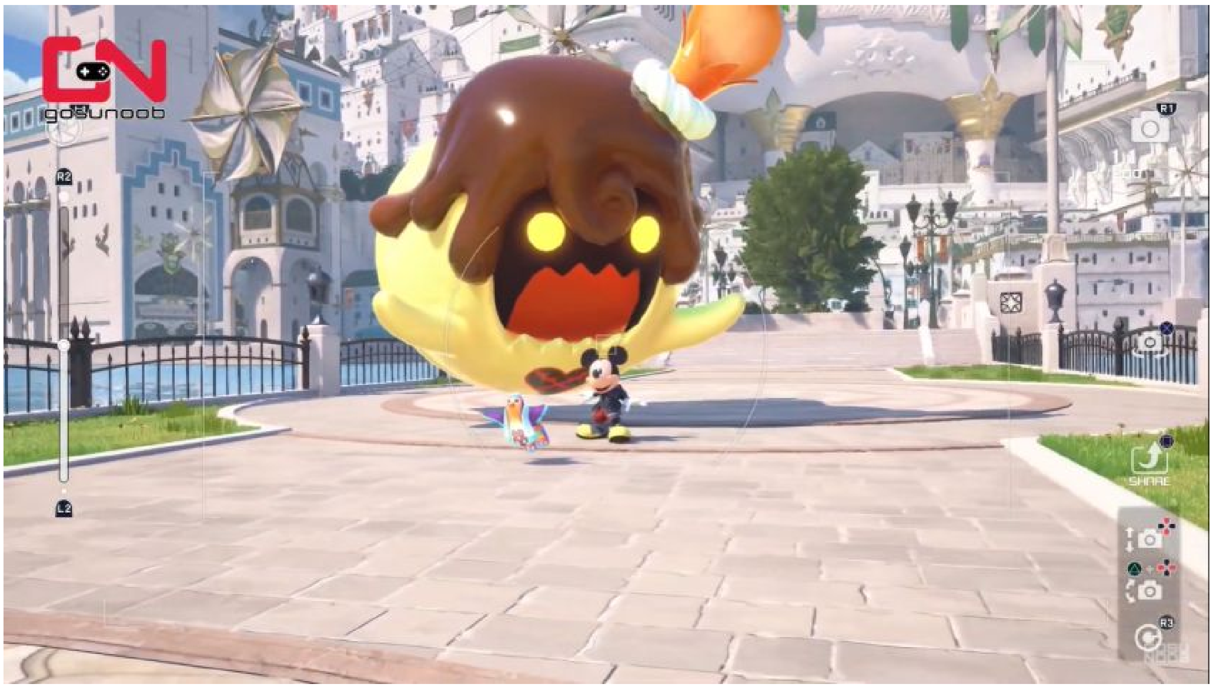


Figure 3: The player turns on photo mode, and aims the camera. He can still also move around/strafe with the left stick while aiming with the right stick and using the triggers to zoom the camera. While our project isn't using a game controller, it will feature similar degrees of movement for the camera, allowing the user to get all the shots he wants. (GosuNoob 2020)

2. Technical Challenges

2.1. Saving a snapshot of the screen to an image file [Carlos Marques]

This challenge was the simplest of them all, at least in concept. The goal was to save the user the trouble of using the Print Screen button or the Snip & Sketch tool on Windows 10.
We didn't want to force someone to not only manually photograph the correct area of the screen (in the cases where the program isn't running on the full screen) but also having to go out of their way to take the photo ready to be pasted and actually save it to a file.
As such, we aimed to have a button that took a snapshot of the entire screen and automatically saved it to a folder.

2.2. Scene post-processing through render targets, e.g. using multiple image filters [Carlos Marques, Francisco Mendes]

The goal here was to apply a filter to the scene, such as making the whole image grayscale, inverting the colors, or adding a blur effect. To achieve this we would need to do post-processing, meaning whatever effects we applied to the scene it had to be after it was rendered.
As such the project will need two render targets, a base one, with the imported objects and textures with their default colors and another one, where we apply the filter using a shader different from the default one that gives the image the desired effect.

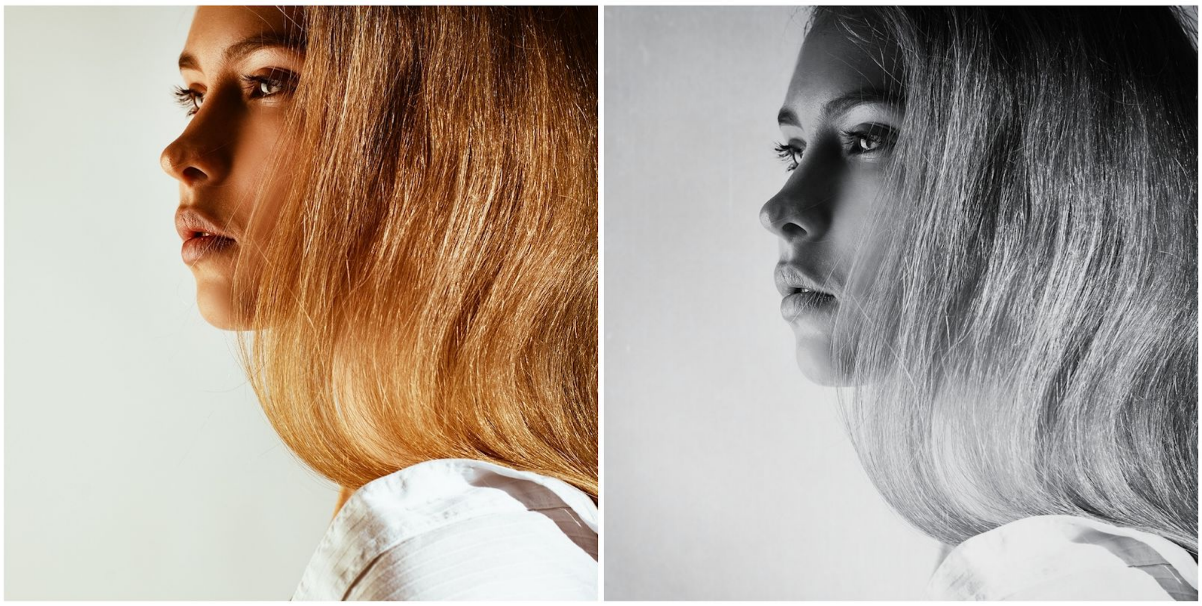


Figure 3: Example of a grayscale filter being applied to an image. This is from the online photo editing tool Fotor. (Abbey 2020)

2.3. Generic scene graph handling hierarchical drawing (e.g. matrices, shaders, textures [Francisco Mendes]

The goal for this challenge was to allow for the creation of arbitrary scenes by combining small pieces into a larger whole. The scene graph is composed of multiple objects, each having specific attributes, like it’s color, mesh, or texture, if any, as well as attributes that can be inherited from it’s parent object, like transformations and shaders.

The creation of arbitrary scenes would be done via a programmatic interface, which has access to more features and more precise control over the scene. However some control is possible at runtime, objects can be created or destroyed anywhere in the scene, meshes, textures and shaders can be swapped at will, and new meshes and textures can be loaded and used.

2.4. A non-physically-based “photorealistic” lighting/shading model, e.g. Phong or Blinn-Phong model [Francisco Mendes]

For this feature an extra set of shaders was created that accounted for ambient, diffuse and specular light following the Phong shading model, providing a nice contrast to the cel shaders that were also implemented for this project.

2.5. A non-photorealistic lighting/shading model with silhouette in modern OpenGL, e.g. Cel or Gooch shading [Lucas Silva]

In contrast with the Phong shading that is also implemented in the project and where the lighting is smooth, the cel shading shows a distinct boundary between lighting intensities, giving the object a cartoonish look. This feature also includes a silhouette of the objects, also known as object inking, reinforcing the cartoon aspect.

2.6. Buffer based special effects such as shadows [Lucas Silva]

The objective of this feature was to change the rendering pipeline to also include a step where the lighting, beyond the simple shading method that uses the normals and the relative position of the object and the light source, was calculated taking into account the light obstruction caused by other objects present in the scene.

2.7. Picking and manipulating objects with mouse and keyboard [Luís Vital]

This feature revolves around being able to select the objects within the scene and manipulating it within the scene, moving them around, changing the meshes they use, textures and even the shaders, allowing the user to change them all individually as they wish.

2.8. Creating a format allowing to save/load the full scene, meshes, and materials [Luís Vital]

The objective for this feature involves being able to store the information of the scene, including the window, as well as objects and their respective meshes and materials, into a file that can then be loaded to restore the scene to how it was when it was saved.

3. Proposed solutions

3.1. Saving a snapshot of the screen to an image file

3.1.1. Explored approaches

Searching for a way to easily take a snapshot in *OpenGL* quickly turned up results. As basically all of the results suggested, we decided to use an external library. We ended up deciding on using *FreeImage* and basing ourselves on this example from *Stack Overflow* (huy 2011):

```
// Make the BYTE array, factor of 3 because it's RBG.
BYTE* pixels = new BYTE[3 * width * height];
glReadPixels(0, 0, width, height, GL_RGB, GL_UNSIGNED_BYTE, pixels);

// Convert to FreeImage format & save to file
FIBITMAP* image = FreeImage_ConvertFromRawBits(pixels, width, height, 3 * width,
24, 0x0000FF, 0xFF0000, 0x00FF00, false);
FreeImage_Save(FIF_BMP, image, "C:/test.bmp", 0);

// Free resources
FreeImage_Unload(image);
delete [] pixels;
```

To quickly explain it, first a *BYTE** for all the pixels of the image is allocated. The number of pixels (width X height) is multiplied by 3 as it's necessary to store each value of each pixel's RGB values.
Then the pixels from the screen are read using the *glReadPixels* method, specifying the start coordinates (0,0) and the size of the screen (width, height). It's also specified that the pixels are to be read in RGB format and come as unsigned bytes to be stored in the previously allocated *pixels*.
Those bits are then converted to a *FITBITMAP**, a bitmap format exclusive to *FreeImage*, and saved into a file by the *FreeImage_Save* method, which receives the format, the path and the number of flags additionally passed (in this case none). (FreeImage 2018, 18)
This approach had two problems though: the path for the file was always the same and it saved the file as a .bpm, which isn't that convenient. Most people would rather have a .png or even a .jpg.

3.1.2. Final implementation

The first thing done was to search the *FreeImage* API (FreeImage 2018, 20) for a method to save to PNG. All that was needed was to change the *FIF_BMP* flag to the *FIF_PNG* flag. Additionally instead of using *FreeImage_Save*, we used *FreeImage_SaveU*, which is an identical method, but it supports UNICODE filenames.

Since we're on Windows, that uses UTF-16, this was necessary. We also decided that every snapshot would be called "SnapshotX.png" where X starts at 1 and increases with every photo taken. So we had to create the correct name for each file (the *path* parameter shown above). So we have the following code:

```
const auto filename = "snapshot" + std::to_string(snap_num_) + ".png";
const auto path = snapshot_dir_ / filename;
```

where *snap_num_* is an integer that is an attribute to the Engine class where this method is and increases with each snapshot taken. *snapshot_dir_* is a *std::filesystem::path* and also an attribute of the Engine class. How that attribute is set will be explained in challenge 3.3 as that includes the initialization of the Engine class. For the purposes of this challenge, it leads to a folder called "CG_Snaps" that will store the files. We use simple concatenations to generate the right name for the file.
To get the width and height we create a constant from the *size_* attribute of the Engine class.
The only other changes are that instead of a *BYTE** we use a *std::vector<BYTE>* to store the pixels data, as it is more effective, and we have a *#ifdef* directive for debugging purposes, where we actually store the *bool* returned by *FreeImage_SaveU* and print the result. As such we have the method *snapshot* from the Engine class.
We call this method every time the F2 key is pressed. We check for this event in the *onKeyboardButton* method of the main *CGJProject* class.

3.2. Scene post-processing through render targets, e.g. using multiple image filters

3.2.1. Explored approaches

As we researched post processing in *OpenGL* it soon became clear what we had to do. We needed to create an additional framebuffer, that renders the scene to a quad that is the size of the screen. After, we needed to render that quad to the screen, this time with the shader corresponding to the filter we created. (LearnOpenGL, n.d.)

3.2.2. Final implementation

As such, we ended up creating a Filter class that handles setting up all the framebuffers of the project.
In the constructor of the Filter class, we set up all the buffers necessary to render to the quad, whose vertices are defined in the constant *quad_vertices*.
We use *glGenTexture* to use as an image for the color buffer of the framebuffer, so we can use its result to apply the filter.
Since we do this for every filter, we do in fact set up framebuffers for all of them, even though we only activate one at a time (though this concept could be expanded upon, by having more quads, and applying a filter to each, for example, to create fun combinations).
All the buffers needed to render to the screen are also handled here.
The main methods in this class are the *Bind* and *Finish* methods. When a filter is selected as the active one, *Bind* is called before the render and *Finish* after.
Bind changes the rendering framebuffer to the one corresponding to the filter we have selected. This renders the scene as normal to the quad.

Finish undoes that and sets the framebuffer back to the one that draws the quad to the screen. Here we used the filter shaders to apply the effect to the scene.
So to summarize, the rendering steps are:

1. Bind to the filter's framebuffer.
2. Render scene normally.
3. Unbind framebuffer (switching back to the default framebuffer).
4. Use color buffer texture from normal framebuffer in postprocessing shader.
5. Render quad of screen-size as output of postprocessing shader.

Each filter is a different frag shader, applying a different effect to the quad as it is rendered to the screen. They're all in the *Shaders\Filters* folder. So let's go over them:

1. *Red, Blue, Green tints (red_frag, blue_frag, green_frag).*

These 3 work identically. We simply get the FragColor of the texture and multiply the two lesser components by 0.2. So for example for the Red tint, we get the RGB values, and reduce G and B by 20% while keeping R the same. This gives the image a tint corresponding to that color.

2. *Grayscale (grayscale_frag)*

The key for a grayscale effect is to have each component of the RGB carry the same weight, so no color overwhelms any other. That's why [250, 0, 0] is red but [100,100,100] is a shade of grey. A good way to accomplish this is to simply get the average of the RGB values and set them all to that average. (Kristiansen 2013)
But there's an even more correct way. The human eye perceives each of these three colors in a different way. Hence why HD TV's for example, calculate the Luma of each pixel differently:

$$Y = 0.2126R + 0.7152G + 0.0722B$$

Figure 4: Formula used by TV's to calculate the Luma of each pixel. The eye is best at noticing green, so that color contributes more to the perceived brightness of each pixel, while blue contributes the least. (Wikipedia, n.d.)
As such we used this formula to set the FragColor.

3. *Sepia (sepia_frag)*

Sepia is a brownish color and tinting photos with it is very prevalent in photography. It gives it an old and nostalgic feel to it. It comes from an old process in the darkroom where photographs were tinted this way to increase their longevity. (Photoancestry 2021) Since there are a lot of tones of brown that can be used, there isn't really a mathematical justification for how to convert it as there is for grayscale. However there is a widely accepted formula to get the RGB values that we used: (Dyclassroom, n.d.)

```
lowp float
outputRed=(textureColor.x*.393)+(textureColor.y*.769)+(textureColor.z*.18
9);
lowp float
outputGreen=(textureColor.x*.349)+(textureColor.y*.686)+(textureColor.z*.
168);
```

```
lowp float
outputBlue=(textureColor.x*.272)+(textureColor.y*.534)+(textureColor.z*.1
31);
FragColor= vec4(outputRed, outputGreen, outputBlue, 1.0);
```

4. Inverted (invert_frag)

This one is very self explanatory. We just take the change the RGB values to 1 - RGB, thus changing the colors to their opposite. (LearnOpenGL, n.d.)

5. Sharpen (sharpen_frag)

Some of the effects are accomplished using kernel effects. By creating a kernel of the pixel and the 8 pixels surrounding it, we can take a small area of the texture and apply a lot of effects. A kernel is supposed to sum to 1 when all the weights are added together, any other value and the colors end up brighter or darker than in the original texture. Additionally, a small offset is added to the texture coordinates so that we can get the pixels around the pixel we're reading.
For sharpening we use this kernel [2,2,2,2,-15,2,2,2,2], where we boost the pixels on the edges and make the center one more muted. Then we get texture values (+ offset) for all 9 pixels, we multiply each by the corresponding kernel value, and we add it all up to from one vector, that will become the new FragColor.

6. *Edge Detection (edge_frag)*

This one is fairly similar to sharpen with the matrix [1,1,1,1,-8,1,1,1,1]. The difference with this one is we don't boost the edges, we keep them the same, we just mute the inside part of it. This highlights the edges between different colors. Other than the matrix the code is identical.

7. *Emboss (emboss_frag)*

Embossing is when we give an image the illusion of depth, and to do this, we make apparent the difference in pixels across a line. We use the [-2,-1,0.-1,1,1,0,1,2] matrix, so it's a line from top left to bottom right. This makes the image look bumpy and every color looks like it's at a different depth. (Powell, n.d.)

8. *Blur (blur_frag)*

This is another kernel based effect. This time we use the matrix [1,2,1,2,4,2,1,2,1]. When things are blurry you notice them more at the center, less to the sides, and even less to the corners of your vision. But this doesn't add up to 16, so we divide each value by 16, and we get a correct matrix. (LearnOpenGL, n.d.)

9. *Sketch (sketch_frag)*

This one is a form of Sobel edge detection, which is similar to the edge detection filter we also have (we actually had a Sobel filter working, but since it was so close to the edge detection one, we figured it was redundant). Basing ourselves on the code found at (Larson 2012). First we create a matrix of the dot product between RGB values of each of the 9 pixels and a vector, representing grayscale (with the same values we used in the grayscale filter). We actually tried it out with W = (1,1,1) instead and we found that the black lines ended up much more pronounced and unrealistic, so the grayscale makes everything look softer.
Sobeling is also a kernel effect, but in the code, we admittedly copied, they do it is a more roundabout way, creating a matrix of [im1p1, i0p1, ip1p1, im10, i00, ip10, im1m1, i0m1, ip1m1] (so top row p1, center is 0, bottom is m1, and left column is m1, center is 0, and right is p1). Afterwards, they create a vector of two dimensions were each

component is the product of that matrix with [-1,-2,-1,0,0,0,1,2,1] and [-1,0,1,-2,0,2,-1,0,1] respectively. This is because in sobelling the x coordinate increases in the right direction, while y increases to the bottom. At any point of the image the gradient is the length of this vector. (Wikipedia, n.d.)
Finally embossing usually makes the edges white and the rest black, but we want the reverse of it, to give that charcoal on paper look. So we reverse the target radiant, to reverse those colors. To apply the filter to the color we linearly interpolate, starting from the color on the texture all the way to the gradient at an intensity we set as 0.8.

10. Oil Painting (oilPainting_frag)

The next filter is also copied from (Larson 2012), and makes the image look like an oil painting. It's actually a Kuwahara filter (Wikipedia, n.d.). The basic idea is that around the pixel we have a square of *2a + 1* side, where *a* is the side of each quadrant.

a	a	a/b	b	b
a	a	a/b	b	b
a/c	a/c	a/b/ c/d	b/d	b/d
c	c	c/d	d	d
c	c	c/d	d	d

Figure X. A kuwahara square where a = 2. So each square has a side of size 3 and they overlap, with the center pixel belonging to all four squares.

This is because the code will pick the color of the central pixel based on which quadrant has less variance.
So what the code does first is create two arrays, *m* and *n,* that store the sums of the vector3 representing the pixel's RGB values. We could store all the values we read, but given we only need them to calculate the arithmetic mean, we will be adding them up right as we get them. We will use one to store the RGB values and another to store the squared RGB values (to clarify, that's the product of that vector with itself).
We also set the radius variable, which is equivalent to the *a* mentioned above. We used 5 as we felt that gave good results. In addition we also calculate n which is *(a+1)*(a+1)* because that's how many vector3 we will have added to each index of the array by the time we finish running the cycles. We have to count the overlapping squares too, hence the +1.
After we get to the end, we set a very large sigma. because we need to find the smallest one so we need a base to compare.
So for each quadrant, we divide the corresponding *m* value by *n*, meaning that now the index of the array stores the mean. Now to get the variance. Usually the square of the variance is the mean of the square of the deviation of each data point from the arithmetic mean. But this code actually uses this property:

$$\sqrt{\mathrm{E}[X^2] \quad (\mathrm{E}[X])^2}.$$

Where E is the estimated value, in this case the arithmetic mean. And since we want the variance, which is the square of this we don't use the root. So in case it wasn't clear, this is why we stored the square of each vector in a separate array. So now, to

get the variance we simply do *s[k]/n - m[k]*m[k]* (which as a reminder, *m[k]* now stores the average).
But this returns a vector, so to have a value we can compare, we add up all its components. Then we compare with the minimum sigma we had set before, and if it's smaller, change that value and set the frag color to the average of the correct quadrant.

3.3. Generic scene graph handling hierarchical drawing (e.g. matrices, shaders, textures

3.3.1. Explored approaches

The design process for this feature was split into three facets, how to store and manage the scene and assets, how to create scenes programmatically, and which parts would be controllable at runtime and how. We decided that we'd have a single camera and light source. The design stayed mostly the same along most of the project's development, but had to account for the lack of a proper user interface.

3.3.2. Final implementation

The ***Scene*** class owns the multiple assets (Meshes, Textures, Shaders and Filters), which, like the rest of the project, use ***R.A.I.I.*** to manage resources, like memory, files or OpenGL object handles. The scene also contains the root object and default shader and white texture.

Meshes are loaded from Wavefront obj. files via a two stage custom loader and parser in the ***MeshLoader*** class and ***createVao*** function and their corresponding OpenGL object handle is kept in the ***Mesh*** object. C++ move semantics and R.A.I.I. are used to ensure that the OpenGL object is properly disposed of and not duplicated, resulting in memory unsafe errors.
Textures follow a similar pattern: They are loaded by a ***TextureLoader***, using the ***FreeImage*** library to read and convert the texture file into a buffer, which the ***Texture*** class uses to create the OpenGl texture itself.

Each ***Object*** owns its children and is able to reference its parent, as well as a mesh, shaders and texture. Each object also has a local transformation and multiple attributes related to its color and lightning.
Since child objects are owned by their parents, that limits the scene graph to a tree, however, due to the separation of asset ownership from the objects that use them, reusing objects via more complex object graphs was not considered to be a worthy or useful feature. Each object is responsible for updating and cascading the default shaders and world transform to its children, as well as drawing itself if it has a mesh.

The scene can be created using the ***Scene::Builder*** in the ***setupScene*** callback, where meshes, textures and shaders can be loaded and emplaced, objects can be created and emplaced on their parents, via the many ***emplaceChild*** overloads, all properties of the objects are directly available and can be set to desired values. The light source and default camera positions are set and the filters created.

The runtime controls for the scene are handled via multiple ***Controllers***. These Controllers follow similar patterns: They all allow for bidirectional iteration of a collection, accessing the current element, if any is selected, along with specific interactions for some controllers:

- The ***ObjectController*** allows for creation, deletion and movement of objects, as well as navigating the object graph.
- The ***MeshController*** and ***TextureController*** allow for emplacing more of their specific assets from the corresponding Loader classes.
- The ***FileController*** can be set to scan for either mesh or texture resource files in their corresponding asset directories. The scanned files can then be iterated like in the other controllers. A file is then selected to create the appropriate Loader object to be used by the other controllers.

Along these there also exist the ***PipelineController*** and ***FilterController*** which allow for changing the shaders for the current object, and setting the active filter, respectively.
These functions are then bound to multiple keybindings in ***onKeyboardButton*** and used by users to manipulate the scene.

This functionality also intersects with the object manipulation feature via the ObjectController, the specifics of which will be mentioned in that feature's section.

3.4. A non-physically-based "photorealistic" lighting/shading model, e.g. Phong or Blinn-Phong model

3.4.1. Explored approaches

This feature was relatively straightforward, the Phong model was chosen in the project proposal and the implementation was rather simple but touched many parts of the project.

3.4.2. Final implementation

This model was implemented in its own shader, allowing for custom ambient and specular colors, as well as controlling how "shiny" the object should be, corresponding to how concentrated or diffuse the specular highlight is. These values are used alongside the object's main color in the fragment shader as such:

- For the ambient light, the *ambient color* is passed as is.
- For the diffuse lightning, which is brighter the larger the dot product between the normal and the direction of the light, the object's *main color* property is used.
- For the specular highlights, the dot product between the view direction and the reflection vector is calculated and raised to the *shininess factor*. This strength value is then multiplied by the *specular color*.
- All of the resulting colors are then added together and multiplied with the active texture.

Along with the shader, additional properties were added to the objects, indicating the values for the aforementioned inputs: the ambient color, specular color and shininess value, each defaulted to an appropriate value: the ambient color is slightly brighter than the background and the specular color and shininess grant a noticeable but not overwhelming highlight effect. These values can be customized by the programmatic api.

3.5. A non-photorealistic lighting/shading model with silhouette in modern OpenGL, e.g. Cel or Gooch shading

3.5.1. Explored approaches

This feature, as the previous one, was relatively straightforward, Cel shading, also known as toon shading, was the one that we chose to implement from Cel or Gooch shading.

3.5.2. Final implementation

In the same way as the other shading model used in this project, the Cel shading was implemented in its own shader.

This shading model is very simple, and is divided in two parts, the coloring of the object and its outlining.

To achieve the desired effects, the shader starts by calculating the dot product between the light direction and the normal of the object surface, the resulting value is the intensity of the light at that point. After that the intensity is parsed into steps, for example, if the initial intensity was higher than 0.95 it would be changed to 1, if it was higher than 0.5 but lower than 0.95 its final value would be 0.6 and so on. This intensity would then be multiplied by the color and active texture. This stepping of the light intensity gives the effect of distinct light intensities instead of a smooth shading.

To create the outlining effect the approach was very similar, but this time the dot product was between the normal and the viewing direction, and only had a limit, where if the value of the dot product was lower than 0.3 the intensity was 0. This meant that only near the edge of the object where the normal and the viewing direction were at almost 90 degrees of each other the render color was black.

3.6. Buffer based special effects such as shadows

3.6.1. Explored approaches

To implement this feature we had two options, both being based on calculating depth maps from the point of view of the light source, but on was a simple texture depth map and the other being a cube texture, we ended up implementing the second because our scene light was a point light and the first option only works for directional lights.

3.6.2. Final implementation

To create this feature we inserted a new step in the rendering pipeline, right before rendering the scene. In this step we render the scene, but to a texture, and instead of being from the viewpoint of the camera we do it from the viewpoint of the light source, also, the rendering is not about the color of the objects but about the z distance. As our light is a point light we need to do this in every direction, so we use a cube map texture where each face represents the direction of the light.

After we have the shadow mapping texture, we render the scene in an almost normal way, with the difference being that when we go to calculate the color first we compare the fragment distance to the light source with the value of the corresponding point in the shadow texture, if the distance has a bigger value that means that there is an object closer to the light and that the fragment is in shadow, and so only having the color coming from the color texture and the ambient light.

Although we understood the process for creating this special effect we did not manage to make it functional.

3.7. Picking and manipulating objects with mouse and keyboard

3.7.1. Explored approaches

As the engine had the objects incorporated, a simple way to select and control them would be through a hierarchy of father/child, going from the whole of the objects, to the smallest parts that make it up. Each part eventually splits into multiple components that can be selected and manipulated individually.
An alternative would be selecting each object with the mouse, and manipulating them with a combination of the mouse, dragging it to alter the objects accordingly, while using the keyboard to change the way they're altered.
As the controllers for filters, meshes and shaders were already implemented, the first option was selected, implementing a controller for the objects which takes advantage of the way the other controllers were implemented.

3.7.2. Final implementation

The Engine of the application contains a controller for the objects within the scene, which allows manipulation of each individual object. The objects are set in a tree style, with the main branch being the entirety of the scene's objects, controlled by the "floor" plane. The Plane can be controlled by immediately pressing the U or I key and then manipulated with multiple keys on the keyboard, which allow it to be moved, its mesh changed, textures and shader.
The Tetris construct on the plane can be accessed with the T key, followed by the U or I keys, being possible to be manipulated in the same way as the plane. The user can return to the plane, and as a result, the entirety of the objects, with the Y key.
Using the T and U/I combination while controlling the construct allows the user to control the individual coloured blocks, with the U and I keys allowing the user to switch to the previous and next coloured block respectively. Pressing the U or I key while the first or last coloured block respectively is selected causes no block to be selected.
Finally, using the T and U combination while controlling a coloured block allows the user to manipulate the individual objects within the specified block.
At any time, the user can create new objects in the branch they're currently controlling, with the C key, creating a new object which takes up the first position of the branch. Additionally, pressing the X key while controlling an object deletes said object. Assuming the branch contains other objects, the user can press the U key to switch to another one. Pressing the X key when no object is selected causes the application to crash.
Object manipulation can only be done with the keyboard, with the WASD and QE keys for movement, the LÇ keys for mesh changes, JK for texture changes and GH for shader changes. The mouse is used for camera control.

3.8. Creating a format allowing to save/load the full scene, meshes, and materials

3.8.1. Explored approaches

The simplest option to implement a way to save and load the contents of the scene, including the meshes and materials, was to gather the information of the scene, along with the directories and names of the assets' files and store it in a single file that organises said information, and could later be accessed to pull that information. This can be done through JSON, however due to the lack of native support for it in C++, some additional way to add it is required.

3.8.2. Final implementation

An external library was used to implement JSON compatibility to C++, named "JSON for Modern C++" by Niels Lohmann. This library is exclusively used to store the information in JSON format and write and read files into JSON.
With this library, a class was created with two functions that are called to save or load information into the JSON files. Taking the scene's data as an argument, the information of the window, along with the filepath of the meshes, is stored in the JSON, and saved in a file that is written automatically. To load the information, the name of the JSON file is required first, and once it's obtained, the file is loaded, and the information is taken to be used by the application.
In the current implementation, JSON files that are saved are only saved with a timestamp of the system's current datetime, and only a single JSON file can be loaded, with no input for the filename in either the save or load functions. Additionally, while the save function saves the info, the load function only loads the info and displays it in the console.

4. Post-Mortem

4.1. What went well?

The completed features ended up really nice. The project's code is very well organized and structured and will allow for future expansion quite easily.
We met six of the eight challenges, and while it's not as many as we would have wished, we got pretty close to getting them all and learned a lot.
Taking snapshots was surprisingly easy, and it was good to get such a core component of the project done fairly quickly.
All the filters turned out great, with all of them working perfectly, and enhancing the scene. We managed to not only get simple stuff like color tints working, but also advanced stuff like the Sketch and Oil Painting filters.
We got both types of shading right, and they both look amazing. We were very pleased not just with how every model looked, but also that different objects in the scene could have different shadings, allowing for even wilder scenes.
And though the placement of objects in the scene could have been more intuitive, we got textures and meshes to load to the scene, we got them to spawn and we managed to be able to select any object and alter its texture, mesh or position, creating for some fun formations, as seen in the screenshots that come with the project.
Teamwork went smoothly without conflict, and we all learned a lot and that's the end goal of any group project, so overall the project was a success.

4.2. What did not go so well?

We were swamped with work from other projects, and failed to properly allocate time for this one. It was the key factor in our failure to meet all eight challenges. We also had some communication problems with the teachers, in regards to the first deadline (which were entirely our fault, as only one member of the group showed up for the final classes), which led to us implementing the first filter in a rushed manner, leaving less time to learn about it.

4.3. Lessons learned

We need to better manage our various projects from other disciplines, as this one ended up a bit neglected. While we enjoyed it, it is clear we did not devote enough time.
Still it was an educational experience, and we're all much more knowledgeable not just in OpenGL, but also the tech that makes our favorite games possible. In other disciplines we use game engines like Unity a lot and it's easy to take graphics for granted.

References

Abbey. 2020. "19 Popular Online Photo Filters to Make Your Shots Stunning." Fotor. https://www.fotor.com/blog/19-popular-online-photo-filters/.

Dyclassroom. n.d. "How to convert a color image into sepia image." Dyclassroom. Accessed 01 18, 2021. https://dyclassroom.com/image-processing-project/how-to-convert-a-color-image-into-sepia-image.

FreeImage. 2018. *FreeImage Documentation Library version 3.18.0*. N.p.: FreeImage.

GosuNoob. 2020. "KH3 NEW Data Greeting Feature - Custom Photo Mode Showcase - Kingdom Hearts 3 ReMind DLC." Youtube. https://youtu.be/Fn362HPXcew.

huy. 2011. "How to take screenshot in OpenGL." Stack Overflow. https://stackoverflow.com/questions/5844858/how-to-take-screenshot-in-opengl.

Kristiansen, George. 2013. "HLSL: Greyscale Shader Tutorial." gamedev.net. https://www.gamedev.net/articles/programming/graphics/hlsl-greyscale-shader-tutorial-r3263/.

Larson, Brad. 2012. "How can I do these image processing tasks using OpenGL ES 2.0 shaders?" Stack Overflow. https://stackoverflow.com/a/9402041.

LearnOpenGL. n.d. “Framebuffers.” LearnOpenGL. Accessed 01 17, 2021. https://learnopengl.com/Advanced-OpenGL/Framebuffers.

Photoancestry. 2021. “What is Sepia Toning in Old Photos?” Photoancestry. https://www.photoancestry.com/what-is-sepia.html.

Powell, Victor. n.d. “Image Kernels Explained Visually.” Explained Visually. Accessed 01 18, 2021. https://setosa.io/ev/image-kernels/.

Wikipedia. n.d. “Kuwahara Filter.” Wikipedia. Accessed 01 18, 2021. https://en.wikipedia.org/wiki/Kuwahara_filter.

Wikipedia. n.d. “Luma (video).” Wikipedia. Accessed 01 18, 2021. https://en.wikipedia.org/wiki/Luma_(video).

Wikipedia. n.d. “Sobel operator.” Wikipedia. Accessed 01 18, 2021. https://en.wikipedia.org/wiki/Sobel_operator.