
# Computer Graphics Final Project

## *MatCaps Creator*

**Author**

Alba Reinders Sánchez - s212729

December 19, 2021

# Contents

# List of Figures

# Listings

# 1 Introduction

*MatCap* shaders are effects used in 3D modelling tools to shade models by simply looking up into a texture that captures a material from an image of a sphere. There is no need to specify lighting and reflections on the model, because the shading of the sphere is applied to the surface of the model, by mapping the corresponding normals from the sphere surface onto that of the model surface.

The main goal of this final project is to develop a program, named *MatCaps Creator*, that allows users to create their own *MatCaps*. To carry this out, 3 different features are implemented: "Create texture", "Create *Matcap*" and "View *MatCap* on a 3D object". These functionalities are programmed using *WebGL* [1] and are available in a webpage format.

In the following sections it is explained the method followed, code snippets of the implementation are shown, as well as the results obtained and a final discussion of the limitations of *MatCaps* is given.

# 2 Method

A Material Capture, known as *MatCap*, is a technique that combines material properties and lighting information in an image. It is a simple and very powerful shader technique, because it does not calculate the way that the light sources interact with the surfaces of objects, as other shaders do, but it simulates these interactions encoding every aspect of a lighting setup and material properties into a single texture [2].

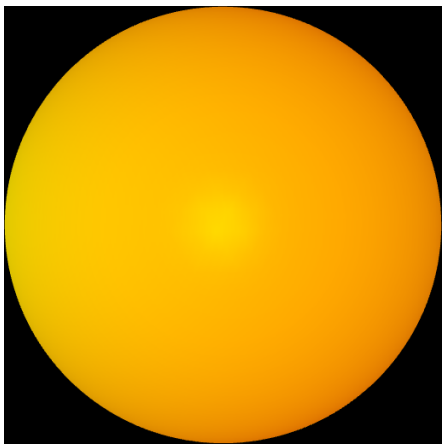


Figure 1: Image of *MatCap* created with the *MatCaps Creator*

The image is interpreted as a hemisphere, where each point on the sphere is assigned to a normal, so when a *MatCap* is applied to an object, normals are calculated for each fragment, and that information is used to map the *MatCap* image and to get the fragment colour. Figure 1 shows a *MatCap* example.

The way it works is by mapping the object's normals, which exist in relation to the camera, and the sphere's normals that can be obtained from the image. So as the camera moves around the object, the reflections and highlights move too, as if the object was moving and not the camera.

# 3 Implementation

The strategy decided to implement the whole process of creating a *MatCap* was splitting the functionalities into different webpages, to have cleaner code and to give users the possibility to do the activity they want, without the need of having to follow certain steps.

The main page is a webpage made with *HTML* and *CSS* that leads to the 3 functionalities implemented. The most relevant parts of the code for each of the features are shown and explained below.

## 3.1 "Create texture" feature

The implementation of the first feature is an improvement of the code created for the 2D drawing program. The webpage allows users to create a texture of size 1024x512, so it fits better when it is mapped to the sphere. Users can change the background colour whenever they want, without removing any element already drawn. The elements available to draw are points, triangles and circles, for all of them the colour can be changed and for triangles and circles, if different colours are selected, then the final colour of the shape is interpolated.

There is also the possibility to clean the canvas to restart the design of the texture and when the texture is finished the image can be saved.

## 3.2 "Create *MatCap*" feature

For the feature of creating a *MatCap*, the implementation of the full phong reflection model to a sphere is used. Also, a loading functionality is created to let users upload a texture, so it can be mapped to a sphere. Then users can modify the material parameters (ambient, diffuse and specular), the light emission and rotate the sphere, everything with sliders. With these settings, they can obtain the results they want and then save the final image of the *MatCap*.

The parts of how the vertex and fragment shader are implemented are the most important, as the code from the *JavaScript* has only the logic behind the updating of the values and the functionalities to load a texture and to save the new *MatCap*. Listing 1 and Listing 2 show the main functions of these 2 parts.

Listing 1: Vertex shader from "Create *MatCap*" feature

```
void main() {
  vec3 pos = (v * m * a_Position).xyz;
  vec3 light = (v * lightPosition).xyz;
  vec3 L;

  // check for directional light
  if (lightPosition.w == 0.0){
    L = normalize(light);
  }
  else{
    L = normalize(light - pos);
  }

  vec3 E = -normalize(pos);
  vec3 H = normalize(L + E);
  vec3 N = normalize((v * m * v_Normal).xyz);

  vec4 ambient = ambientProduct * ka;
  vec4 diffuse = max(dot(L, N), 0.0) * diffuseProduct * kd;
  vec4 specular = pow(max(dot(N, H), 0.0), shininess) * specularProduct *
      ks;

  if(dot(L, N) < 0.0){
    specular = vec4(0.0, 0.0, 0.0, 1.0);
  }

  gl_Position = p * v * m * a_Position;
  v_Color = (ambient + diffuse + specular) * le;
  v_Color.a = 1.0;

  fTexCoord = texCoord;
  fNormal = v_Normal;
}
```

In the vertex shader, the parameters are programmed to adjust the ambient, diffuse and specular components, depending on the value of the coefficients ($ka$, $kd$, $ks$) that are modified with the sliders. This configuration is stored in the $v\_Color$ varying variable to pass it to the fragment shader, along with the normal of the sphere ($fNormal$).

Listing 2: Fragment shader from "Create *MatCap*" feature

```
void main() {
  float u = 1.0 - (atan(fNormal.z, fNormal.x) / (2.0 * 3.14159));
  float v = 1.0 - acos(fNormal.y) / 3.14159;
  vec2 tCoord = vec2(u, v);

  gl_FragColor = texture2D(texMap, tCoord) * v_Color;
}
```

As it can be seen in the fragment shader, the texture coordinates are calculated with the normals of the sphere by using spherical coordinates. Inverse trigonometric functions are applied to obtain the UV coordinates, and then they are used to looking up the colour, that is also multiply by the $v\_Color$ to apply the precalculated illumination value.

## 3.3 "View *MatCap* on a 3D object" feature

The code to view the *MatCap* on a 3D object is based on the *MatCap* code from *mua's* repository [3]. The webpage allows users to visualize multiple 3D objects with some default *MatCaps* or load their own. There are 3 objects already on the webpage (Lamp and Mjölnir, designed by the author, and the *Blender* Monkey), but users can also upload their 3D models.

The relevant code from this feature is the fragment shader, that can be seen in Listing 3, and the load and update of the 3D object, shown in Listing 4.

Listing 3: Fragment shader from "View *MatCap* on a 3D object" feature

```
void main() {
  vec2 muv = vec2(view * vec4(normalize(vNormal), 0))*0.5+vec2(0.5,0.5);
  gl_FragColor = texture2D(matcapTexture, vec2(muv.x, 1.0-muv.y));
}
```

In the fragment shader, it is defined a colour for each vertex normal direction relative to the camera. First multiplying the normalized value of the normal by the view matrix, and then transforming from the $[-1, 1]$ interval to the texture space $[0, 1]$. Lastly, these values are used to look up the colour from the *MatCap*.

Listing 4: **updateObject()** and **render()** from "View *MatCap* on a 3D object" feature

```
function updateObject(src) {
  g_objDoc = null;
  g_drawingInfo = null;
  readOBJFile(src, gl, model, 1, true);
  interval = setInterval(function(){render(model)},1000);
}


```

```
9
10 function render(model) {
11   if (!g_drawingInfo && g_objDoc && g_objDoc.isMTLComplete()) {
12     // OBJ and all MTLs are available
13     g_drawingInfo = onReadComplete(gl, model, g_objDoc);
14
15   }
16   if (!g_drawingInfo) return;
17
18   clearInterval(interval);
19
20   ...
21 }
```

For loading a 3D object to the canvas, the *OBJParser.js* is used. As this is an asynchronous call, an interval call to the **render()** function is used for being able to display the object once it is loaded. To avoid infinite recursion, the interval is cleared when the object is loaded.

Then to load a different 3D object, it is necessary to reset the variables $g\_objDoc$ (which is an instance of the $OBJDoc$ class) and $g\_drawingInfo$ (which has the drawing information of the object and sets all the buffers), this is done in the **updateObject()** function, that is called once at the start with the default object source and then, every single time that a new object is selected or loaded.

# 4 Results

The results are shown with screenshots of the final web program, *MatCaps Creator*, where each feature will be exposed, and with some examples of textures and *MatCaps* created with the program and later displayed on 3D objects.


Figure 2: Main page of the *MatCaps Creator*

The main webpage contains a quick explanation of the 3 features implemented and give access to them by clicking on the corresponding card, as shown in Figure 2.

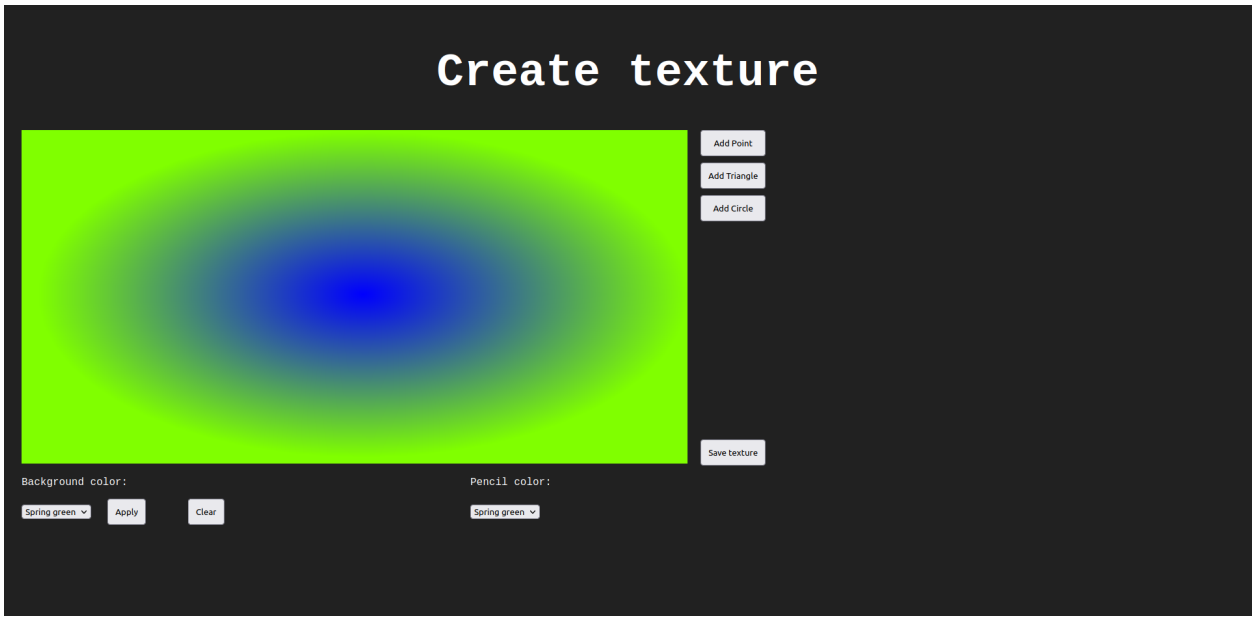



Figure 3: "Create texture" page of the *MatCaps Creator*

The first feature is to create your own texture by drawing in a canvas. It can be seen in Figure 3 the different buttons to be able to create for example a texture with a green background and a blue circle with blurred edges in the middle.

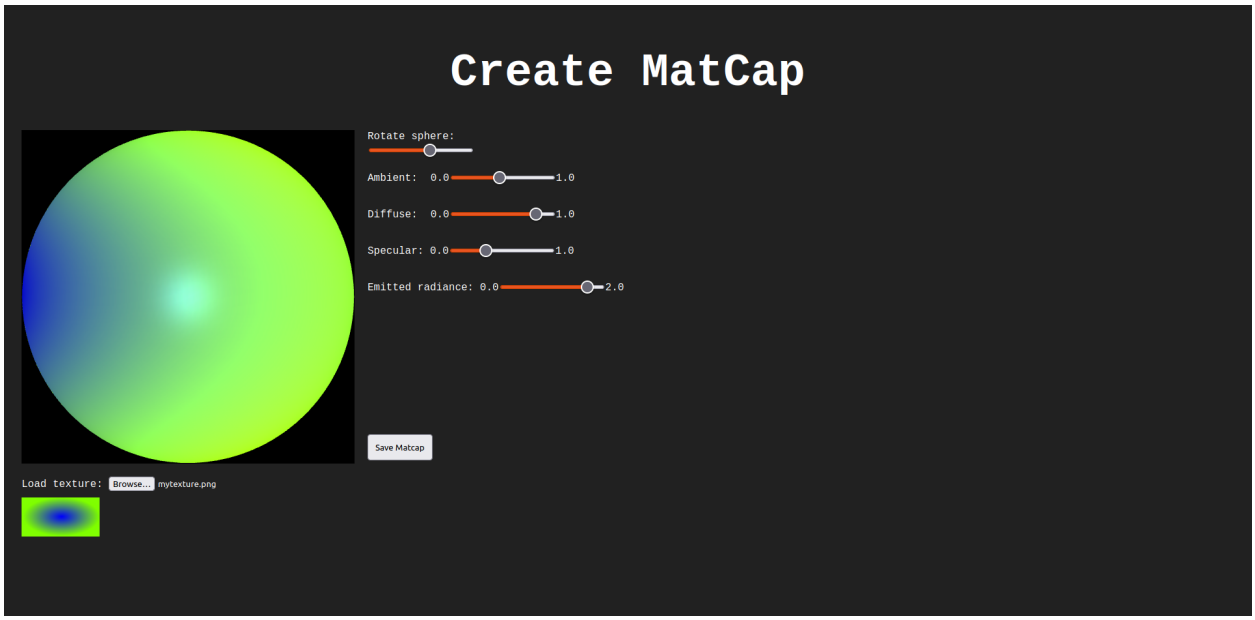



Figure 4: "Create *MatCap*" page of the *MatCaps Creator*

The second feature is to create your own *MatCap* by loading a texture and adjusting certain parameters to obtain the desired *MatCap*. The original texture is shown below the sphere and the sliders next to it, as shown in Figure 4

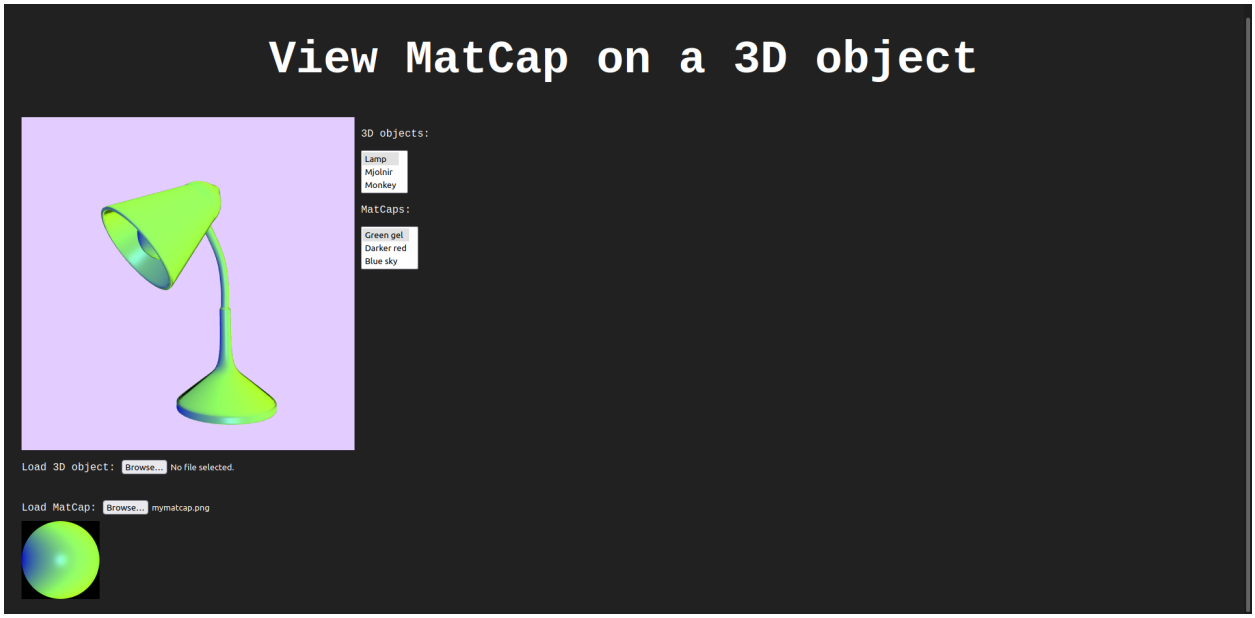


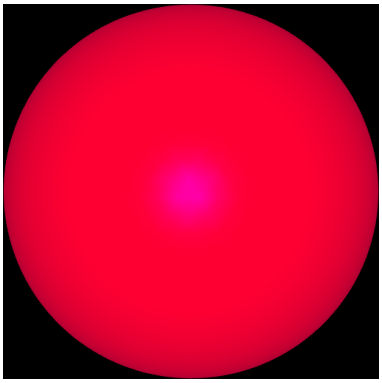

Figure 5: "View *MatCap* on a 3D object" page of the *MatCaps Creator*

The final feature is to visualize a *MatCap* on a 3D object by using the default ones or by loading your own *MatCap* or 3D object. These possibilities can be seen in Figure 5, where

there has been loaded the previous *MatCap*, shown below the object and then the actual object, in this case the Lamp model, has the *MatCap* applied.

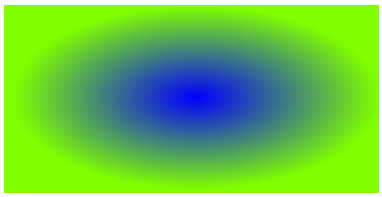
Below it is shown some different textures and *MatCaps* created with the webpage and applied to the Lamp object, visible in Figure 6.

(a) Texture 1

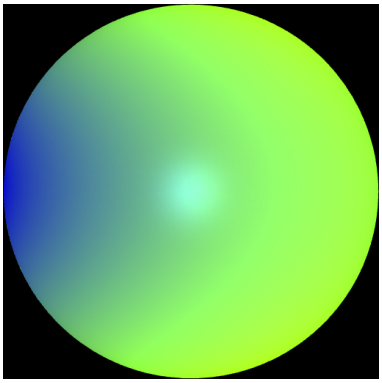


(b) *MatCap* 1

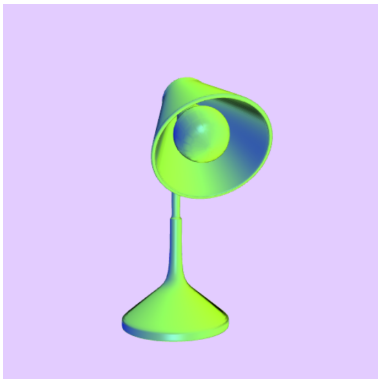
(c) 3D object with *MatCap* 1



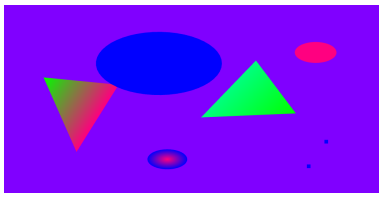
(d) Texture 2



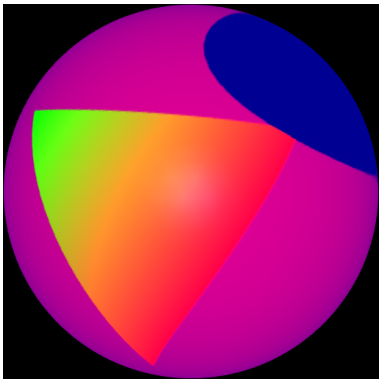
(e) *MatCap* 2



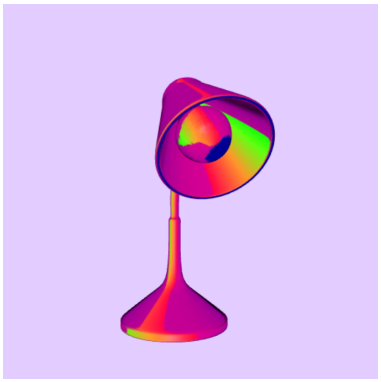
(f) 3D object with *MatCap* 2



(g) Texture 3



(h) *MatCap* 3



(i) 3D object with *MatCap* 3

Figure 6: Examples of textures, *MatCaps* and 3D objects with its *MatCap*

# 5 Discussion

In conclusion, the results achieved are really good, as the program provides users the functionality to create their *MatCaps* and also to visualize them on 3D objects. What is more, they can upload their own models and visualize other *MatCaps* on them, and not only the ones they created with the program.

There are some functionalities that could have been interesting to implement, for example giving the option to load an image as background for the creation of the texture feature, or giving the option to draw on the sphere instead of on a canvas and then mapping it to the sphere. The last one could have been an advantage, as the part of mapping the texture to the sphere could have been skipped. But it was thought that maybe it would be easier for users to draw on a flat canvas.

Finally, we can discuss the limitations of *MatCaps* summarizing them into the following points:

- *MatCaps* do not use real-time lighting sources, as they describe static appearance, the lighting and material are fixed to the image.
- *MatCaps* work good for previewing objects in 3D modelling tools. However, they are only useful for that and for sculpting. The effect disappears when the camera moves around the object, as the texture only stores one side of the sphere.
- As the image of the *MatCap* is related to the camera, if a texture is applied, it will look as an environment mapping.

# References

[1] *Webgl*, Jul. 2011. [Online]. Available: `https://www.khronos.org/webgl/`.

[2] C. Barros, *Matcap-render and art pipeline optimization for mobile devices*, Jul. 2021. [Online]. Available: `https://medium.com/playkids-tech-blog/matcap-render-art-pipeline-optimization-for-mobile-devices-4e1a520b9f1a`.

[3] M. U. Altinkaya, *Mua.github.io*, `https://github.com/mua/mua.github.io`, 2014.