# 1 GitHub repository



The GitHub repository for this project can be found at:
`https://github.com/hawkaa/bugfree-octo-meme`
The image processing code can be found in */matlab* and the OpenGL c++ code can be found in */cpp/src*

# 2 Image processing part

The image processing part can be split into following parts parts:

1. Sampling colors from the
2. Performing morphological operations
3. Thresholding the image to a black and white bitmap for each predefined color.
4. Generating geometries by closing the thresholded image and grouping.
5. Using the color information and geometry information to generate a text file to the OpenGL application.

## 2.1 Sampling

A color is sampled by using the eye dropper tool in an image editor. When enough samples are gathered, the mean and standard deviation vectors are calculated, and sent to the main function for generating the text file.

## 2.2 Thresholding

```
function [ BW ] = RGBDistanceThreshold(I, Avg, Std, t)
    RGBD = RGBDistance(I, Avg, Std);
    [r,c] = size(RGBD);
    BW = zeros(r, c);
    for y = 1:r
        for x = 1:c
            if RGBD(y,x) < t
                BW(y,x) = 255;
            end
        end
    end
end
```

This function is run for each color. Each color has a mean and standard deviation vector for each R, G and B values. If the RGBDistance for each pixel is low enough, the image is colored white. The RGBDistance for a pixel is the sum of the deviation for each color value.

## 2.3 Performing morphological operations

```
BW_Filled = imfill(BW);
BW_1 = imerode(BW_Filled, SE);
BW_2 = imdilate(BW_1, SE);
BW_3 = imdilate(BW_2, SE);
BW_4 = imerode(BW_3, SE);
```

The thresholded image is first filled (lightning effects on the non-stops are creating holes), then closed and opened.

## 2.4 Generating geometries

```
function [ out ] = GetGeom(BW, reverse)
    [rS, cS] = size(BW);
    [Labeled, n] = bwlabel(BW, 8);
    out = zeros(0,3);
    for i=1:n
        [r, c] = find(Labeled==i);
        radius = sqrt(size(r,1)/pi);
        if reverse
            out(end+1,1) = cS - mean(c);
            out(end,2) = rS - mean(r);
            out(end, 3) = radius;
        else
            out(end+1,1) = mean(c);
            out(end,2) = mean(r);
            out(end, 3) = radius;
        end
    end
    out = round(out);
end
```

This function accepts a BW image and a boolean value if the geometries should reverse the y-axis or not.

## 2.5 Writing the file

```
function [] = PopTheNonStops(I, filename, colors, SE, shape, t)
    file = fopen(filename, 'w+');
    length = size(colors);
    length = length(1);

    % Number of colors
    fprintf(file, '%i\n', length);

    % Print color definitions
    for i=1:length
        fprintf(file, '%i %i %i %i\n', i, round(colors{i,1}));
    end

    % Print the vertices
    for i=1:length
       BW = RGBDistanceThreshold(I, colors{i,1}, colors{i,2}, t);
        BW_Filled = imfill(BW);
        BW_1 = imerode(BW_Filled, SE);
        BW_2 = imdilate(BW_1, SE);
        BW_3 = imdilate(BW_2, SE);
        BW_4 = imerode(BW_3, SE);
        Vertices = GetGeom(BW_4, true);
         for j=1:size(Vertices)
            fprintf(file, '%i %i %i %s %i\n', Vertices(j, 1), Vertices(j, 2),
            Vertices(j, 3), shape, i);
```

```
        end
    end
end
```

This function saves the color data and geometries to the given filename, by iterating over the color definitions.

# 3 Exchanging data between the applications

The file consists of three parts

1. Header
2. Color definitions
3. Vertex data

The header contains the number of color definitions, so it will be easier to parse the data
The color definitions have the following syntax:

```
[id] [R] [G] [B]
```

id is a unique number that defines the color (to be used in the vertex data. A color definition for yellow would look like this:

```
1 255 255 0
```

The last part of the document is the vertices. They have the following format:

```
[x] [y] [size] [shape] [color_id]
```

- x: X coordinate (X-axis pointing right)
- y: Y coordinate (Y-axis pointing up)
- size: Radius of the shape, measured in pixels
- shape: String containing the shape. 'c' (circle) is currently the only supported shape.
- color_id: Integer telling what color definition to be used when rendering the shape

A typical entry for a 16 pixel radius point would look like this:

```
278 157 16 c 3
```

# 4 OpenGL

The first thing the code does, is to load shaders, bitmaps and objects from the file generated by MatLab. All of the geometries are loaded into the following class definition:

```
class ImageObject
{
public:
        ImageObject(ObjectType type, float x, float y,
        float radius, glm::vec4 color, int colorIndex);
        ~ImageObject(void);

        static ObjectType getObjectTypeFromString(char* s);
```

```
        float getX();
        float getY();
        float getRadius();
        int getColorIndex();
        ObjectType getType();
        glm::vec4 getColor();

private:
        float x,y,radius, colorIndex;
        ObjectType type;
        glm::vec4 color;

};
```

These objects are then sent to the renderer class, which generates ellipsoids into the VBO. Squares with textures (object count) are also loaded.

Lightning is set in the texture fragment shader:

```
vec3 lightPos = vec3(300, 1000, 0);
vec3 lightColor = vec3(1, 1, 1);
float lightPower = 2.0f;
// Diffuse color
vec3 normalDir_worldspace = normalize(vertexNormal_worldspace);
vec3 lightDir = normalize(position_worldspace - lightPos);
float diffuse = max(0.3, dot(normalDir_worldspace, lightDir));
vec3 normalDir = normalize(normal_cameraspace);
vec3 eyeDir = normalize(eyeDirection_cameraspace);
vec3 reflectDir = reflect(-lightDir, normalDir);

float specAttenuation = clamp(dot(eyeDir, reflectDir), 0, 1);
color = clamp(vec4(texture(myTexture, UV).rgb, 1), 0, 1);
```

## 4.1 Animating

The program has the following main loop:

```
do {
        time2 = glfwGetTime();
        delta = float(time2-time1);

        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_HIDDEN);

        input->update();
        text->update(this->renderer, delta);
        camera->move(delta);
        glfwSetCursorPos(window, Octobrain::screenWidth/2, Octobrain::screenHeight/2);
        renderer->invalidate();

        glfwSwapBuffers(this->window);
        glfwPollEvents();
```

```
        time1 = time2;
```

```
}while(glfwGetKey(this->window, GLFW_KEY_ESCAPE) != GLFW_PRESS && !glfwWindowShouldClose(t
```

On each update, this method will rotate the text objects (object count) and move the camera. renderer-¿invalidate() will load all these changes into the VBO and render the scene.

# 5 Aspects of improvement

The OpenGL part of this project is where we have put the most effort, so this solution is rather robust and versatile. Most of the improvements is therefore related to the image processing part.

The first challenge, is that every photo needs to be sampled before being processed. A solution to this, could probably be to do some sort of preprocessing to all the images so the colors of the smarties would be normalized and recognized with fewer samples.

Secondly, we decided not to keep the point radius fixed, instead creating an estimate based on the pixel area. This is indeed the best solution if the morphed image actually kept all of the highlighted smartie white. This is not the case, our current function does not always detect all of the smartie. More research on image opening and closing could have been done here.