COMPUTER GRAPHICS & 3D

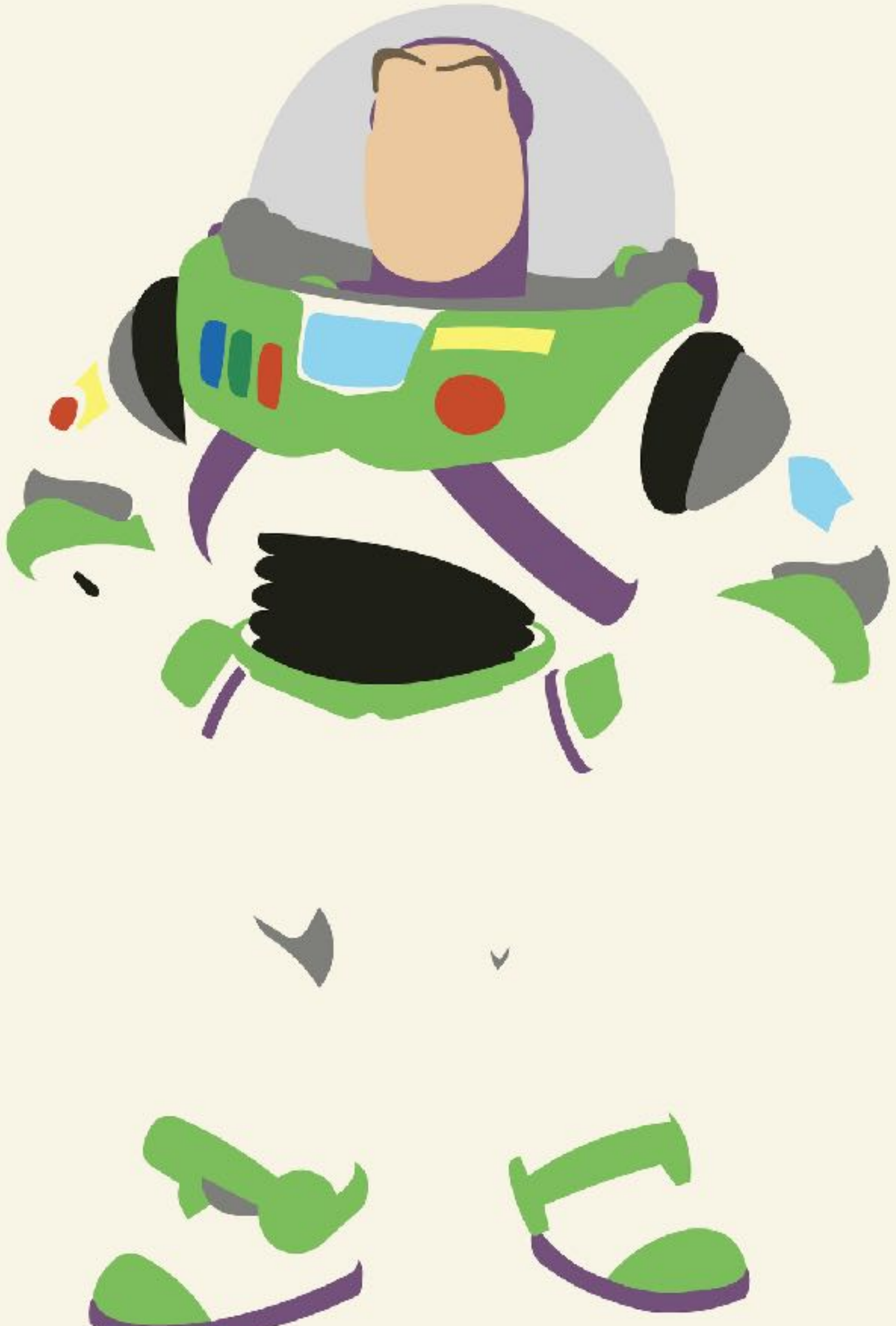
TOY STORY – DEPTH OF FIELD

INTENTS

- Creating a scene using WebGL
- Simulating Depth of Field effect on the custom scene
- Using shaders to create the DoF effect

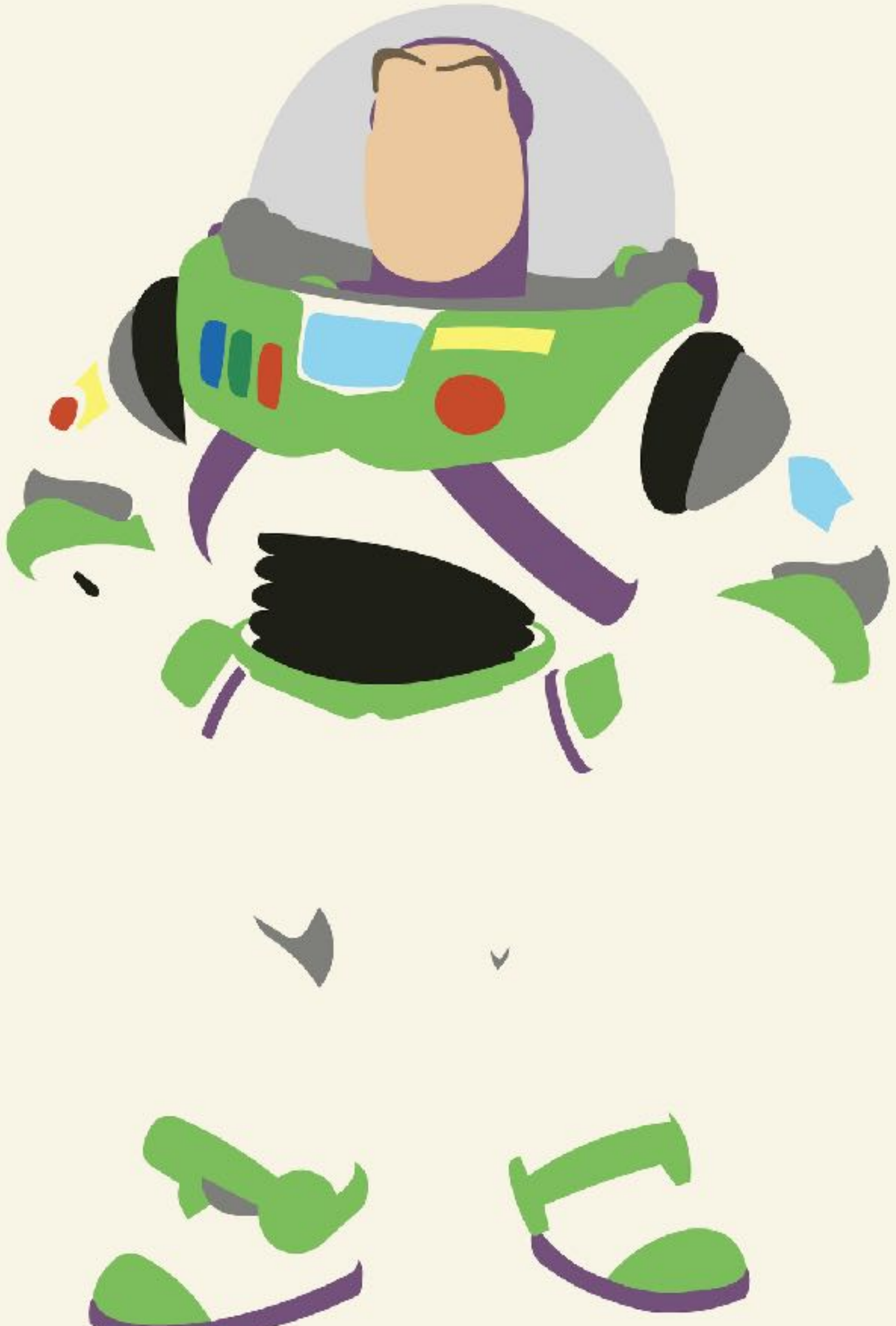
THE DEPTH OF FIELD EFFECT

- Depth of Field is the effect in which objects within some range of distances of a scene appear in focus.
- The objects nearer and farther than this range will appear out of focus.
- A lens usually let the light pass through a film or the retina: when the light converges to a single point in the film, the light's source will appear in focus.
- Everything else will be projected to an area called ***Circle of Confusion***.
- This area defines what part of the object will appear in focus or not.



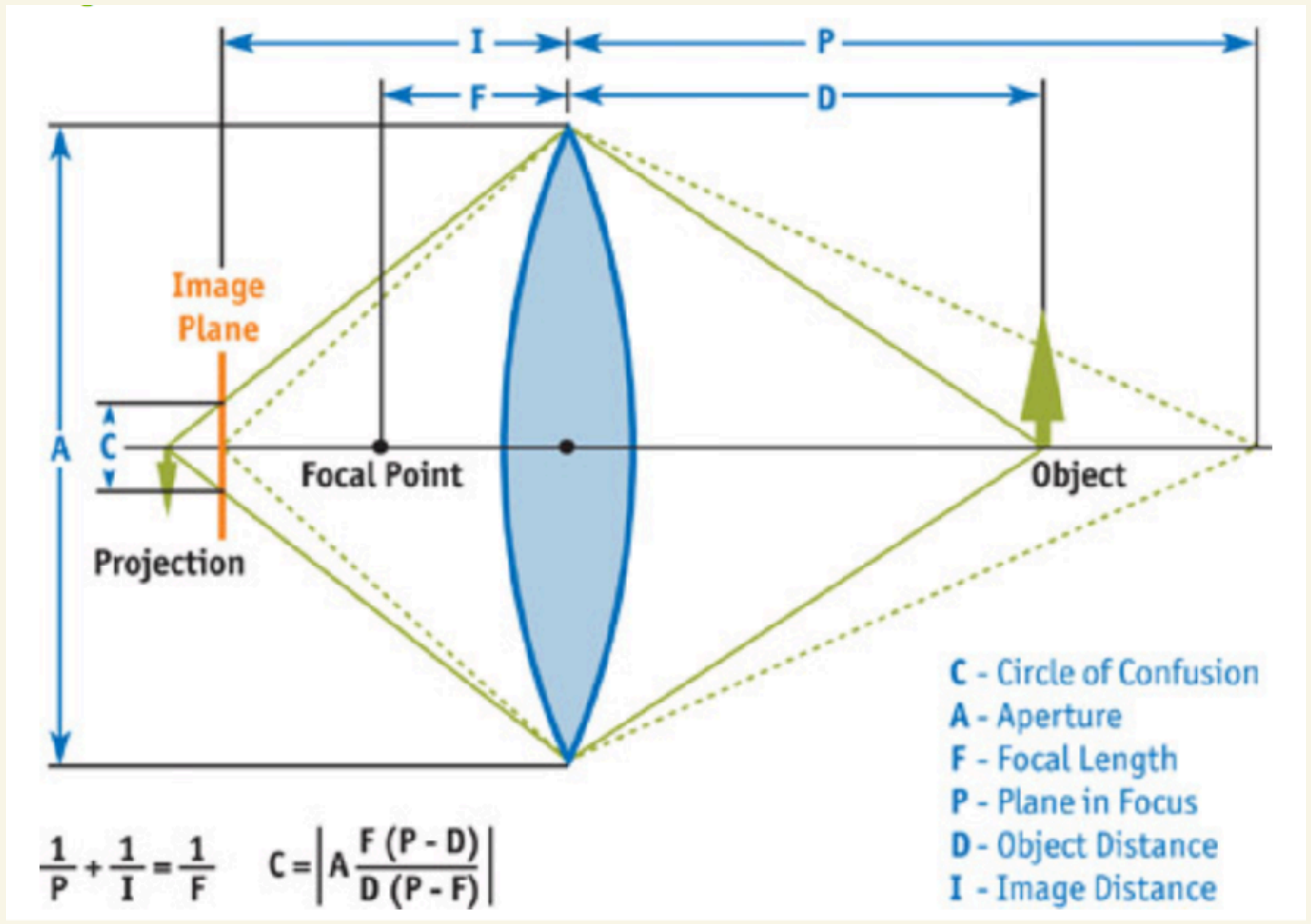
DEPTH OF FIELD – Involved Parameters

- ***CoC:*** Circle of Confusion, used to define parts in and not in focus.
- ***Aperture:*** dimension of the lens diaphragm in which the light passes through. Small values bring to low DoF effect, big values bring to high DoF effect.
- ***Focal length:*** measure of how strongly the system converges or diverges light.
- ***Plane in Focus:*** the plane that must be in focus.
- ***Object Distance:*** object distance from the lens.
- ***Image Distance:*** distance from the lens of the plane where light is projected.

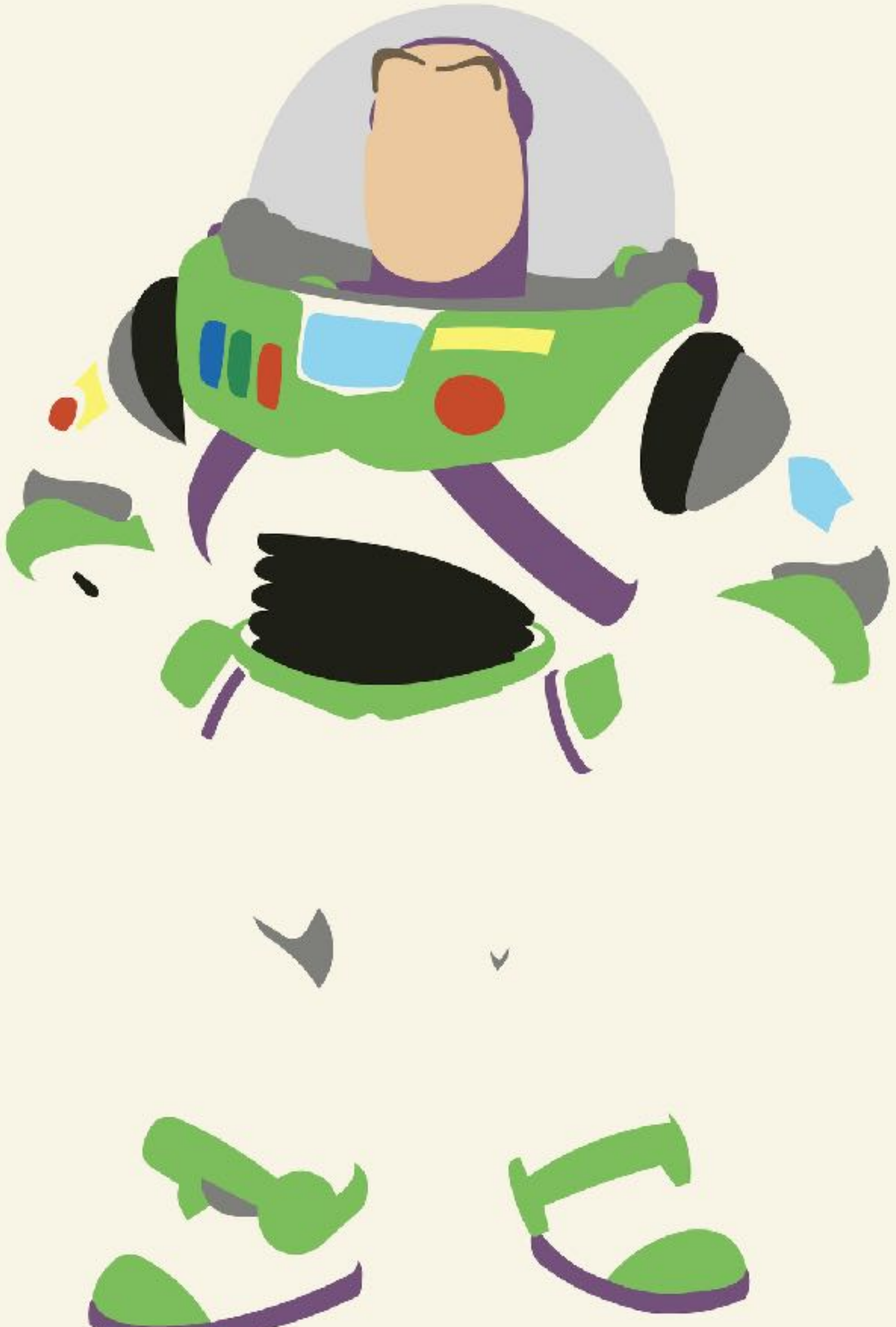


DEPTH OF FIELD – Model

- As shown in the figure, the part of the projected object out of its Circle of Confusion will appear out of focus.

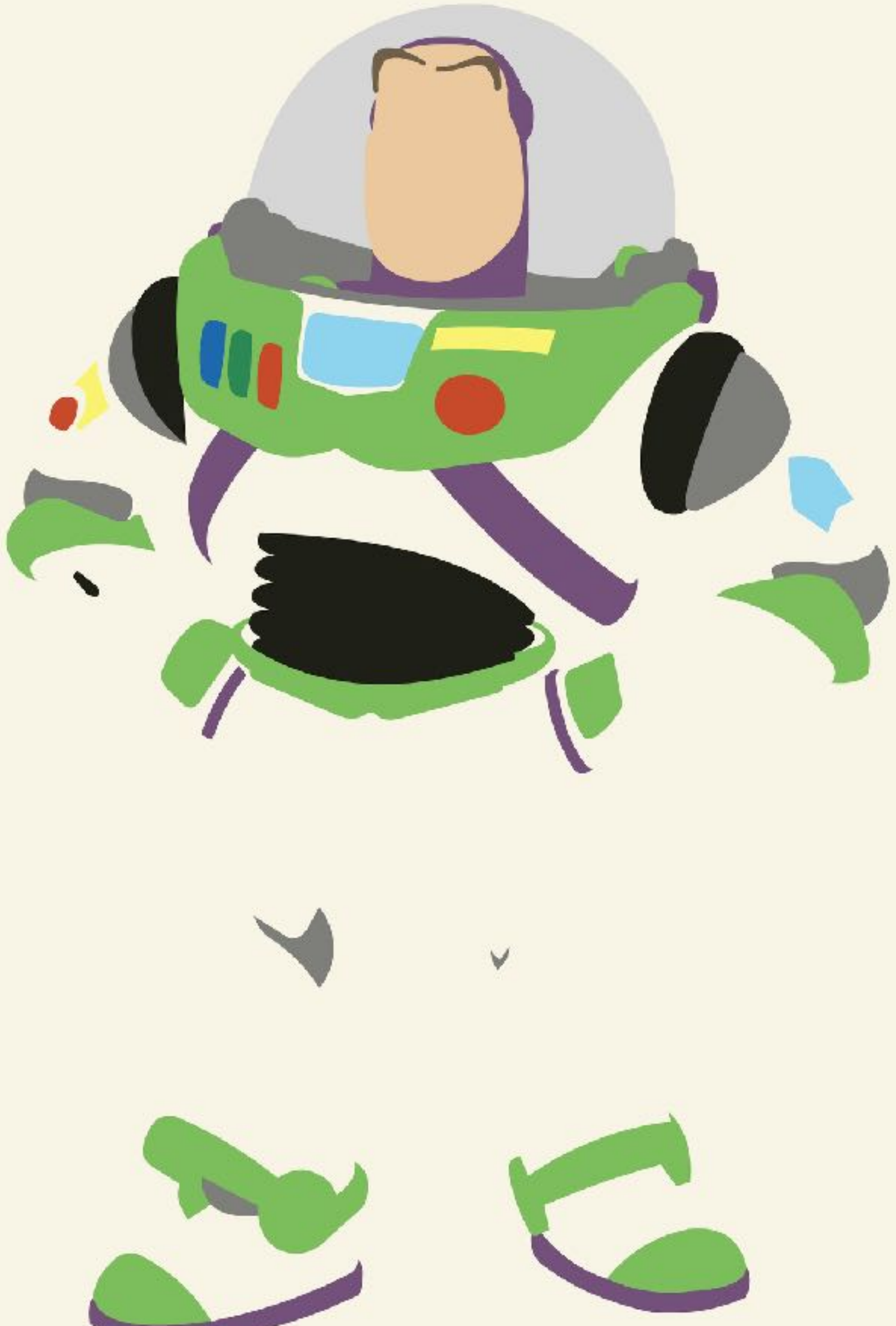


$$\frac{1}{P}+\frac{1}{I}=\frac{1}{F} \qquad C=\left|A\frac{F(P-D)}{D(P-F)}\right|$$



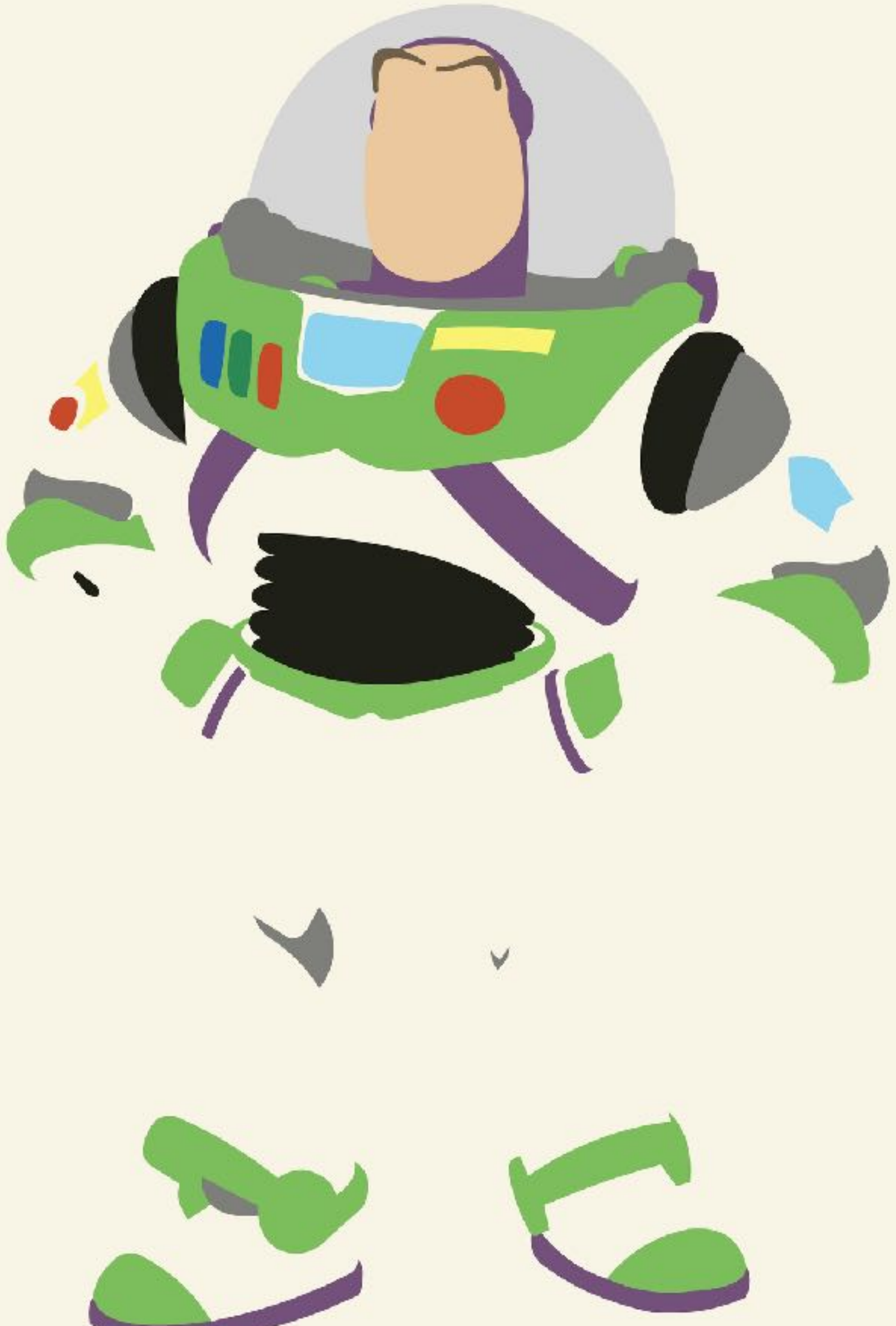
THE PROJECT

- The project has been implemented in **WebGL**.
- The scene has been created by using **three.js**.
    - three.js is a cross-browser Open Source Javascript library that uses WebGL to create and display animated 3D computer graphics scene. It simplifies the creation of complex 3D scenes (which could be difficult using only Javascript)
    - three.js creates a scene adding objects defined by a **geometry** (which corresponds to a vertex shader) and a **material** (which corresponds to a fragment shader).
    - It also provides method to implement user controls, post-processing effects, etc..
- The Depth of Field effect has been implemented by a **fragment shader**.



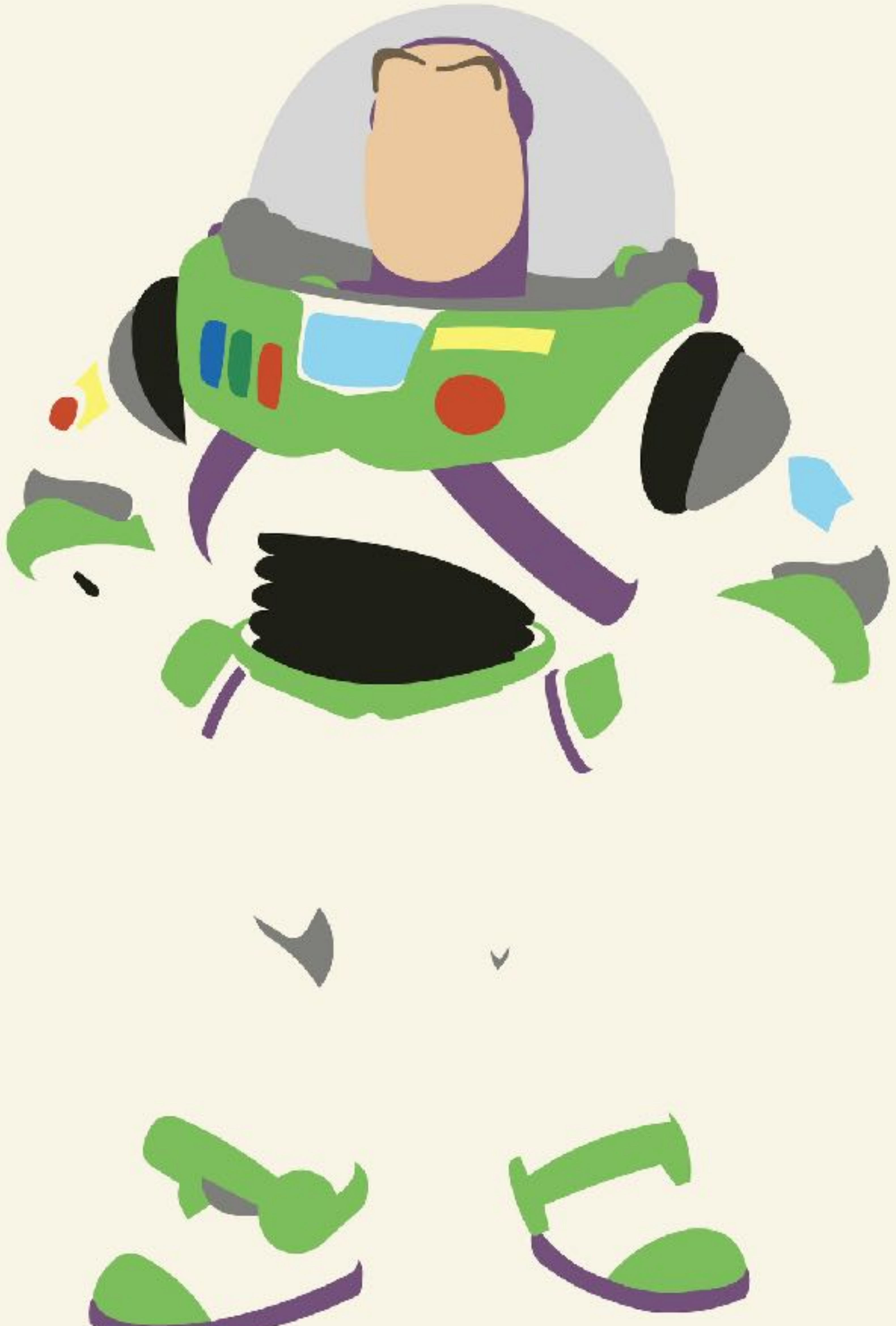
PROJECT – How it works, part 1

- Initialize the Scene:
  - Camera
  - Controls (***Pointer Lock Controls***, provided by three.js)
- Load the objects on the Scene (***CreateWorld()*** function):
  - Light and Chandelier
  - Create the room (floor, walls and ceiling) as set of *Plane Geometries* with textures
  - Add dices as *Box Geometry* with textures
  - Add ball as a *Sphere Geometry* with texture
  - Add Woody, Buzz and a bed (imported objects, loaded using three.js' **MTLoader & OBJLoader**)



PROJECT – How it works, part 2

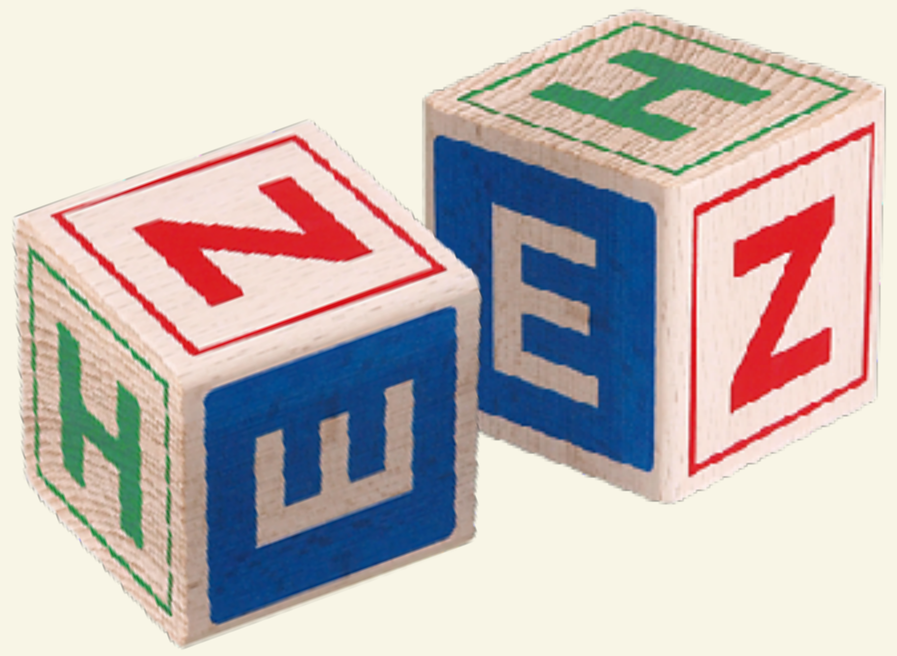
- Load auxiliary walls next to the bed, to detect collisions:
  - By using **CollisionDetection()** function, the user won't go through the walls (in this case auxiliary walls were not required) and through the bed
- Call the routine of post-processing effects so that to render the **Depth of Field effect**.
- Add an audio file to the scene.
- Create a **GUI** to let the user to vary *Aperture* and *Plane in Focus*.



THE SCENE – Three.js Objects

The Room

- Floor
- 4 Walls
- Ceiling
  - Plane Geometry
  - MeshPhongMaterial
- Every part of the room uses a texture instead of a simple color

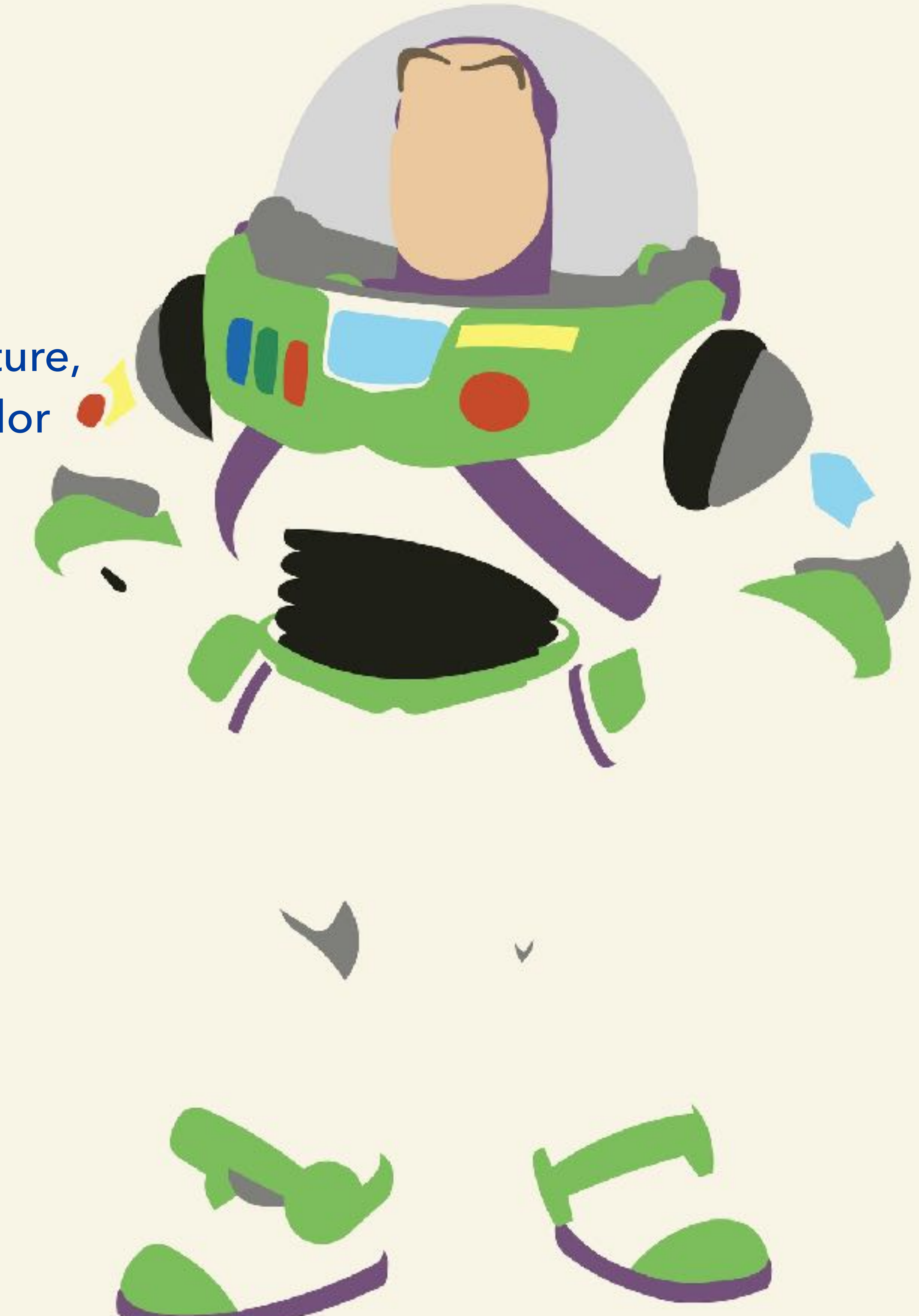


The Dices

- Box Geometry
- MeshPhongMaterial
- Every face of the dice uses a texture with a different letter, instead of a simple color

The Ball

- Sphere Geometry
- MeshPhongMaterial
- The ball uses a texture, instead of a simple color



THE SCENE – Imported Objects

The Bed

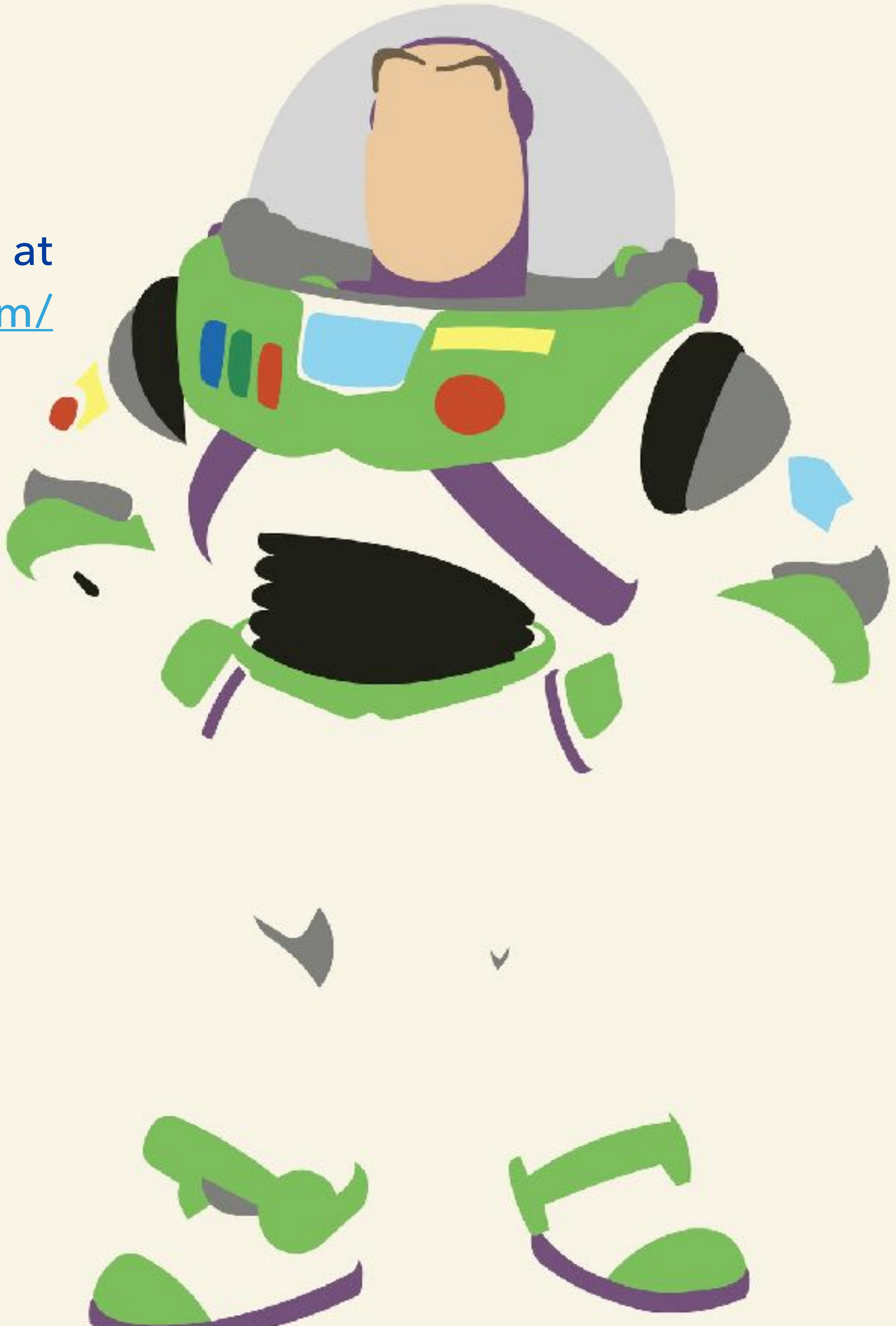
Model can be found at https://www.blendswap.com/blends/view/77055

Buzz Lightyear

Model can be found at https://www.yobi3d.com/q/buzz-lightyear

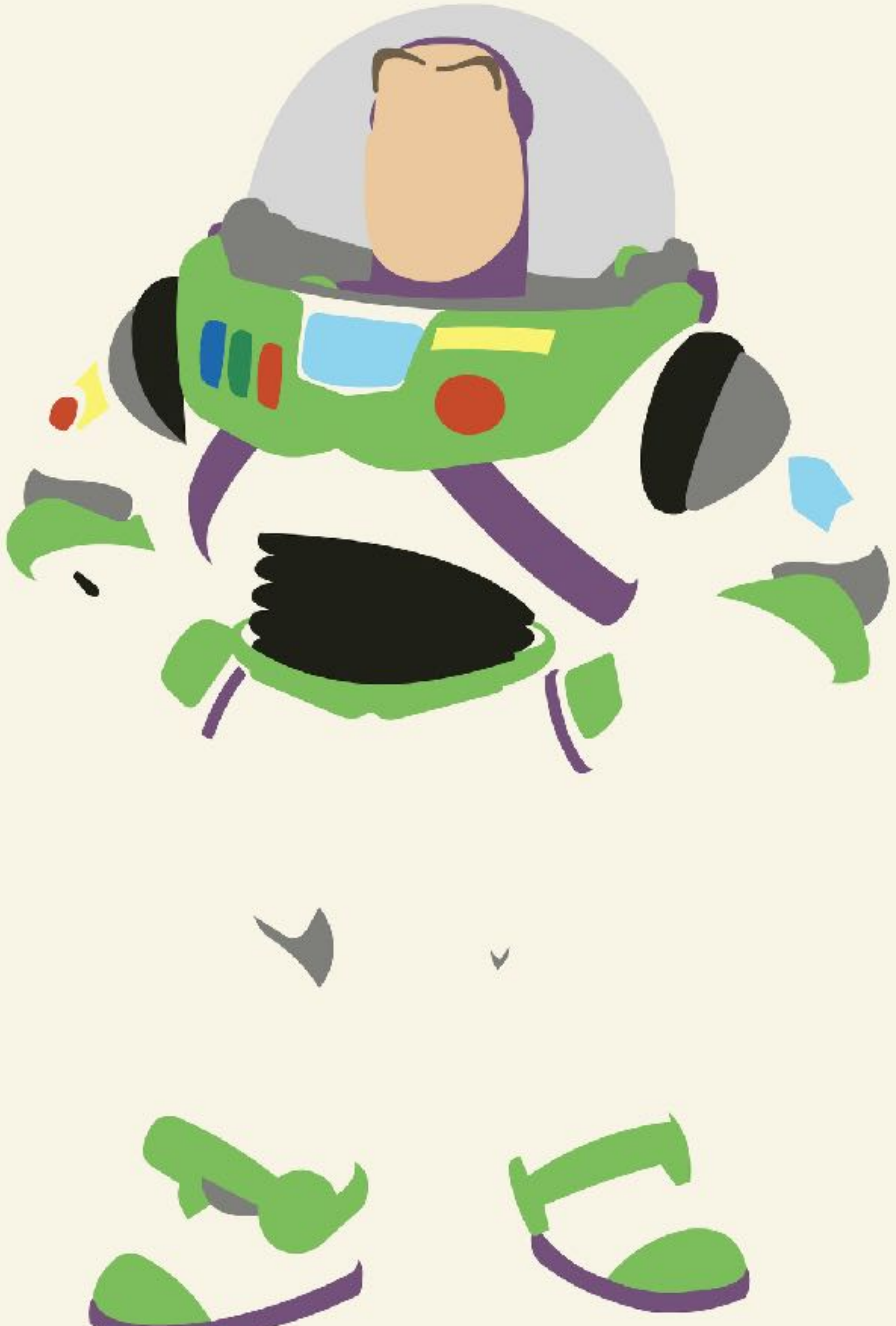
Woody

Model can be found at https://www.yobi3d.com/q/woody



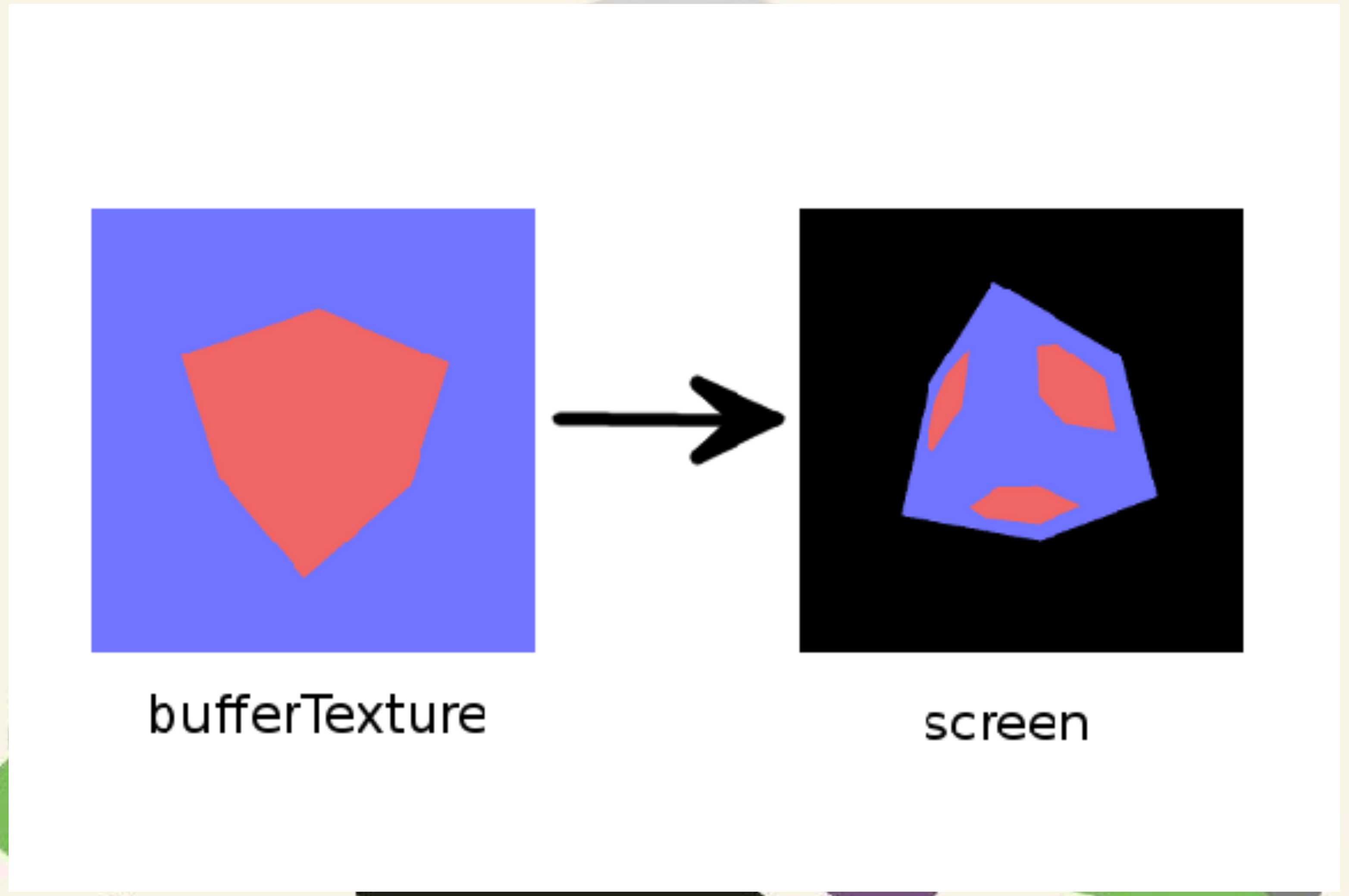
SIMULATING DEPTH OF FIELD

- There are lot of techniques to simulate Depth of Field.
- **PROBLEM.** The biggest problem of DoF is that the model of it assumes that the lens behaves like a retina or like any kind of camera, while the typical Computer Graphics model uses a a pinhole camera (everything is in focus)
- Because of the DoF model and the difficulties that can be found to get all the parameters needed, we will follow a different approach.
- **Step 1.** Implement a routine that let us apply an external GLSL shader to a three.js scene.
- **Step 2.** Implement a fragment shader that simulates DoF without using DoF Model.



POST-PROCESSING WITH RENDER TO TEXTURE

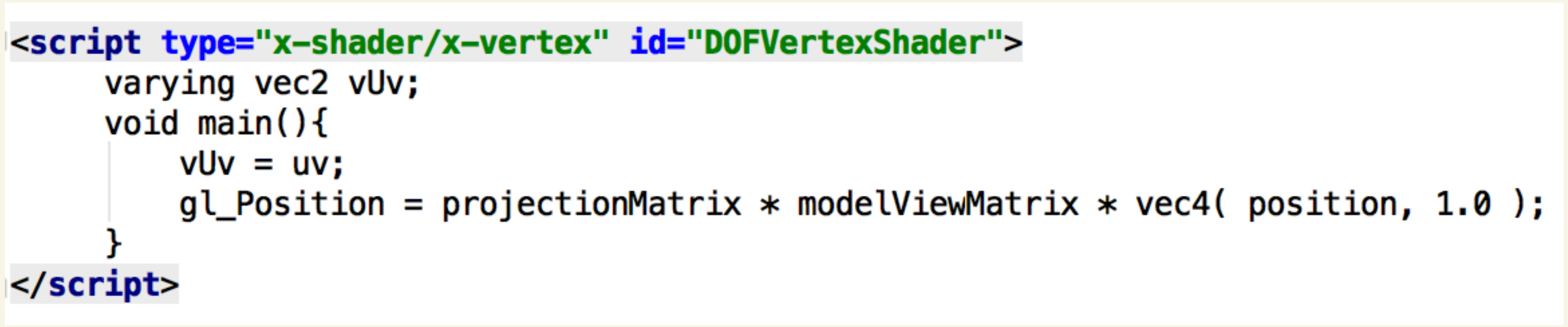
- To apply a shader over the whole scene, we have to render the scene to a texture.
- **Rendering to Texture** is a technique that let to render a scene to a part of another scene.
- By this, it is possible to create lot of effects, such as **post-processing effects**.
- When we use post-processing effects, we render the whole scene to a texture that won't be initially displayed. This texture will be used to do all the calculations needed to get the desired effect. Once this is done we then render a scene that is composed by a single plane: the previous texture will be displayed on this plane.



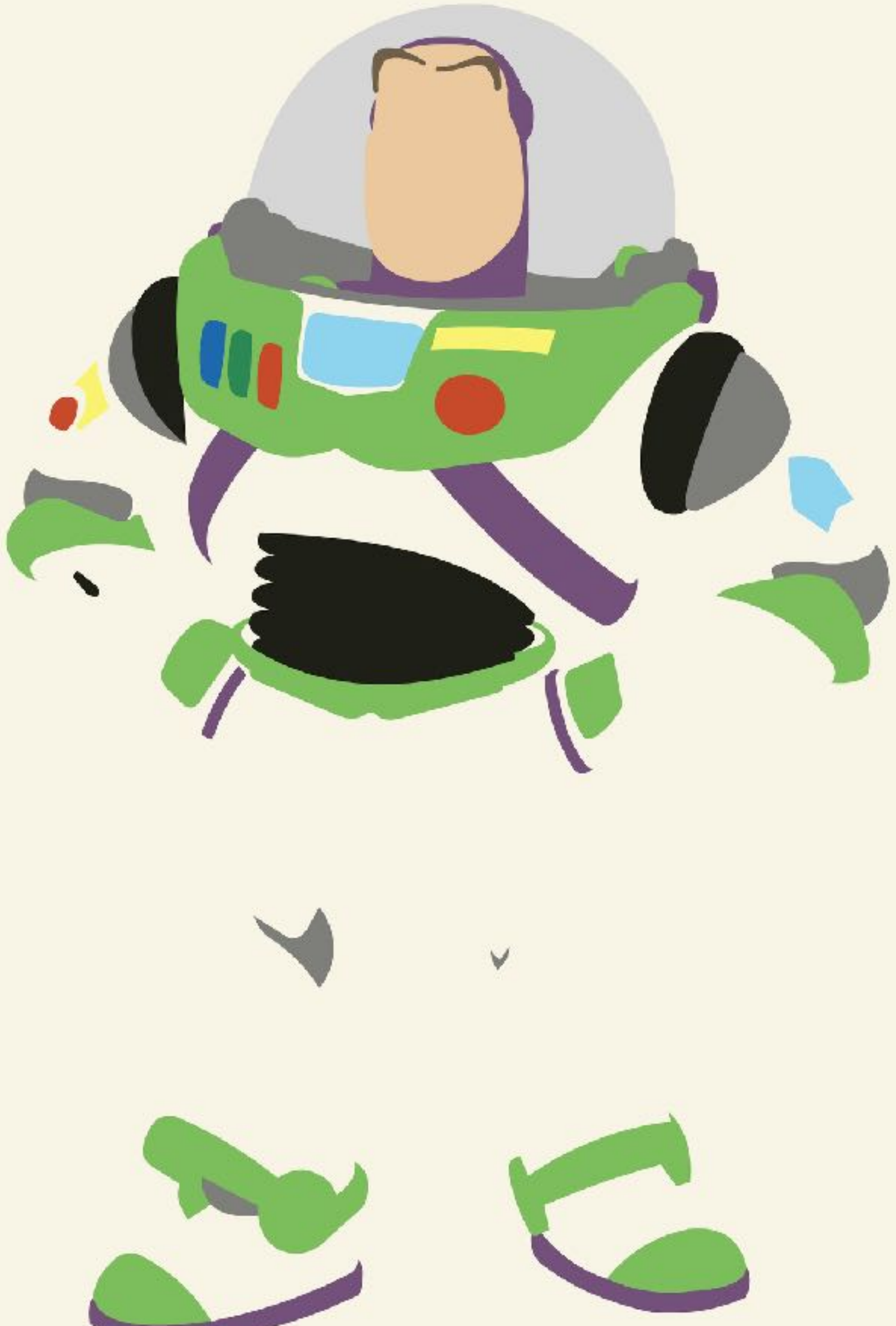
- When we render to texture in order to simulate Depth of Field, we need to get values about colors and depth of the scene.
- This can be done using Three.js **WebGLRenderTarget** and **DepthTexture.**
- These information will be passed to our shaders, using Three.js **ShaderMaterials** as uniforms variable. We also pass parameters like Aperture and Plane in Focus, whose values can be change by the user at runtime.
- Once this is done, we can finally use our shaders to simulate the Depth of Field.

DEPTH OF FIELD – Vertex Shader

- To get the Depth of Field effect we only have to change our pixels' color values: the Vertex Shader doesn't have to do anything.

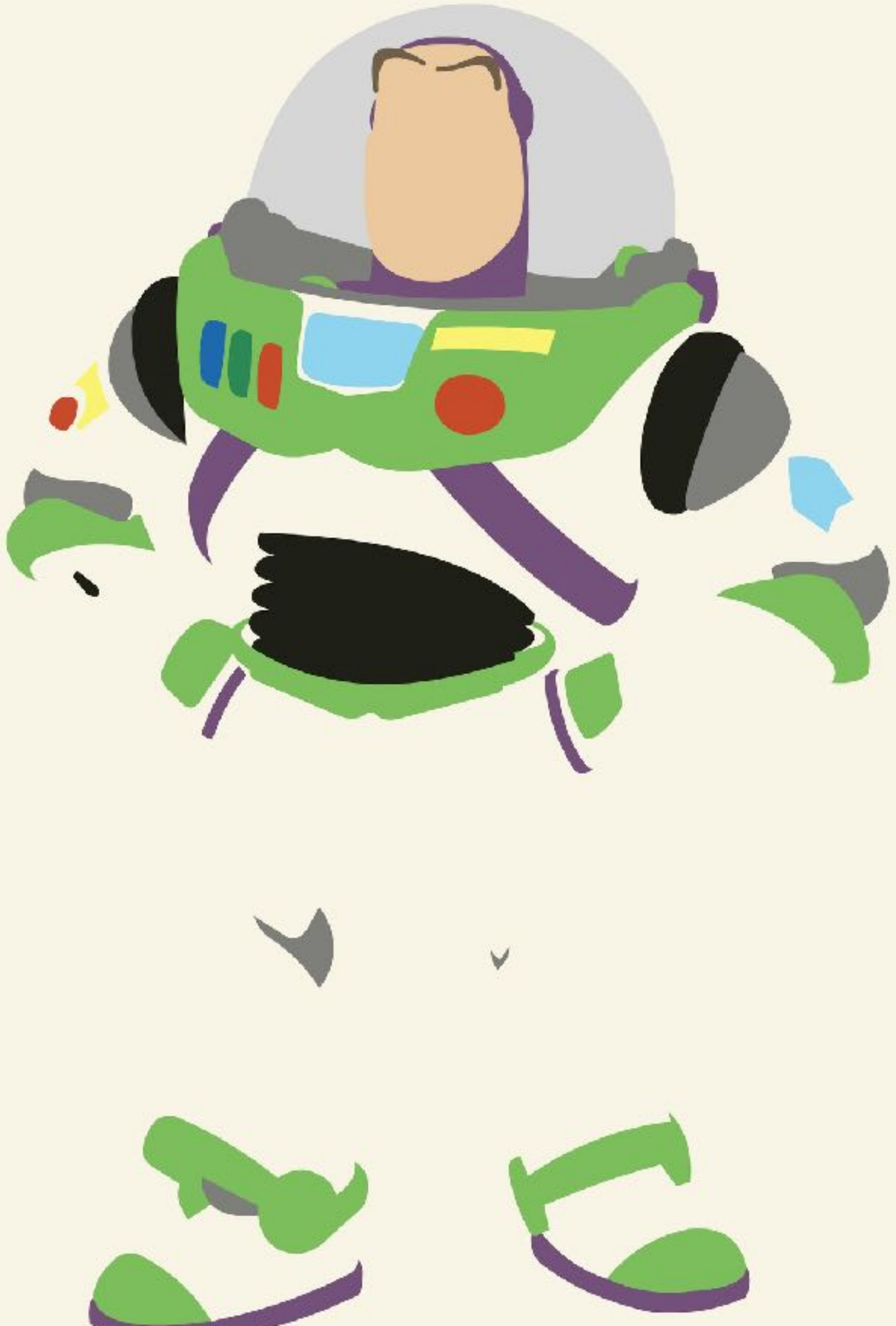


```
<script type="x-shader/x-vertex" id="DOFVertexShader">
    varying vec2 vUv;
    void main(){
        vUv = uv;
        gl_Position = projectionMatrix * modelViewMatrix * vec4( position, 1.0 );
    }
</script>
```



DEPTH OF FIELD – Fragment Shader

- To simulate Depth of Field we need to blur all the pixels that are out of the plane in focus chosen by the user with the GUI.
- One of the most common way to blur an image is applying on it a **Gaussian Blur**.
- For each pixel, we take the surrounding neighbors and we make an average of their values weighted by a bivariate Gaussian distribution. Once we get the new value, we change the original one with it.
- The number of neighbors depends on the size of the kernel used.

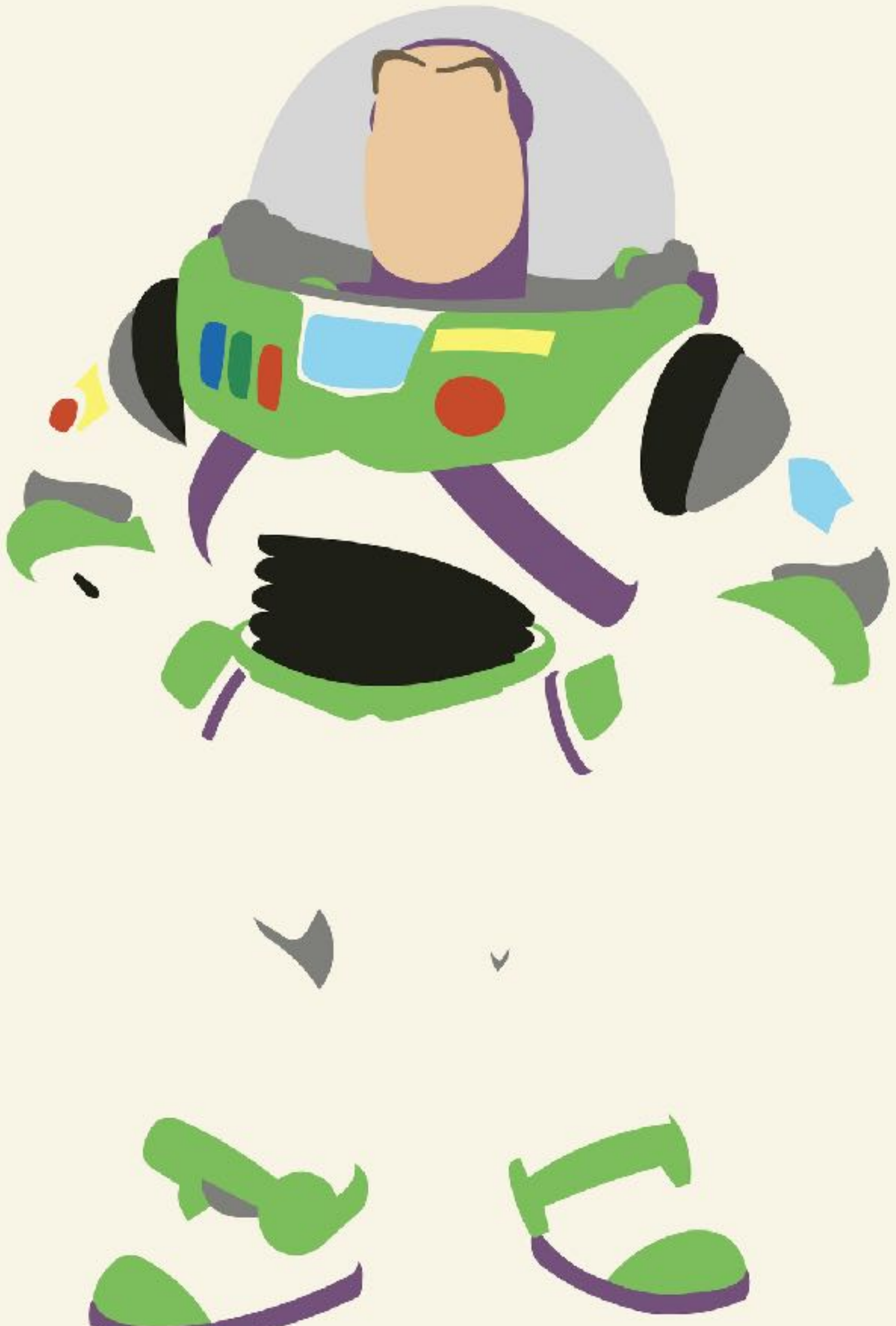


DEPTH OF FIELD - Fragment Shader

- As already said, we compute the Gaussian weights using a Gaussian distribution:
$$w(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

- The x and y values represent the location of a neighbor at a certain **offset** distance from the pixel we want to process. This offset is passed to the method that compute the weights.

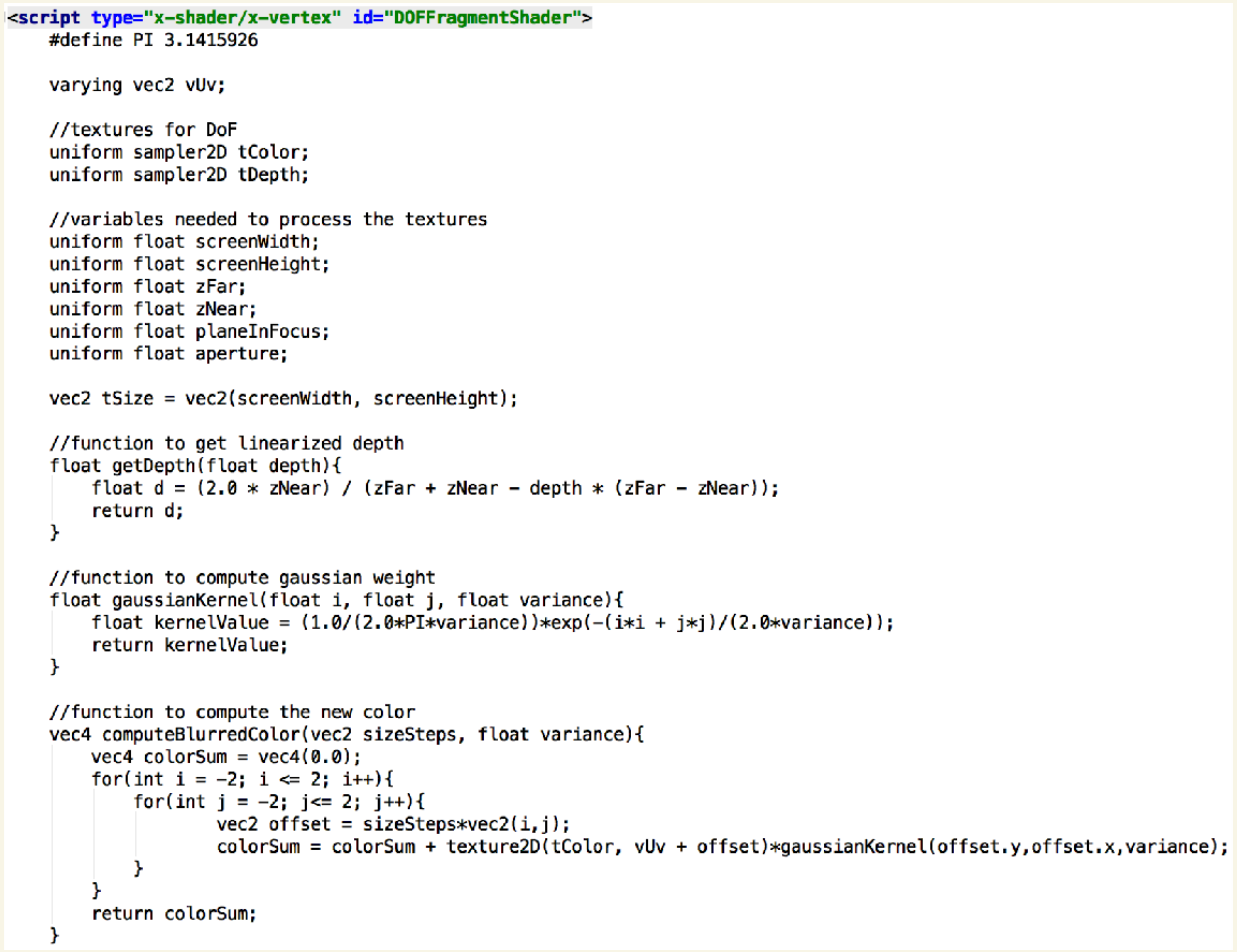
- For different values of the offset, it is possible to get a greater (or smaller) amount of blur: the more (less) the neighbor is far (near) from the pixel we're processing, the more (less) it should appear blurred.



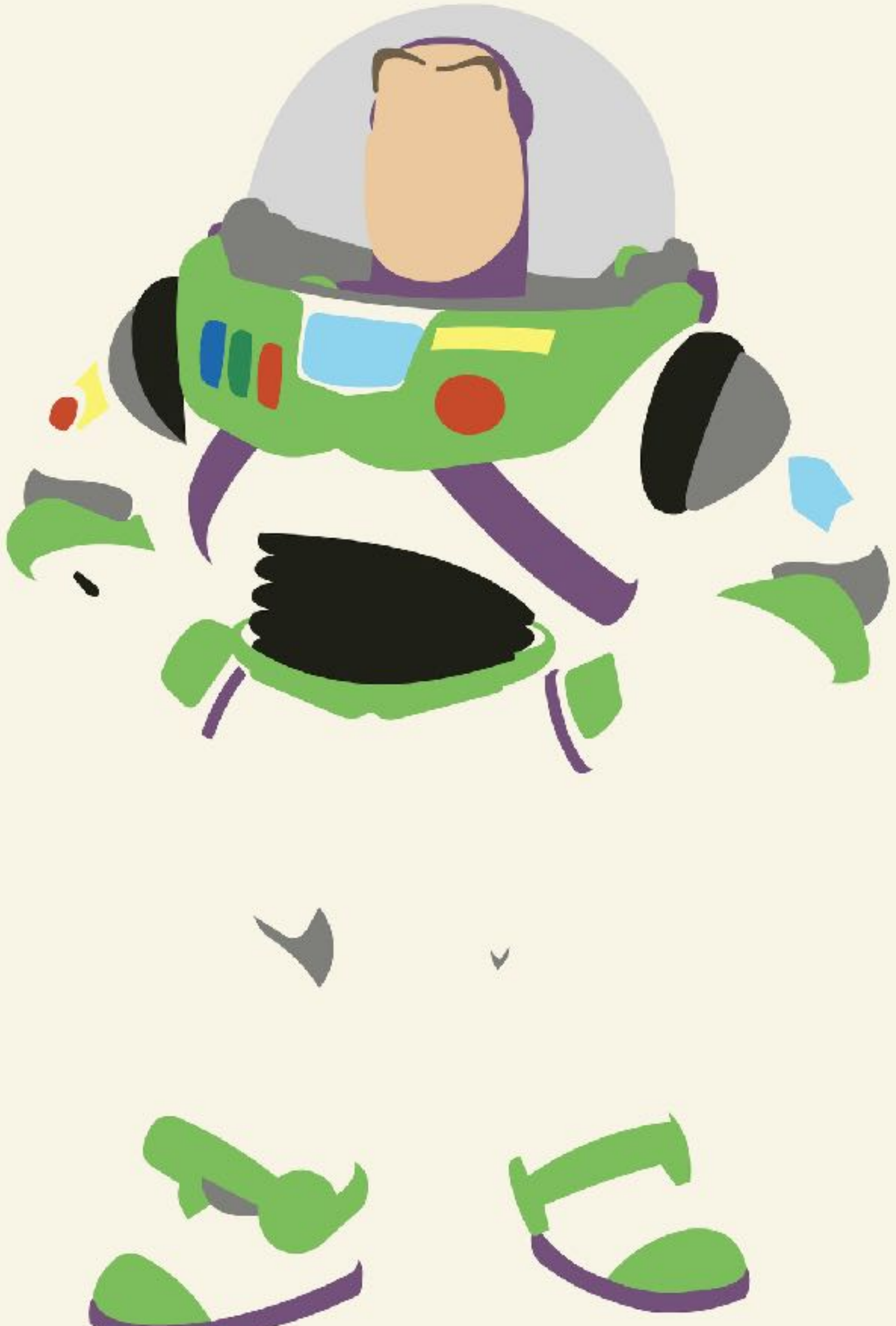
DEPTH OF FIELD - Fragment Shader

- In fact, the offset value is the product of *aperture* and *distanceFromPlaneInFocus* variables. If one of these is set to zero, all the scene will appear in focus.
- By this way, all the pixels that are on the plane in focus will appear in focus, while all the pixels that are far from this plane will appear blurred, with an amount of blur that is directly proportional to the distance from the plane.
- The distance from the plane in focus is computed by subtracting the depth value of each pixel from the *planeInFocus* variable.
- At the same time, changing *Aperture* value, the user can increase or decrease the DoF effect.

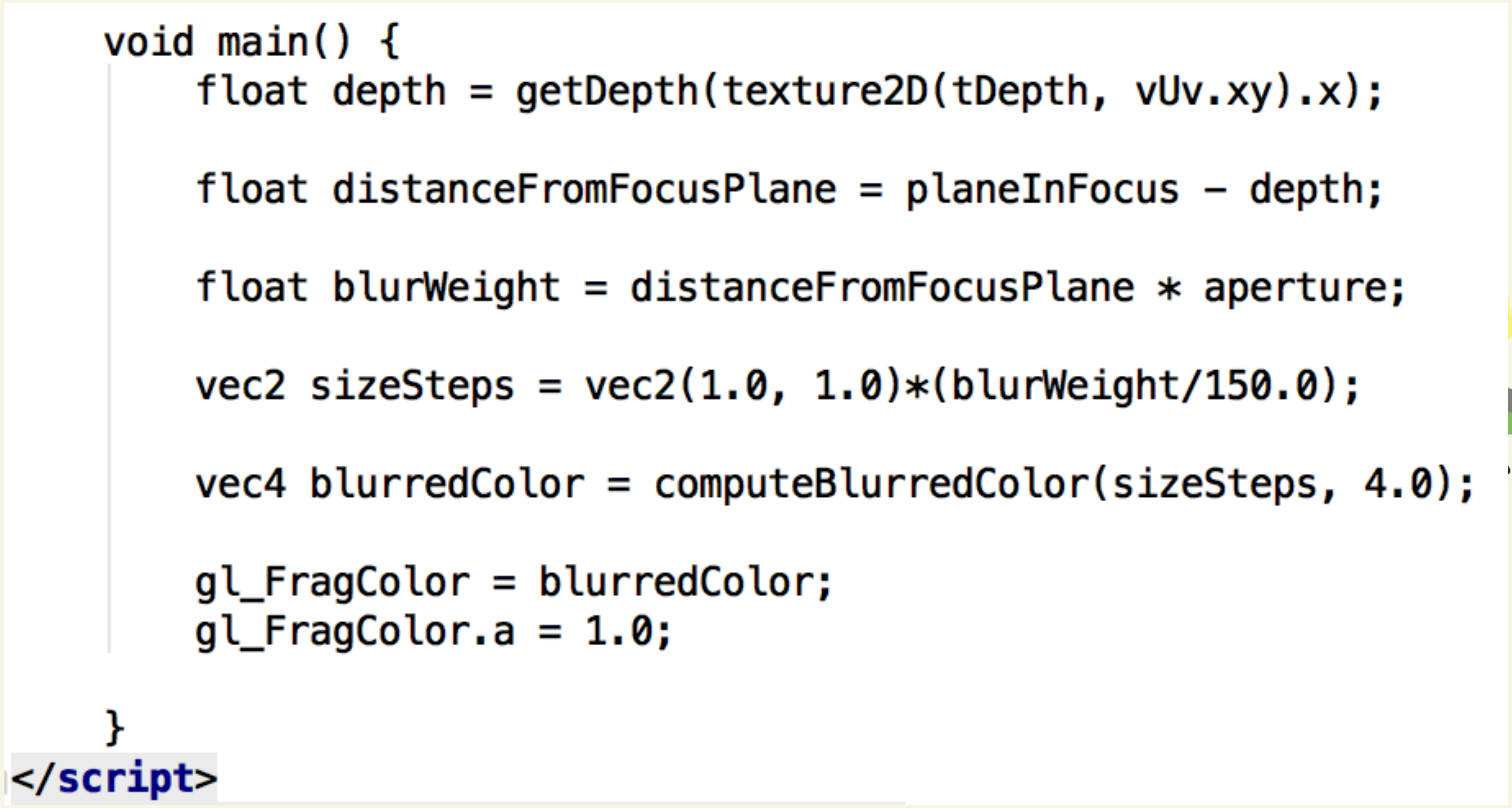
DEPTH OF FIELD – Fragment Shader



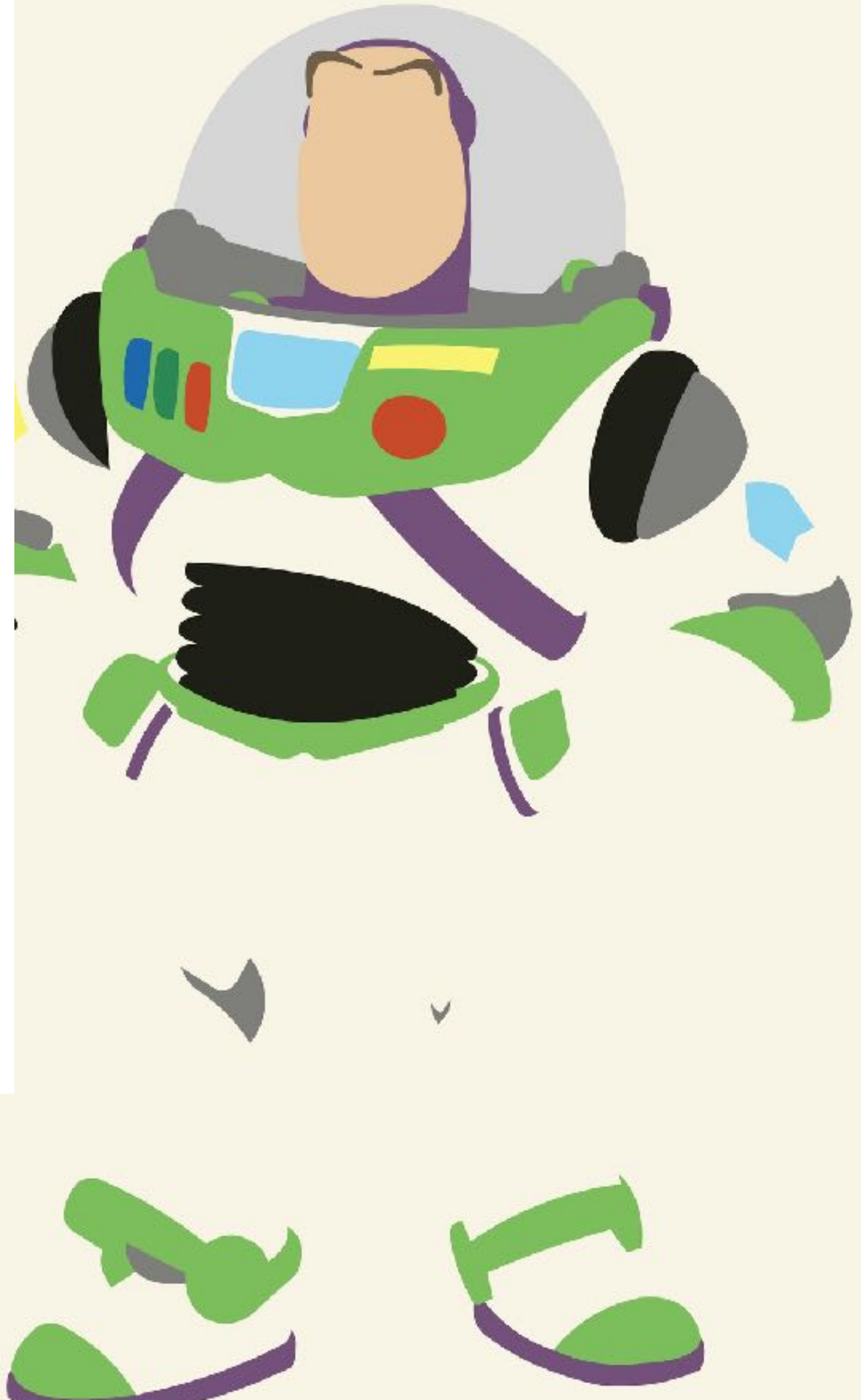
```
<script type="x-shader/x-vertex" id="DOFFragmentShader">
    #define PI 3.1415926

    varying vec2 vUv;

    //textures for DoF
    uniform sampler2D tColor;
    uniform sampler2D tDepth;

    //variables needed to process the textures
    uniform float screenWidth;
    uniform float screenHeight;
    uniform float zFar;
    uniform float zNear;
    uniform float planeInFocus;
    uniform float aperture;

    vec2 tSize = vec2(screenWidth, screenHeight);

    //function to get linearized depth
    float getDepth(float depth){
        float d = (2.0 * zNear) / (zFar + zNear - depth * (zFar - zNear));
        return d;
    }

    //function to compute gaussian weight
    float gaussianKernel(float i, float j, float variance){
        float kernelValue = (1.0/(2.0*PI*variance))*exp(-(i*i + j*j)/(2.0*variance));
        return kernelValue;
    }

    //function to compute the new color
    vec4 computeBlurredColor(vec2 sizeSteps, float variance){
        vec4 colorSum = vec4(0.0);
        for(int i = -2; i <= 2; i++){
            for(int j = -2; j<= 2; j++){
                    vec2 offset = sizeSteps*vec2(i,j);
                    colorSum = colorSum + texture2D(tColor, vUv + offset)*gaussianKernel(offset.y,offset.x,variance);
            }
        }
        return colorSum;
    }
```



DEPTH OF FIELD – Fragment Shader



```
    void main() {
        float depth = getDepth(texture2D(tDepth, vUv.xy).x);

        float distanceFromFocusPlane = planeInFocus - depth;

        float blurWeight = distanceFromFocusPlane * aperture;

        vec2 sizeSteps = vec2(1.0, 1.0)*(blurWeight/150.0);

        vec4 blurredColor = computeBlurredColor(sizeSteps, 4.0);

        gl_FragColor = blurredColor;
        gl_FragColor.a = 1.0;

    }
</script>
```



33 FPS (18-40)

TOY STORY - Depth of Field

(Arrows - Move, MOUSE - Look around)

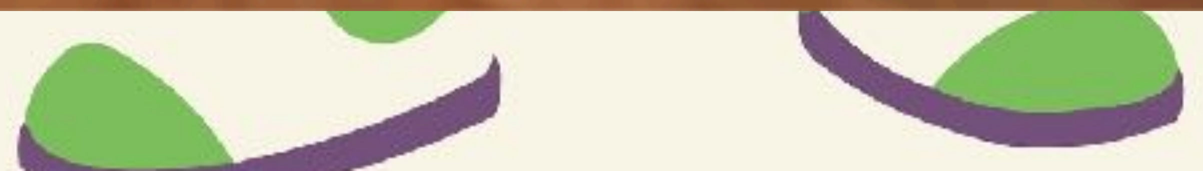
Enable DoF ✓

Parameters

focusPlane 596

aperture 5

Close Controls



CONCLUSIONS

- The best performances were reached using a MacBook Pro 2015 and Google Chrome: despite the 36 fps, the scene has been rendered without twitches.
- Safari has performed a highest frame rate (45 fps), but the scene always twitched, bringing to an unpleasant experience.
- The project is reachable at: https://riccardoyorkereali.github.io/FakeStory-DepthOfField/