Configurable Rendering Effects For Mobile Molecule Visualization

Lukas Prost[*]
Supervised by: Reinhold Preiner[†]

Institute of Computer Graphics and Algorithms
Vienna University of Technology
Vienna / Austria

Abstract

Due to their omnipresence and ease of use, smart phones are getting more and more utilized as educational instruments for different subjects, for example, visualizing molecules in a chemistry class. In domain-specific mobile visualization applications, the choice of the ideal visualization technique of molecules can vary based on the background and age of the target group, and mostly depends on the choice of a graphical designer. Designers, however, rarely have sufficient programming skills and require an engineer even for the slightest adjustment in the required visual appearance. In this paper we present a configuration system for rendering effects implemented in Unity3D, that allows to define the visual appearance of a molecule in a JSON file without the need of programming knowledge. We discuss the technical realization of different rendering effects on a mobile platform, and demonstrate our system and its versatility on a commercial chemistry visualization app, creating different visual styles for molecule renderings that are appealing to students as well as scientists and advertisement.

Keywords: Molecule Shading, Mobile, Unity3D

1 Introduction

Mobile molecule visualization can be useful for many different groups e.g, scientists and students. Yet, every target group has their own requirements due to their different purposes. Often it is up to a designer to create a visual appearance that best meets those requirements. Designers, however, rarely have the technical skills to realize their design in a graphical rendering framework on their own. An engineer has to build the design and alternate it every time the slightest adjustment has to be made. As a result, there are always at least two people required to maintain an application's update life cycle.

In this paper, we present a mobile molecule visualization implemented in Unity3D, that allows to easily modify the visual appearance with the help of a JSON configuration file. Designers can change the rendering by setting parameters in these files e.g. which shaders to use or where lights should be placed with no required programming skill whatsoever. In the remaining paper, we will demonstrate how to apply high quality rendering effects like screen space ambient occlusion, depth of field and comic shading/outline rendering in a mobile commercial chemistry visualization app (see Figure 1) and how to make them configurable using JSON files.

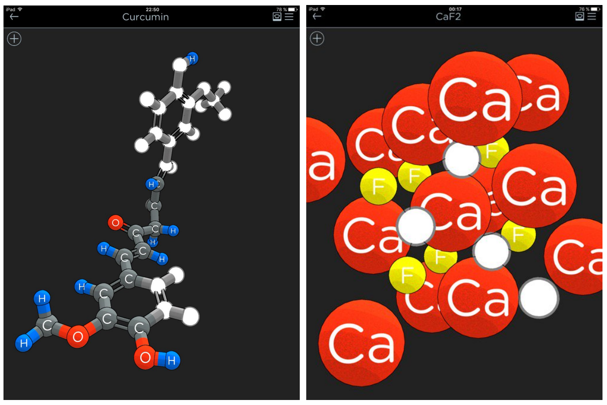


Figure 1: Exemplary screenshots of a commercial chemistry visualization app.

The rest of this paper is structured as follows: Section 2 reviews some related mobile molecule visualization apps, gives background in Unity and JSON, and discusses the related work on the rendering effects used by our system. Section 3 shows the JSON file that is used for appearance parametrization and how the textual information is used for molecule rendering. In Section 4 different rendering techniques are explained in more detail that are used to visualize molecules. Finally, in Section 5 we present some results and show different rendering styles that can be achieved by our system.

[*]lukas.prost@tuwien.ac.at
[†]preiner@cg.tuwien.ac.at

2 Background and Related Work

2.1 Molecule Visualization Systems

NDKmol [3] is an open source visualization app for smart phones. It supports many different visualization techniques like bond and ribbon diagrams and has direct access to the *RCSP Protein Data Bank* (RCSP PDB). The rendering, however, looks very plain and scientific. *RCSB PDB Mobile* [4] is a molecule visualization app that is officially provided by RCSP PDB. Because it is based on *NDKmol*, it has the same visual appearance. Molecules [2] is an alternative open source app that can load molecules directly from RCSP PDB. Unfortunately, it is only available for iOS. *Atomdroid* [10] is an app for Android with a lot functionality besides visualization. Among other things it allows to build molecules and do trajectory analysis. The last update, however, was 2012.

2.2 Unity3D

Unity3D is a free to use game engine which supports deployment for many different platforms. A Unity3D project consists of different *scenes* which be thought of as levels in a game. They contain all elements, e.g. scripts and models, and information, e.g. level architecture, required to run the scene as a program. All objects appearing in a scene are *Game Objects* (GO). A GO is the core element of Unity3D and can be thought of a container for components. The properties of a GO are defined by the components that are attached to it. These components can e.g. be a transform component, defining the GO's position, orientation and scale, or the camera component that enables the GO to render the scene. Later in the paper we will show how to manipulate a Unity-based system to define a GO's visual appearance using JSON config files.

2.3 JSON

The *JavaScript Object Notation* (JSON) [1] is an up-to-date, easy to read file format for transferring data and is mainly used in web development. It is lightweight and widely supported. JSON is used in the presented system to store visualization meta data. XML would have been the other option, yet it was dismissed because it is verbose and therefore not as legible as JSON. Data is stored as name/value pairs. While the *name* is always a string, the *value* store different types of data, ranging from simple types (number, string) to complex types like arrays or objects. An array can contain values, arrays and objects. Objects can store name value pairs. For more details about JSON and its syntax, see the JSON specification [1].

2.4 Realtime Rendering Effects

Comic Shading One technique to shade objects with a flat cartoon look is *hard shading* presented by Lake et al. [15]. The shading is done by a texture lookup based on the dot product between the normal vector and the light direction, but without interpolation resulting in a shading with few solid colors. Mitchell et al. [17] create a cartoon look without hard shading by using a 1D lookup texture and a modified Lambert lighting model. Vanderhaeghe et al. [20] present an approach for creating stylized renderings (including toon shading) dynamically by composing procedural primitives. A primitive describes a shading behavior and its parameters can be defined dynamically.

Outline Rendering Akenine-Möller, Haines and Hoffman [5, p.512] describe a heuristic method, that marks surface points as part of an object's silhouette if the dot product between the view and the normal vector is close to zero. Isenberg et al. [13] mark edges that share a front facing and a back facing polygon relative to the viewer as silhouette edges. Another approach presented by Akenine-Möller et al. [5] is the *halo* or *shell* method. An object is rendered by two passes, where the first pass renders the front faces of an object and the second pass renders its enlarged back faces. Kolivand and Sunar [14] detect silhouette edges for shadow volumes by sending a ray for each edge from the light source to one along this edge translated end vertex of the edge. If the ray does not intersect with any face of the object, the processed edge is a silhouette.

Ambient Occlusion The concept of ambient occlusion (AO) and its benefits are described by Landis [16]. An implementation is given by Pharr and Green [19]. A technique that enables dynamic real time computation is *screen space ambient occlusion* (SSAO) which first was presented by Mittring [18]. It simulates occlusion from nearby surfaces by using the depth buffer to approximately reconstruct local geometry. To do so, random samples are placed around each fragment's view space position which is then compared against the depth of the surrounding geometry using simple depth buffer lookups. The more samples are covered by the surrounding geometry, the more the fragment is occluded. Filion and McNaughton [11] describe an improved version of Mittring [18] by aligning the samples on a hemisphere around the surface normal reducing self occlusion dramatically.

Depth of Field Physically correct *Depth of Field* (DoF) rendering is presented by Cook, Porter and Carpenter [7], who simulate light distortion by ray tracing through a virtual lens. Haeberli and Akeley [12] render the scene from several slightly different view points and use the accumulation buffer to blend the renderings together into a final image. Demers [8] simulates DoF in screen space by separating the scene into layers based on the depth buffer. After blurring these layers based on their depth, the scene is composed back together resulting in a visual appealing DoF effect. Filion and McNaughton [11] present an implementation of this approach that uses five layers.

3 Appearance Parametrization

In this Section we will demonstrate how we parameterize the visual appearance of a molecule by a simple JSON configuration file, and how it is integrated in a Unity3D application to control its scene rendering.

Listing 1: Template for the JSON configuration file

```
{
"camera" : {
        "orthographic" : <boolean> ,
        "bgcolor" :
                [ <integer> , <integer> , <integer> ] }
                    ,
"mapping" : {
        <<string> : <string>>*
        "post_effects" : [ <string>* ] },
"shaders" : [
        <{
        "name" : <string>  ,
        "shader" : <string>  ,
        "properties" : { ... } },
        }>* ]
"lights" : [
        <{
        "type" : <string>,
        "position" :
                [ <integer> , <integer> , <integer> ],
        "color" : [ <integer> , <integer> , <integer> ]
             ,
        "intensity" : <float> ,
        "shadow" : <string> ,
        "strength" : <float> ,
        "movable" : <boolean> ,
        }>* ]
}
```

3.1 JSON Config File

The whole visual appearance of a scene is stored in a JSON config file (JCF). A template of its structure and syntax is shown in Listing 1. Our configuration file has four main name/value pairs:

- **Camera** has an object that stores a background color and a boolean defining whether the projection is orthographic or perspective.

- **Shaders** stores an array of shader objects. A shader object contains a name for the shader (*name*) and the name of the used shader (*shader*). The first one functions as a reference/id which is valid inside the current JCF, whereas later one is the actual name of the used shader inside the application. Moreover, a shader object contains a property object storing parameters for each individual shader. The available shaders are the Unity3D default shaders as well as custom shaders described in Section 4.

- **Mapping** has an object with name/value pairs where the name refers to an actual object in the scene and the value is the reference to a shader object in *Shaders*. The available scene objects depend on the application. *post_effects* stores an array of post processing shader references that will be applied in the order of the array.

- **Lights** stores an array that contains light objects. A light object contains all properties of a light source e.g, its type (point or directional), its position and what kind of shadow it casts (none, soft or hard).

3.2 Integration in Unity3D

The system that applies the JCF described in Section 3.1 to the Unity3D Scene consists of the three independent modules: a *ShaderProvider*, a*CameraProvider* and a *LightingProvider*. The interaction of these modules with the core entities of an Unity application is illustrated in Figure 2.

The ShaderProvider works with the data stored in *Shaders* and *Mapping*. Every Game Object (GO) that will be rendered requests a shader from this module, either by specifying its defined type or by asking for a specific shader reference. The ShaderProvider first checks if the requested shader is available. If this is the case, it loads the shader from the system, sets the properties stored in the corresponding JCF shader object and then applies it to the requesting GO. Besides providing shaders for GOs, it can also provide post processing shaders for the camera.

The CameraProvider reads the parameters specified by *Camera* in the JCF and modifies the parameters of the Unity camera component accordingly. In a similar way, the LightingProvider module processes the data given by *Lights*. For each JCF light object a new light is placed in the Unity scene.

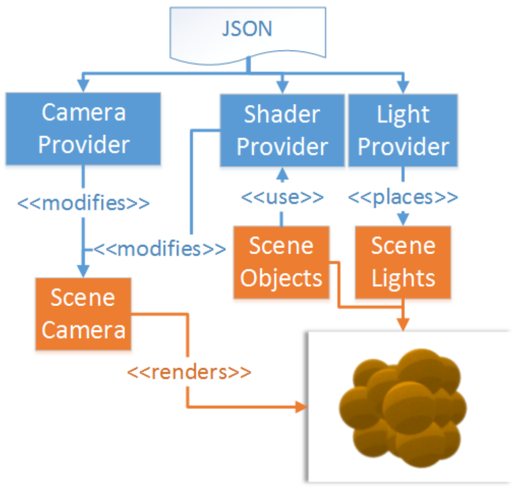


Figure 2: Relation of the Config Loader Modules (blue) to the Unity3D Scene Elements (orange).

4 Molecule Rendering Effects

In this section, we describe three major rendering effects that are supported in our molecule visualization app, and discuss their realization in a mobile real-time rendering framework.

4.1 Comic Shading and Outlining

To support a non-photo realistic, stylized look, a highly customizable shader was implemented to provide a high degree of visual variety to the designer. The shader does actual hard shading and allows outline rendering in object space. For the latter, both the dot product method and the hull method of Akenine-Möller et al. [5] were implemented. Both provide a good trade-off between performance and results depending on the model.

Comic Shading To achieve an efficient comic style that is easy to configure by a designer, we implemented hard shading without texture lookup. Using a texture-based approach, the designer would have to provide a texture for every single GO, which would not be possible just based on a modification in the JCF. The basis for the brightness calculation is a modified Lambert term, shown in Equation 1 [17]. The original Lambert reflection model is extended by scale constant α, a bias β and an exponent γ.

$$(\alpha(\hat{n}\cdot\hat{l})+\beta)^{\gamma} \tag{1}$$

These constants can also be configured by the user in the JCF and allow to modify the distribution of the hard shading borders. To achieve a hard shading look, Equation 1 is discretized by clamping, resulting in the final shading formulation:

$$\frac{(\alpha+\beta)^{\gamma}}{s}\left\lfloor\frac{s(\alpha(\hat{n}\cdot\hat{l})+\beta)^{\gamma}}{(\alpha+\beta)^{\gamma}}\right\rfloor \tag{2}$$

The subdivision parameter s defines the number of gradients/shading borders and is again configurable by the JCF.

Figure 3 shows examples of hard shaded atoms with different subdivision parameters s.

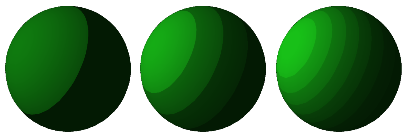


Figure 3: Comic shaded spheres ($\alpha = 0.5$, $\beta = 0.5$ and $\gamma = 2$) with $s = \{2,4,8\}$ from left to right.

Outline Rendering For outline rendering [5], we need to calculate the dot product between the view vector and normal vector at each pixel. If the result lies below a specific user defined threshold (typically values between 0.25 and 0.5), the pixel gets rendered in a border color. Both the threshold and the border color are parameters that can be set by the user in the JCF. This algorithm is very efficient because it adds only one additional dot product and comparison evaluation to the pixel color. An outline of a sphere rendered with this method can be seen in the upper row of Figure 4.

The second outline rendering technique available in our system is the hull method [5]. This method creates an outline by first enlarging a model and then rendering its backfaces. The enlarging is done by a vertex translation along the vertex normal. To do so, the vertex and its corresponding normal vector need to be transformed into view space. Then, the vertex is translated along the x and the y coordinate of the normal vector. The length of the translation defines the hull size and can be modified by the user. This value depends on the size of the objects because it is happening in view space. For the atoms, it is normally between 0.005 and 0.03. After the translation, the front faces are culled and the back faces are rendered with the defined border color. The results can be seen in the lower row of Figure 4.

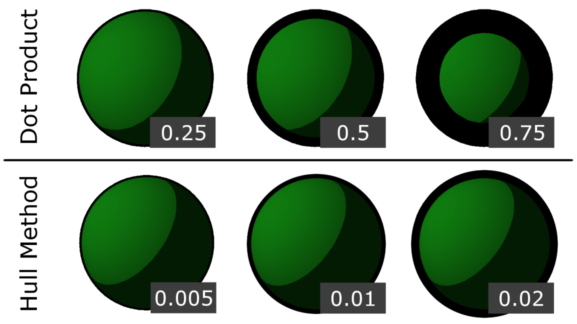


Figure 4: Outlines rendered with different methods. The numbers show the dot product threshold (upper row) and the hull size (lower row).

4.2 Screen-Space Ambient Occlusion

SSAO is a fast screen space effect suitable for mobile real time applications, that can greatly enhance the visual quality of the resulting images. It uses the depth buffer as a discretized representation of the scene, based on which it estimates the ambient occlusion for each pixel in screen space.

To this end, we first need to calculate the view space position of the pixel. Samples are then placed around this point by adding a set of predefined offset vectors to their view-space position. Each new sample point is then projected back to screen space where their z-values are compared to the stored depth at their target screen space position (similar to shadow mapping). Each neighboring sample with a z-value larger than the stored depth increases the ambient occlusion value of the center pixel. To keep the ambient occlusion value independent of the number N of used samples, it is normalized by N. Because this procedure is repeated for every pixel with the same samples, this process can also be seen as a convolution of the discretized scene with a sample kernel.

For high-quality AO effects, Christensen [6] recommends at least 256 samples. Yet, such a high sample count is not feasible for mobile real-time rendering. Therefore, Engel [9] presents an SSAO implementation that achieves a moderate result with only 16 samples. Without further regard, such a small sample count would lead to a visible pattern of the sample kernel. For this reason, this technique randomly rotates the sample kernel for each pixel using a random rotation matrix. Filion and McNaughton [11] suggest using a random vector provided by a noise texture instead of random rotation matrices. Each offset vector of the sample kernel is reflected by this offset vector resulting in a semi randomization of the kernel per pixel. Since it is the most efficient SSAO variant, we choose this random-vector-based technique for our mobile real-time rendering system.

The randomization of the sample kernel reduces kernel artifacts, but results in a coarse SSAO image. For this reason, the buffer that stores the SSAO values are blurred with a kernel that should be small enough ($4x4$ pixels) to preserve borders as good as possible.

Finally, each pixel of the rendered scene is darkened by its corresponding value in the SSAO texture. Because these values lie in the interval $[0,1]$, they can simply be multiplied to the pixels color. The effect of SSAO on the visual expressiveness of a scene is demonstrated in Figure 5.

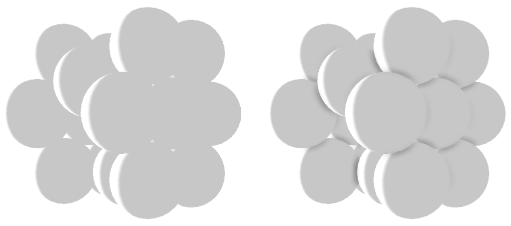


Figure 5: Scene rendered without SSAO (left) and with SSAO (right)

4.3 Depth of Field

Ray-tracing Depth of Field effects, as suggested by Cook et al. [7], or accumulation of multiple render passes as proposed by Haeberli and Akeley [12] would be too costly for a real-time performance on mobile platforms. Therefore, DoF is applied as a post processing effect based on the method of Filion and McNaughton [11].

Based on the values in the depth buffer, the screen-space representation of the scene is divided into five depth layers, as shown in Figure 6. Each layer is defined by a range that can be set by the designer in the JCF. A texel is assigned to a layer if its depth value falls into the layers depth range. The designer can define the five layers by setting the four border depth values TR[0], TR[1], TR[2] and TR[3] between them. The relation between the TR values have to be TR[0] $\leq$ TR[1] $\leq$ TR[2] $\leq$ TR[3].

The DoF effect is applied in four steps. The first step separates the scene based on the given ranges into layer's. The near and the far layer are stored in two separate frame buffers. If a texel's depth is an element of e.g. the near layer, its color and depth are rendered into the near layer frame buffer. Otherwise, the texel is rendered with the camera's clear color. The focus layer does not get stored in a separate frame buffer. Instead, the unprocessed input frame buffer is used. After the pixels are assigned, the near and the far layer frame buffers are blurred with a separated Gaussian. Finally, the layers are composed together based on the same ranges as were used for their separation. If a depth value is in the range of a transition layer, the resulting texel is determined by interpolating between the texels of the two neighbor layers.

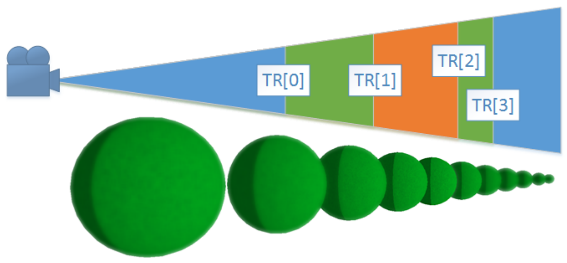


Figure 6: DoF layers (l.t.r): near, transition near to focus, focus, transition focus to far, far.

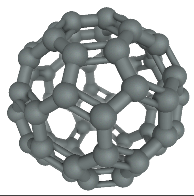
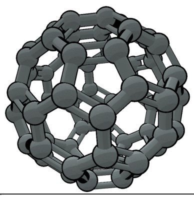
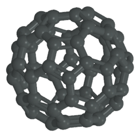
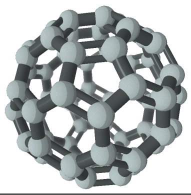
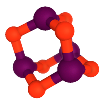
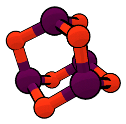
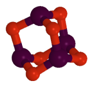
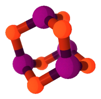
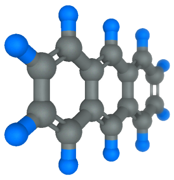
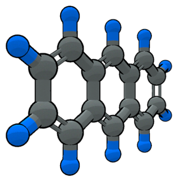
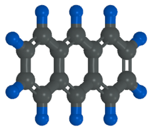
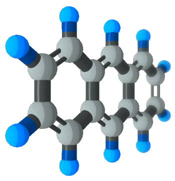
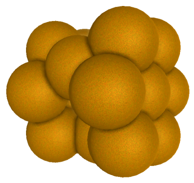
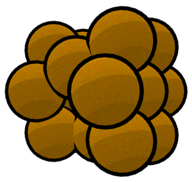
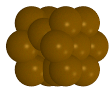
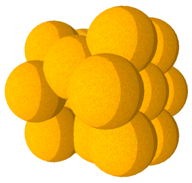
5 Results

Our configurable rendering system can be easily used by people without any programming skills. An engineer has to implement shaders in our system only once. After he made them accessible for the configuration loader, the designer can apply and modify them as he wishes. In the following, we will give an example of four different designs that can be produced in our system, and show their performance on several mobile devices.

5.1 Visual Designs

Table 1 presents four different looks that were produced in our system only by manipulating the JSON config file. The table shows an outline of these config files, and illustrates the resulting visual appearance on four different molecules. The shader referred to as *basic* is the standard shader provided by Unity3D. The shader's *dofPost* and *ssaoPost* denote the post-processing DoF and SSAO shaders, respectively. Finally, *toon* addresses the object space comic shader.

Education The visual appearance for students was created with a simple and flat design in mind. Flat and tactile looking interfaces are currently modern and widely used. Moreover, the design tries to support a visual gamification to be appealing for this target group. To this end, comic shading and outline rendering was used.

	Advertisement	Education	Scientific	Future
Fulleren(C_{60})				
Arsenik(As_2O_3)				
Anthracen($C_{14}H_{10}$)				
Magnesium(Mg)				

```
"camera" : { <...> },
"mapping" : {
  "atom" : "advert",
  "connector" : "advert
      ",
  "post_effects" : [
   "dof", "ssao"] },
"shaders" : [
  { "name" : "advert",
  "shader" : "basic",
  "properties" : {
  "smoothness" : 0.15,
  "metallic": 0.3 }
  },
  { "name" : "dof",
  "shader" : "dofPost",
  "properties" : {
  "layers" :
        [0,0,0.7,0.9]}
  },
  { "name" : "ssao",
  "shader" : "ssaoPost"
      ,
  "properties" : { }
  } ],
"lights" : [ <...> ]
```

```
"camera" : { <...> },
"mapping" : {
  "atom" : "school",
  "connector" : "school
      ",
  "post_effects" : []
},
"shaders" : [
  { "name" : "school",
  "shader" : "toon",
  "properties" : {
  "color" : [0,0,0],
  "hull_size" : 0.45,
  "outline_bias" : 0.0,
  "scale" : 0.5,
  "bias" : 0.65,
  "exponent" : 1,
  "steps" : 4 }
  } ],
"lights" : [ <...> ]
```

```
"camera" : {
  "orthographic" : "
       true",
  ...
}
"mapping" : {
 "atom" : "science",
 "connector" : "science
      ",
 "post_effects" : []
},
"shaders" : [
  { "name" : "science",
  "shader" : "basic",
  "properties" : {
  "smoothness" : 0.5,
  "metallic": 0.2}
  } ],
"lights" : [ <...> ]
```

```
"camera" : { <...> },
"mapping" : {
  "atom" : "future",
  "connector" : "basic"
      ,
  "post_effects" : ["
       dof"] },
"shaders" : [
  { "name" : "basic",
  "shader" : "basic",
  "properties" : {
  "smoothness" : 0.5,
  "metallic": 0.2}
  },
  { "name" : "future",
  "shader" : "toon",
  "properties" : {
  "color" : [0,0,0],
  "hull_size" : 0.0,
  "outline_bias" : 0.0,
  "scale" : 0.5,
  "bias" : 0.65,
  "exponent" : 1,
  "steps" : 4 }
  },
  { "name" : "dof",
  "shader" : "dofPost",
  "properties" : {
  "layers" :
        [0,0,0.7,0.9]}
  } ],
"lights" : [ <...> ]
```

Table 1: Different molecules rendered with different styles and summarized JCF defining the visualizations.

Advertisement To sell a CG-related product, its visual output has to look as stunning as possible. Moreover, the quality of the renderings used in advertisement directly reflect the public image of the company producing the software. Therefore, SSAO as well as DoF are applied to the rendering. By doing so, the molecule looks much more plastic and ”realistic”.

Scientific For science, the visual appearance should support the understanding of the structure of a molecule. The rendering should avoid any additional effects that clutter the image of a molecule. For this reason, standard shading was used without any effects and the camera uses an orthographic projection. This aims at supporting the understanding of the structure of a molecule.

Future This look was created to demonstrate how object types can be shaded differently. The visibility of the atoms gets significantly enhanced by applying a bright comic shader to the atoms and a dark basic shader to the connectors,.

5.2 Performance

Performance data for the advertisement, education and scientific look is shown in Figure 7 for a smart phone and a tablet. The used smart phone was a *OnePlus One* with a resolution of 1920x1080. The tablet data was gathered from a *Nvidia SHIELD TABLET K1* with a resolution of 1920x1200. The performance data was gathered by rendering each example for a short period of time. The FPS value was taken in short intervals. The final results for each example are the average over the collected FPS values.

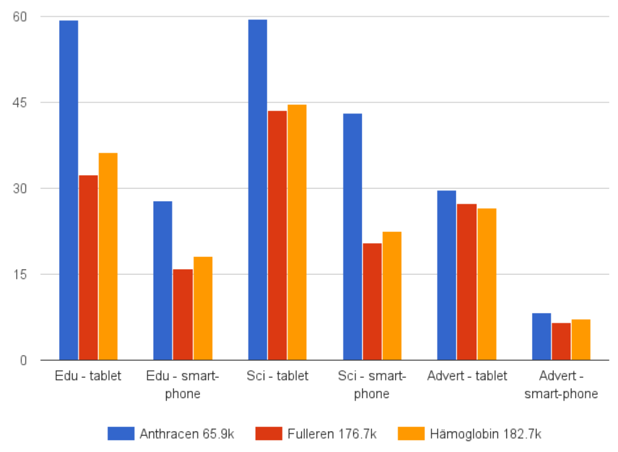


Figure 7: Performance data. Polygon number next to the name.

The performance values indicate an expected dependence on the polygon count of the model. The *Education* and *Science* styles run at acceptable rates even for big models. For all our models, the *Advertisement* style is the computationally most demanding one, with under 10 FPS on smart-phones. This can be attributed to the usage of the Depth of Field effect. This performance is acceptable, as this style is mostly meant for creating still shots used in advertisement.

6 Conclusion and Future Work

We have presented a system that allows to define the visual appearance of rendered molecules only by modifying parameters in a JSON configuration file. The syntax and the structure of this file is simple to read and easy to understand, such that even people without deeper knowledge about rendering and shaders can change the visual appearance of a scene easily. Since the shown system reads the JSON file during run-time, the rendering style can be changed fast without the need of rebuilding the application. This supports prototyping with fast and easy adjustments. A GUI with live feedback would simplify the process even more. This is left open for future work.

References

[1] Json specification. `http://www.json.org/`. Accessed: 13-02-2016.

[2] Molecules. `http://www.sunsetlakesoftware.com/molecules`. Accessed: 13-02-2016.

[3] Ndkmol. `http://webglmol.osdn.jp/`. Accessed: 13-02-2016.

[4] Rcsb pdb mobile. `http://www.rcsb.org/pdb/static.do?p=mobile/RCSBapp.html`. Accessed: 13-02-2016.

[5] T. Akenine-Möller, E. Haines, and N. Hoffman. *Real-time rendering*. CRC Press, 2008.

[6] P. H Christensen. Global illumination and all that. *SIGGRAPH 2003 course notes*, 9, 2003.

[7] R. L. Cook, T. Porter, and L. Carpenter. Distributed ray tracing. *SIGGRAPH Comput. Graph.*, 18(3), 1984.

[8] J. Demers. Depth of field: A survey of techniques. *GPU Gems*, 1(375), 2004.

[9] W. Engel. Shaderx7. *Charles River Media*, 2009.

[10] J. Feldt, R. A Mata, and J. M Dieterich. Atomdroid: a computational chemistry tool for mobile platforms. *J. of chem. inf. and modeling*, 52(4), 2012.

[11] D. Filion and R. McNaughton. Effects & techniques. In *ACM SIGGRAPH 2008 Games*, SIGGRAPH ’08. ACM, 2008.

[12] P. Haeberli and K. Akeley. The accumulation buffer: Hardware support for high-quality rendering. In *Proc. of the 17th Annu. CC on CG and Interactive Techniques*. ACM, 1990.

[13] T. Isenberg, B. Freudenberg, N. Halper, S. Schlechtweg, and T. Strothotte. A developer's guide to silhouette algorithms for polygonal models. *CG and AP, IEEE*, 23(4), 2003.

[14] H. Kolivand and M. S. b. Sunar. New silhouette detection algorithm to create real-time volume shadow. In *DMDCM, 2011 Workshop on*, 2011.

[15] A. Lake, C. Marshall, M. Harris, and M. Blackstein. Stylized rendering techniques for scalable real-time 3d animation. In *Proc. of the 1st Int. Symp. on Non-photorealistic animation and rendering*. ACM, 2000.

[16] H. Landis. Production-ready global illumination. *Siggraph course notes*, 16(2002), 2002.

[17] J. Mitchell, M. Francke, and D. Eng. Illustrative rendering in team fortress 2. In *Proc. of the 5th Int. Symp. on Non-photorealistic animation and rendering*. ACM, 2007.

[18] M. Mittring. Finding next gen: Cryengine 2. In *ACM SIGGRAPH 2007 courses*. ACM, 2007.

[19] M. Pharr and S. Green. Ambient occlusion. *GPU Gems*, 1, 2004.

[20] D. Vanderhaeghe, R. Vergne, P. Barla, and W. Baxter. Dynamic stylized shading primitives. In *Proc. of the ACM SIGGRAPH/Eurographics Symp. on Non-Photorealistic Animation and Rendering*, NPAR '11. ACM, 2011.