Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Computer Graphics and Interaction

DIPLOMA THESIS ASSIGNMENT

Student: **Bc. Tomáš Pastýřík**

Study programme: Open Informatics
Specialisation: Computer Graphics and Interaction

Title of Diploma Thesis: **Visualization of inner structure of complex 3D objects based on opacity modulation**

Guidelines:

Analyze approaches that are using transparency to visualize inner structure of complex 3D objects (objects composed from many parts - e.g., gear box). Design and implement an algorithm utilizing per-pixel sorting of fragments [1, 2]. Compare the rendering speed of the implemented algorithm with at least two different algorithms that are using transparency to visualize inner structure of complex 3D objects (e.g., [3] and [4]). Perform the comparison of rendering speed on at least ten 3D objects of various complexities. For the compared algorithms also analyze to what extent we can modify the rendering equation and what modifications in the algorithm it requires.

Bibliography/Sources:

[1] R. Carnecky, R. Fuchs, S. Mehl, J. Yun, and R. Peikert, "Smart Transparency for Illustrative Visualization of Complex Flow Surfaces," IEEE Transactions on Visualization and Computer Graphics, vol.19, no.5, pp.838-851, 2013.

[2] J.C. Yang, J. Hensley, H. Grün, and N. Thibieroz, "Real-Time Concurrent Linked List Construction on the GPU," Computer Graphics Forum, vol.29, no.4, pp.1297-1304, 2010.

[3] C. Everitt, "Interactive order-independent transparency," Technical report, NVIDIA Corporation, 2001.

[4] L. Bavoil and K. Myers, "Order Independent Transparency with Dual Depth Peeling," Technical report, NVIDIA Corporation, 2008.

Diploma Thesis Supervisor: Ing. Ladislav Čmolík, Ph.D.

Valid until the end of the summer semester of academic year 2015/2016

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta elektrotechnická
L.S.

prof. Ing. Jiří Žára, CSc.
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, November 4, 2014

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Master's Thesis

Visualization of inner structure of complex 3D objects based on opacity modulation

Bc. Tomáš Pastýřík

Supervisor: Čmolík Ladislav Ing., Ph.D.

Study Programme: Open Informatics

Field of Study: Computer Graphics and Interaction

January 5, 2015

Aknowledgements

I would like to thank Ing. Ladislav Čmolík, Ph.D. for his kind guidance and valuable consultations and advices when supervising this thesis. I extend my gratitude to my parents, friends and loved ones for their love and support which guided me through all my life.

Declaration

Abstract

This thesis addresses problem of rendering semi-transparent objects and the enhancement by opacity modulation to reveal object's internal structures. Illustration Buffer - algorithm solving the *Order Independent Transparency (OIT)* is implemented and compared to other OIT solving techniques: *Depth Peeling, Dual Depth Peeling* and *Per Pixel Linked Lists.* We focus on opacity modulation based on object's features and we apply such modulations to the Illustration Buffer while comparing the ease of use with other algorithms. Finally we test all mentioned algorithms and conclude which algorithm is better under specific circumstances.

keywords: order independent transparency, illustration buffer, comparison, depth peeling, dual depth peeling, per pixel linked lists, opacity modulation, inner structure visualization

Abstrakt

Tato práce se zabývá problémem zobrazování částečně průhledných objektů a jeho vylepšení pomocí modulace průhlednosti za účelem odhalení vnitřních struktur 3D těles. Implementovali jsme metodu Illustration Buffer řešící problém *Order Independent Transparency (OIT)*, kterou porovnáváme s algoritmy, které takové zobrazování řeší: *Depth Peeling, Dual Depth Peeling* a *Per Pixel Linked Lists.* Aplikujeme metody modulace průhlednosti založené na vlastnostech objektů na Illustration Buffer a zkoumáme obtížnost použití v porovnání s ostatními zmíněnými algoritmy. Algoritmy řešící OIT problematiku testujeme mezi sebou a ukazujeme, za jakých podmínek je který algoritmus lepší.

klíčová slova: order independent transparency, illustration buffer, porovnání, depth peeling, dual depth peeling, spojové seznamy pro pixely, modulace průhlednosti, vizualizace vnitřní struktury

Contents

List of Figures

List of Tables

Chapter 1

Introduction

There are many challenges in rendering semi-transparent objects. The most important one is to display such objects in a way users understand and can determine their inner structure and shape from one image. It is vital for the users to be able to use their experience from the real world and connect it to what they see to create the final mental model. Considering complex objects the user might not be able to construct this model at all if the object is opaque or transparency of all layers is homogeneous. This implies the need for some image enhancement with knowledge of local geometry based on human perception. Even if we are able to enhance image in this way it might not be enough for the user to understand the structure of examined object correctly. Therefore interactivity is a desired property in object visualization so that user can manipulate the object in space.

When we are rendering semitransparent 3D objects on GPU we can choose from variety of order independent transparency (OIT) algorithms. Yet, it is not clear at what conditions is one algorithm better than another. In other words, when it is worth to invest time and resources to implement more complicated algorithm. In this thesis we therefore address this problem by comparison of several OIT solving algorithms: *Depth Peeling, Dual Depth Peeling, Per Pixel Linked Lists* and the *Illustration Buffer.*

Our implementation is based on one of the latest techniques called the *Illustration Buffer*[12]. However authors focus mainly on flow surfaces while their technique of building the buffer containing the scene can be used in general. When such a buffer - called *Illustration Buffer* is obtained, every single fragment knows its neighbors along the surface as well as along the viewing ray. This knowledge can be later used to determine the most important sections of image for the user such as edges, junctions, silhouettes...

Even though main goal of this thesis is to implement the technique described in [12] we also compare it to other known techniques solving the OIT problem: Concurrent per Pixel Linked Lists, Depth Peeling and Dual Depth Peeling.

1.1 Thesis Structure

Necessary theory background is introduced in Chapter 2, where OIT problem is defined along with its dual representation. Methods of final image composition are introduced as well as the categorization of opacity modulation techniques. Chapter 3 compares discussed OIT solving algorithms with great focus on their usability in several opacity modulation approaches. Chapter 4 is rather practical. The application structure, used technologies and details on the Illustration Buffer implementation are provided. Finally we discuss our results in Chapter 5 and Chapter 6 concludes the thesis.

Chapter 2

Theoretical Background

To achieve and describe some advanced opacity modulation techniques a related theory should be introduced first. This chapter introduces the problem of rendering translucent objects, describes its dual representation that will provide us further flexibility when comparing rendering methods. Finally, methods of final image composition are introduced.

For the purpose of this work let us define *pixel* as the final image element presented to the user. *Pixel* is therefore the smallest element user can address. Considering traditional rendering pipeline (e.g. in OpenGL 5.3), *fragments* are defined as a candidate for a pixel. Both *pixels* and *fragments* contain the colour as well as the depth as the distance to the camera.

The *z-buffer* of the same size as current framebuffer is defined to choose the final pixels containing the depth of the closest pixels to the camera. It is the depth test that turns fragments to pixels:

$$deptBuffer(x,y) = \begin{cases} unchanged & \text{if } fragment.depth > depthBuffer(x,y) \\ fragment.depth & \text{if } fragment.depth <= depthBuffer(x,y) \end{cases}$$

If no depth-buffer is used during the rendering process, pixels are being rewritten by any later fragments, regardless the depth. The order of incoming fragments in general case is not defined.

2.1 Rendering of Translucent Objects

Since rendering of the translucent geometry needs to take into consideration all the translucent fragments, the depth-buffer cannot be used. Without the depth-buffer we do not discard any fragments due to their depth but we have to solve the order of rendered geometry to get correct result, as shown in figures 2.1 and 2.2. Numbers corresponds to the distance to the camera with a red layer being the closest and the blue one the furthest. As it is described in the section 2.1.1, sorting all the triangles of the scene might not always work. Therefore sorting in bigger detail is required and all of the principles described in the section 2.2 are based on rendering the fragments in the right order, not triangles.

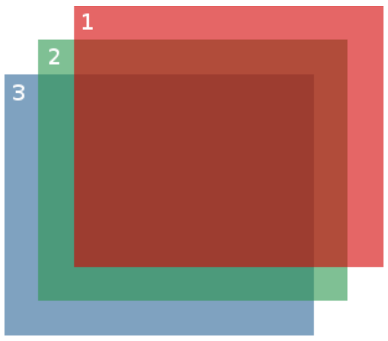


Figure 2.1: Correct order (3, 2, 1)

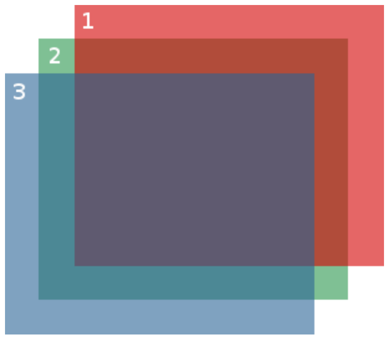


Figure 2.2: Incorrect order (2, 1, 3)

2.1.1 Painter's Algorithm

The *Painter's algorithm* is the simplest method to partially solve the problem of the wrong order. It simply sorts the elements (triangles) in back-to-front order before rendering them. This method however is not completely reliable as it fails to solve following situation shown in figure 2.3, where triangles create a *depth cycle*.

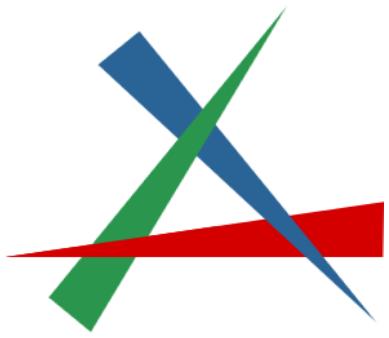


Figure 2.3: Unsolvalble problem for Painter's algorithm

2.2 Order Independent Transparency

As shown in section 2.1.1, sorting the elements (in our case triangles) before rendering them might not be enough. Order *Independent Transparency (OIT)* is a class of techniques that do not require sorting the elements since it is the order of fragments that is more important.

This section describes possible representations of the problem, it shows how to compose final pixel colour from gathered fragments (samples) and presents a theoretical background for opacity modulation methods we use in this work. At the end of the section we also discuss a basic use of colour modulation to provide a base for future discussion.

2.2.1 Dual Representation of the Problem

Techniques that solves the *OIT* problem are described in detail in Chapter 3. However to be able to compare such techniques the following dual representation of *OIT* solution needs to be introduced[25]. Please note that this section does not describe actual researched algorithms, only defines the terminology.

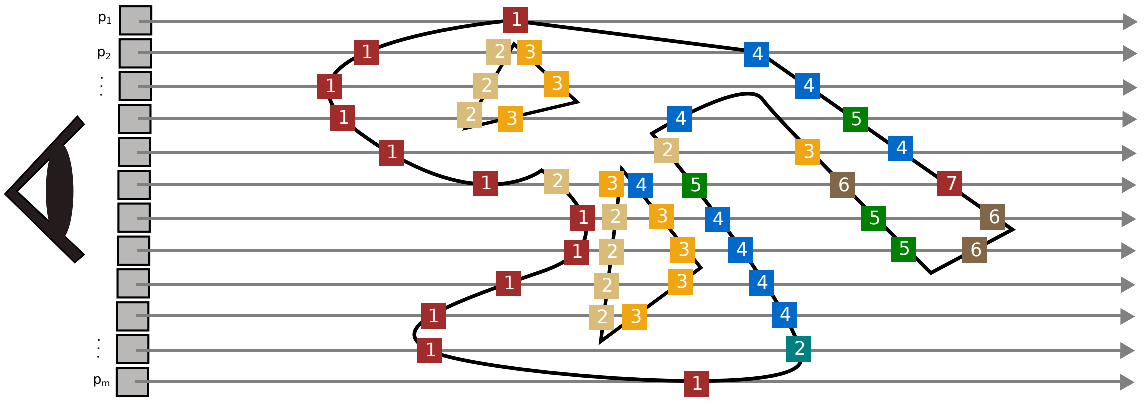


Figure 2.4: For the sake of simplicity *one row display* of pixels $p_i, i \in \{1, ..., m\}$ where m is the number of pixels is shown. Fragments with the same number are in the case of *peeling* methods in the same layer. In case of *ray casting* terminology, numbers denote the order along the ray.

Ray Casting

First way of defining the *OIT* problem is using the *ray casting* terminology, even though algorithms described in the Chapter 3 do not use the *ray casting* algorithm directly. If fragments along the ray beginning in the camera and intersecting the center of pixel are denoted as $f_i, i \in \{1, ..., n\}$ where n is the number of fragments that intersect with such ray, algorithm works as follows:

Algorithm 1: OIT solution using the ray casting terminology

```
1 for all pixels do
2     1. Cast the ray from the camera through the pixel's center.
3     2. Sort fragments $f_i, i \in \{1, ..., n\}$ along the ray by their depth - distance to the camera. Set the
      indices of $f_i$ so that $\forall (i, i-1), i \in n, f_i.depth > f_{i-1}.depth$
4     3. Compose the final colour along the ray (further discussed).
5 end
```

Figure 2.4 shows this analogy using numbers to illustrate the order increasing with the sample distance from the camera. However as shown in section 2.2.2, the direction of ordering samples depends on selected method of composing the image - from *front to back* or *back to front*.

Layers

Terminology of ray casting however might not be suitable for some algorithms solving OIT problem, especially methods based on *peeling*. Layers are denoted as $l_i, i \in \{1, ..., n\}$ where n is the number of total layers created. Layer l_i than consists of all fragments, that would have index i in the Algorithm 4.

Algorithm 2: OIT solution using layers terminology

```
1 while layer to be processed exists do
2     1. Render geometry.
3     2. Peel (ignore) any previously processed layers.
4     3. Store all pixels of the current layer (denoted by the same number in figure 2.4 to a separate
      texture.
5 end
6 Blend all layers to one image using techniques described in section 2.2.2.
```

This results in image representation split into n sorted layers stored as separate images. This differs from the previous method where each fragment is stored separately.

2.2.2 Image Compositing

When compositing the pixel final colour the order of incoming fragments is extremely important. Fragments need to be sorted along the ray as shown in figure 2.4 in front to back or back to front order. Each direction of sorting requires specific blending approach using over or under operator when composing the final colour. Following equations using both operators are derived from equation shown in Chapter 11.4 of the book Moderní počítačová grafika[24].

For both operators let us consider two samples, front sample s_f with colour c_f and opacity α_f and the back sample s_b with colour c_b and opacity α_b. Sample is considered fully transparent for $\alpha = 0$ and fully opaque for $\alpha = 1$. Let the colour of resulting sample be denoted as C of opacity α.

Following equation is used in terms of alpha blending[24] to solve the problem of rendering semi-opaque objects in back to front manner (as the Painter's algorithm does (section 2.1.1)):

$$C = \alpha_f c_f + (1 - \alpha_f)[\alpha_b c_b + (1 - \alpha_b)c_{background}] \tag{2.1}$$

In definitions of *over* and *under* operators we omit the $c_{background}$ on purpose since it can always be replaced as additional layer of samples.

2.2.2.1 Under Operator

Under operator is used when compositing samples in front to back manner. Equation 2.1 needs to be reformulated recursively for compositing final colour from the samples $s_i, i \in \{1, ..., n\}$ where n is the number of samples to be composed along the ray in opposite direction:

$$C = \alpha_1 c_1 + (1 - \alpha_1)(\alpha_2 c_2 + (1 - \alpha_2) \ldots (\alpha_{i-1} c_{i-1} + (1 - \alpha_{i-1}) \alpha_n c_n) \tag{2.2}$$

$$C = \alpha_1 c_1 + \alpha_2 c_2 (1 - \alpha_1) + \ldots + \alpha_i c_i (1 - \alpha_1) \cdot \ldots \cdot (1 - \alpha_n) \tag{2.3}$$

The *under operator* then substitutes following relationships from the equation 2.3:

$$\begin{aligned} C_{out} &= \alpha_{in} c_{in} + \alpha_i c_i (1 - \alpha_{in}) = \alpha_i c_i \textbf{ under } \alpha_{in} c_{in} \\ \alpha_{out} &= (1 - \alpha_{in}) * \alpha_i \end{aligned} \tag{2.4}$$

where α_i, c_i are opacity and colour accumulated by previous operator applications. While we get colour C_{out} from equation 2.4, we need the α_{out} as separate value to be able to accumulate opacities through the composition. Please note that another notation is sometimes used by using transparency defined as $T_i = (1 - \alpha_i)$ which changes the α_{out} relationship to $T_{out} = (T_{i-1})(1 - \alpha_i)$ which is in fact the same only using transparency instead of the opacity.

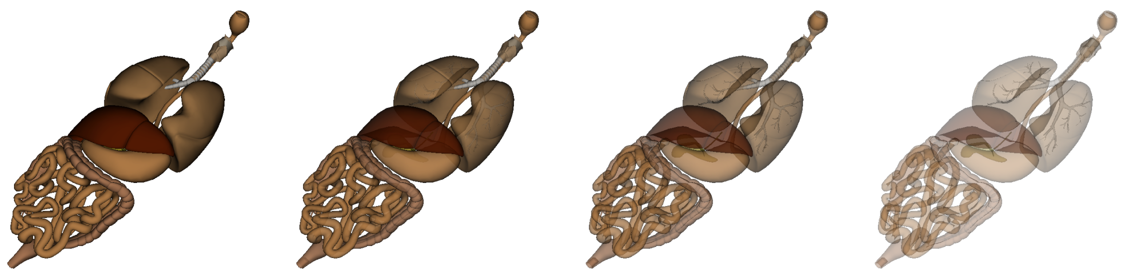
2.2.2.2 Over Operator

In some cases we may need to compose samples in back to front order. For that we need to define *over operator* since in equation 2.3 we cannot swap the front and back colours, it is not commutative operation. Intuitively from equation 2.1, the over operator is:

$$\begin{aligned} C_{out} &= \alpha_i c_i + \alpha_{in} c_{in} (1 - \alpha_i) = \alpha_i c_i \textbf{ over } \alpha_{in} c_{in} \\ \alpha_{out} &= (1 - \alpha_{in}) * \alpha_i \end{aligned} \tag{2.5}$$

where α_i, c_i are opacity and colour accumulated from previous operator applications again.

2.3 Opacity Modulation



a) $\alpha = 1.0$ b) $\alpha = 0.75$ c) $\alpha = 0.5$ d) $\alpha = 0.25$

Figure 2.5: Decreasing opacity globally for the whole model causes the internal structures to appear in exchange for the loss of shape perception.

Opacity modulation is the most important way of affecting how the subject percepts the internal structures of complex 3D objects. It is also accompanied by the colour modulation which can help with highlighting specific sets of object features, as discussed further in section 2.4. This section explains techniques considering the object shape in local context, important shape features in global context, the importance, grouping and the transfer functions. Please note that there are more techniques to achieve perceptively pleasant results in volumetric rendering but only some of them are mentioned for the purpose of this work. Some techniques from volumetric rendering that can be applied in our case with geometry input are included in this section.

2.3.1 Modulation by Differences in Shape

Differences in shape, also called object's *curvature*, can be easily found by following technique depicted in figures 2.6 and 2.7. The curvature of current sample (fragment) with normal $\vec{n}$ depends on distances from normals of its neighbors as shown in figure 2.7. Distances of normalized normals are than summed together to get the curvature value[16].

$$Curvature = |\vec{n} - \vec{a}| + |\vec{n} - \vec{b}| + |\vec{n} - \vec{c}| + |\vec{n} - \vec{d}| \tag{2.6}$$

However it is clear that maximum distance between two normalized normals is bounded by top value of 2. Which gives us range $< 0, 2 >$ for all the summands and range $< 0, 8 >$ for the result. It might be convenient to normalize the curvature to the range $< 0, 1 >$ for most applications. Number of neighboring normals is also not limited by four as it is in equation (2.6) but can be extended to get more precise estimation.

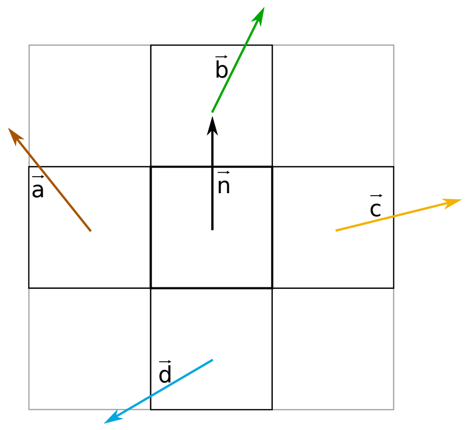


Figure 2.6: Normals $\vec{a}, \vec{b}, \vec{c}, \vec{d}$ as neighbors of normal $\vec{n}$

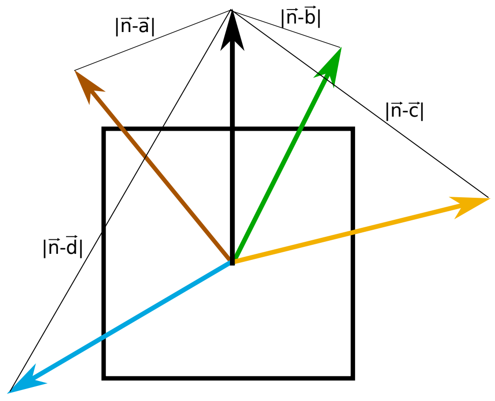


Figure 2.7: The curvature estimation is than computed as the sum of all distances between the $\vec{n}$ and other normals.

2.3.2 Modulation by Distance Between Samples Along the Viewing Ray

We show this principle on the terminology of *ray casting*. Let us have samples s_i, $i \in \{1, ..., n\}$ where n is the count of samples along this ray. Samples are sorted by distance from the camera and index i therefore means the order of sample along the ray (closest sample to the camera has $i = 1$). In the following equation we choose sample s_i to be the current surface and s_j as the context sample.

$$Distance(i) = |s_i.depth - s_j.depth| \tag{2.7}$$

For two samples with indices i, j where $i < j$ and therefore s_i closer to the camera, user defined parameter $focusRegion$ we compute opacity α of sample s_i as:

$$\alpha = saturate\left(\frac{Distance(i)}{focusRegion}\right) \tag{2.8}$$

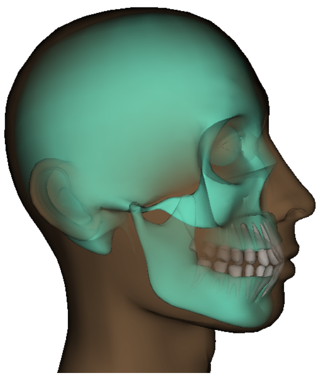


Figure 2.8: α computed using equations (2.8) (2.8)

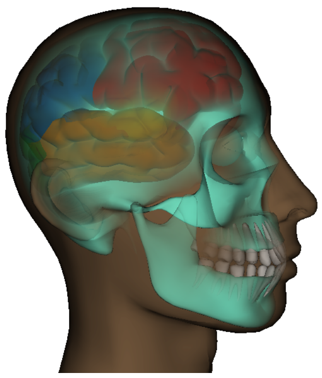


Figure 2.9: α computed using equations (2.9),(2.8)

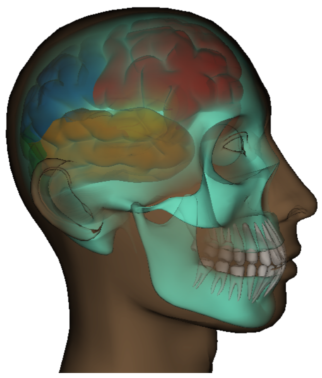


Figure 2.10: α computed using equations (2.9), (2.10)

where *saturate* operation clamps result to range <0,1>. The result of this operation can be seen in figure 2.8. If we store the value *Distance* for every sample along the ray and compute it as:

$$Distance(i) = |s_i.depth - s_{i+1}.depth| \tag{2.9}$$

in each sample stored we then get the distance to the next sample on the ray. This will result in effect that can be seen in figure 2.9. It is very interesting to observe that even-though figures 2.8,2.9 and 2.10 were rendered in our implementation using vertex defined objects, the results seem to be volumetric. This is caused by the human perception of depth in optically active environments.

The problem of this technique is that we do not see samples with high curvatures. If we combine that knowledge with the curvature technique discussed before, we get equation 2.10, which decides what is more important for given sample. User defined parameter δ_c is used to modulate the curvature effect. This effect can be seen in figure 2.10, where the e.g. ear and eye are now clearly visible, since the *curvature* value of those samples is high.

$$\alpha = saturate\left(max\left(\frac{Distance(i)}{focusRegion}\right), curvature(i) \cdot \delta_c\right) \tag{2.10}$$

2.3.3 Modulation by Distance from Important Shape Features in the Layer

Modulation by the distance from important shape features is very useful at places of minimal interest of local shape, and highly appreciated in perception of layers depth, order and crossings. We use the terminology used by Carnecky et al.[12] to define features we work with. For terms c),d),e) we suppose layered surfaces are used as in figure 2.11.

a) **Contour** - is the border of surface silhouette.
b) **Non local transparency** - is a way of modulating the transparency using the distance from important features such as contours. In general this term includes all methods that use the whole context to modulate the transparency, not only a local sample information.
c) **T-Junction** - is an intersection of two perpendicular contours, each of them belonging to one surface in the final image.
d) **XT-Junction** - XT-Junction - is an intersection of two perpendicular contours, each of them belonging to one surface in the final image where some non local transparency method is used and the upper layer therefore shows also the X junction of the bottom layer in addition to the T junction.
e) **Silhouette enhancement** - is a method where the information about the contour is also propagated to the layer below to highlight the silhouette of the upper layer.

We can see the use of *non local transparency* method combined with the *silhouette enhancement* used in figure 2.11. The T-Junction help us to percieve the order of layers in depth. XT-Junctions than reveal the continuing edges of the bottom layer. The highlighted silhouette can be implemented as an opacity modulation, but as discussed later it is much more convenient to use a colour modulation instead to achieve custom colour of the highlight. From simplistic figure 2.11 we can also see that the main goal of non-local transparency is to reveal the most details possible if layers above are not interesting in shape or features.

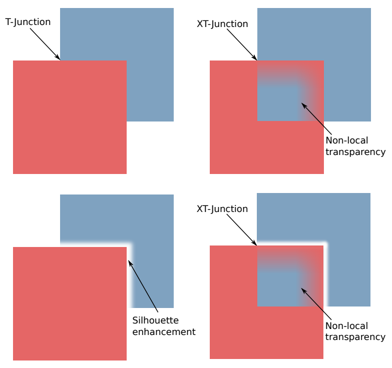


Figure 2.11: T-Junctions

The detection of presented features as well as the algorithm for their diffusion is presented in the next chapter where the use with Illustration Buffer is described.

2.3.4 Modulation by Groups

This is a very basic principle requiring the model to be disassembled to parts, which give functional, spacious, or some user defined meaning. Each part p_k will than have its own opacity α_k to enter the sample compositing equations.

2.3.5 Modulation by Distance from Defined Plane/Area (Cut Motivated)

This method is motivated by cutting through the volumetric data[16].It is, in fact, very similar to the method using distance between specific samples along the ray. In this case however the distance is computed w.r.t. a point, plane, area (called later "the focus element") in general in space which is not the original part of the model. The user is usually in charge of the position and size of such focus element enabling to see just the information in the area of focus.

$$\alpha = saturate\left(\frac{||s_i.pos - f.pos||}{focusRegion}\right) \tag{2.11}$$

Equation 2.11 demonstrates the case of a focus area of size $focusRegion$ and center in point f. Other cases can be solved by point location search methods from *computational geometry*. In case of the focus element defined by plane we search the shortest distance of $s_i.pos$ to the plane (the size of vector perpendicular to the plane originating in $s_i.pos$). The focus element defined by the area can be than solved by determining if $s_i.pos$ is inside or outside of the focus area *convex hull*.

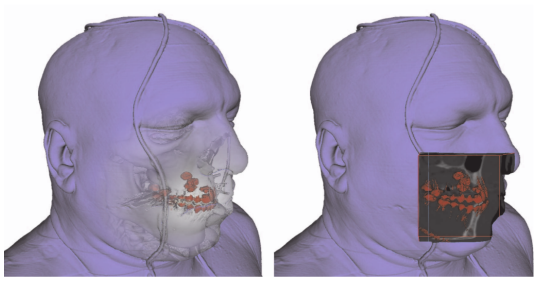


Figure 2.12: Courtesy of Kruger et al.[16] shows distance from the point based modulation on the left and focus area defined by orthogonal box on the right.

2.3.6 Additional Notes to Opacity Modulation

Presented opacity modulation methods are often also called *importance-based*[14],[16]. Even though these methods require different metrics and features to be detected, all of them then affect the samples opacity by the sample importance of given metric.

This chapter shows the most important methods for perception of the internal structure however some other existing and not mentioned methods can be used depending on the use domain.

Great advancement and inspiration in this area can be found in volume rendering where the samples usually contain additional data about the mass they describe [11],[14],[16]. Many of the principles from the illustrative and importance-based volume rendering (which is solvable by ray casting) may be applied to the Illustration Buffer.

2.4 Colour Modulation

Even though the opacity modulation and its application to different methods solving *OIT* problem is the main topic of this work, a colour modulation should not be omitted since it is an important part of the visualization process.

2.4.1 Lighting and Shading

Shading and lighting considerably affects the final perception of what we can see. Even if the opacity modulation would allow us to see everything important we want to see, this might not be enough for the total comprehension of the context. While, in our study, we use only the diffuse lighting as:

$$\begin{aligned} NdotL &= (max(dot(normalize(normal), normalize(eyeDir)), 0.0)) \\ Colour &= sampleColour * NdotL \end{aligned} \tag{2.12}$$

and we do not use any specular part of the lighting, a lighting model can affect our perception of the material and increase the overall understatement of the context. Kruger et al. show results of custom lighting models for the human skin and bones combined with a context preserving opacity modulation[16]. In general we should be very careful using the specular part of the lighting combined with presented opacity modulation methods. The reason of such care is that specular light peak could hide the detail we need in places where it is crucial for the user to understand the object shape and structure.

The selected shading model also affects the perception of the surface smoothness. While in our application the Phong's shading model is used, flat or Gouraud shading might be important to use in specific-use cases[24]. This matter is however not covered in this work.

2.4.2 Silhouette Enhancement

As shown in section (2.3.3),a silhouette enhancement can have great impact when it is combined with methods of non local transparency. This is however best achieved by a colour modulation and not by opacity modulation as in [12]. We therefore use following formula for the silhouette enhancement:

$$Colour = Colour + colourOfHallo * saturate(distance) \tag{2.13}$$

where $colourOfHallo$ is the desired colour of the highlight multiplied by $distance$ to the contour of the silhouette saturated to $< 0, 1 >$. This is further discussed in 3.3.1. Please visit algorithm 9 in the next chapter to see how this can be included in the colour composition process.

2.4.3 Transfer Functions

Transfer functions are a powerful tool especially in volume rendering where each sample is defined by additional data, e.g. density. The transfer function is defined as:

Function $f(x)$ is a transfer function $f(x) = \tau_d$ if x is a scalar value and τ is a value of dimension d. In visualization the τ_4 is usually used to map the scalar value to RGBA colour.

This definition can be extended to multiple dimensions of the mapped scalar value as well to achieve $f(x_0, ..., x_n) = \tau_d$ where multiple scalar values will affect the result. Even though our work does not focus on the volumetric rendering, this method can be used to geometry based models as well. For example we can use it to modulate an opacity in a custom way based on the distance. We use transfer function for debugging purposes and visualization of the Illustration Buffer as shown in Chapter 4.

Chapter 3

Analysis of Compared Methods

This chapter presents analysis of the OIT problem and OIT solving algorithms. Further, comparison of presented algorithms is discussed w.r.t. ease of use and application of opacity modulation methods presented in Chapter 2.

3.1 Order Independent Transparency

Problem of the Order Independent Transparency (OIT) is well known in scene rendering and there is no standard – already included implementation in either OpenGL or DirectX. It is a general problem consisting of rendering objects with a uniform or non-uniform alpha channel. To display such geometry correctly all fragments need to be blended in the correct order, thus sorting the fragments is often the key requirement for techniques solving OIT. This section is a summary of several most common techniques to deal with this problem.

3.1.1 Depth Peeling

In 2001 presented method by C.Everitt[15] is based on multiple geometry passes, peeling just one layer of visible geometry per pass. It is in fact based on a shadow mapping technique, which helps to determine visibility between scene points and a certain light source. Shadow mapping uses additional depth buffers for every light source to be able to compare the depth of rendered fragment and the depth buffer created by the light source.

While standard depth test provided by GPUs would give us only the nearest fragment's depth, it is not a sufficient test for OIT problem since knowledge of all layers depths is required. The algorithm works as follows:

Algorithm 3: The Depth Peeling on GPU

```
Data: Depth Buffer D, Shadow Texture S, Layer Texture L, Result Texture R, Geometry
Result: All layers blended into R
1  Depth Test ON
2  Render the geometry
3  Store the first layer consisting of the nearest fragments to R, store the depth of such fragments to S.
4  while Layers to be processed exist do
5      Depth Test ON
6      Render the geometry
7      Peel the previously captured layer using S.
8      Store result in L
9      S = create shadow map from the camera view.
10     Depth Test OFF
11     Render full-screen quad
12     Blend the L to R
13 end
14 In the final pass render texture R to the framebuffer.
```

Since we need two depth tests per pass, first to peel away the previous layer and the second to render only the currently nearest fragments, OpenGL depth buffer is not enough. For the second depth test we use standard OpenGL depth buffer, shadow-mapping[24] principle is used to create the depth map of layer to be peeled. This shadow map is however rendered from the point of the camera and not the light, see figure 3.1. By storing only z coordinates instead of the colour we get the depth map for the next layer to be peeled.

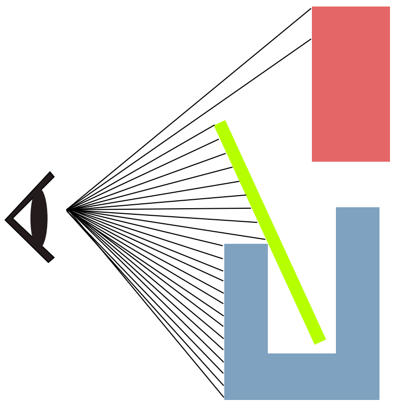


Figure 3.1: Creating *shadow map* from the camera point of view to retrieve the depth of nearest fragments.

This algorithm therefore performs n geometry passes to retrieve n layers. While more advanced algorithms such as Dual Depth Peeling[9] blend these layers "on the fly" during the peeling passes, depth peeling algorithm[15] stores currently retrieved layer and performs another blending pass using full-screen quad, using OpenGL blending functions.

With graphical hardware being more advanced several extensions of original *Depth Peeling* algorithm are published, such as *Z-fighting Aware Depth Peeling*[22] and more importantly Dual Depth Peeling[9] discussed below this section.

3.1.2 Dual Depth Peeling

Dual depth peeling method[9]is a modification of the original Depth peeling algorithm discussed in previous section allowing to peel two layers at once. In one pass it peels back and front layers simultaneously. Since this is not possible to do with default depth buffer and GPU does not have multiple depth buffers to perform front to back and back to front rendering, custom min-max depth buffer has to be used.

To prevent peeling any fragments by both *front to back* and *back to front* directions, the algorithm uses mechanism of sliding window for two consecutive layers. The *min-max depth buffer* is implemented as RG32F two channel texture where depth values can be compared by MIN blending in every pass. By using the sliding window mechanism single fragment shader can be used for the depth comparison and blending. Default depth buffer offered by hardware is turned off. The mechanism of using a sliding window for two consecutive layers works as follows:

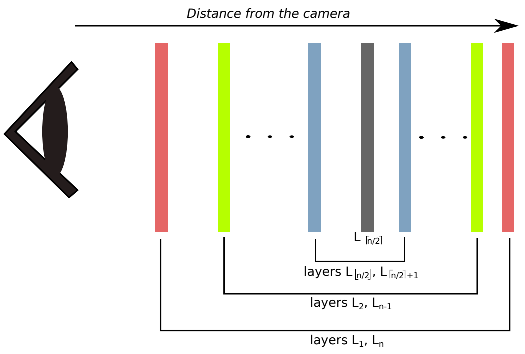


Figure 3.2: Peeling layers in both front and back direction. If number of layers n is odd, the last layer would be blended twice if the sliding window mechanism was not used.

Algorithm 4: The Dual Depth Peeling on GPU

```
   Data: Min-Max buffer, Layer Texture F,Layer Texture B, Geometry
   Result: All layers blended to one
 1 Depth Test ON
 2 Render the geometry
 3 Store the first layer consisting of the nearest fragments to F and the last layer of furthest fragments
   to B.
 4 In the first pass no fragments are peeled. The min-max buffer values are initialized by depths of two
   outside layers. while both directions did not meet do
 5     Render the geometry
 6     All fragments with equal depths from the previous pass are peeled from the front and back.
 7     Update the min-max buffer.
 8     Blend results of front peeling with F and results of back peeling with B in the same pass.
 9 end
10 To process the last fragments (if exists) only one direction needs to be used to avoid
   double-processed fragments, using the sliding window.
11 The additional final step is needed to blend both layers F,B created by front to back and back to
   front progress.
```

It is also important to state that for the front to back direction under-blending equation needs to be used while for all fragments peeled in the opposite direction, over-blending equation is needed. Both equations are explained further in Chapter 3.

While in original depth peeling N geometry passes were necessary to process the scene, where N is the number of layers it created, Dual depth peeling performs $N/2 + 1$ geometry passes only, where the additional $+1$ step is described as step 11 in algorithm description.

3.1.3 Alpha Blending Approximations

As published by Houman Meshkin[17] at GDC[1] it is possible to approximate rendering of transparent objects in a single geometry pass. The main idea of the Weighted Sum algorithm is to remove completely order-dependent terms of the rendering equation and splitting order-independent terms to multiple render targets. This method however does not produce plausible results for higher alpha values, only for alpha values $\alpha < 0.3$. Higher alpha values result in too dark or too bright images (since we do not compare this method to other algorithms in greater detail, please head to cited paper for details and images).

Much better results were achieved with *Average Sum* algorithm published by NVIDIA[9]. Their idea is based on the fact, that if all the layers had the same colour, result would not depend on layers' order. To fulfill such condition colours are replaced by average colour per pixel and weighted by fragments' alpha if not uniform. Even though the algorithm is called[9] *Single-Pass Approximation*, at least two passes are needed to accumulate RGBA colours first and then to compute alpha-weighted average colour. As this algorithm does not omit any order-dependent terms of the rendering equation, results are much more plausible even for high or non-uniform alphas compared to Dual depth peeling and Weighted sum algorithms. Such comparison of images can be found in[9].

Even though this approach is very fast, it gives only the approximation of the OIT problem which is not extendable to any methods considering non-local transparency and therefore this method is not further examined in this text.

3.1.4 Concurrent Linked List Construction on the GPU

So far described methods are complex and unintuitive due to graphical hardware of the time they were published. A much more intuitive method is to store every fragment that belongs to one pixel to a linked list originating in that pixel and sorting it by fragment's depth to achieve correct behavior in the colours compositing equation. Method[23] described below is very similar to original *A-buffer* published in 1984[13], it only achieves OIT by using linked lists constructed in memory of GPU. While first GPU implementations of A-buffer presented by Meyers and Bavoil called *Stencil routed A-buffer*[18] and Bavoil et al.[10] called *K-buffer* was able to store fixed amount of fragments per list, method presented by Yang[23] is unbounded. Since this work is based on creating linked lists on GPU, this topic is discussed further in following chapters.

A GPU version of A-buffer can be constructed in two geometrical passes. In the first pass we create a linked lists of fragments per pixel. To do that we need to store the fragments along with the pointers to their neighbours along the viewing ray, as well as the pointer to the first fragment per pixel, called a *pixel head*. First fragments that are processed per pixel are stored as la inked list head. With the following fragments we continue to build the linked lists per pixel until all fragments are processed.

[1]Game Development Conference

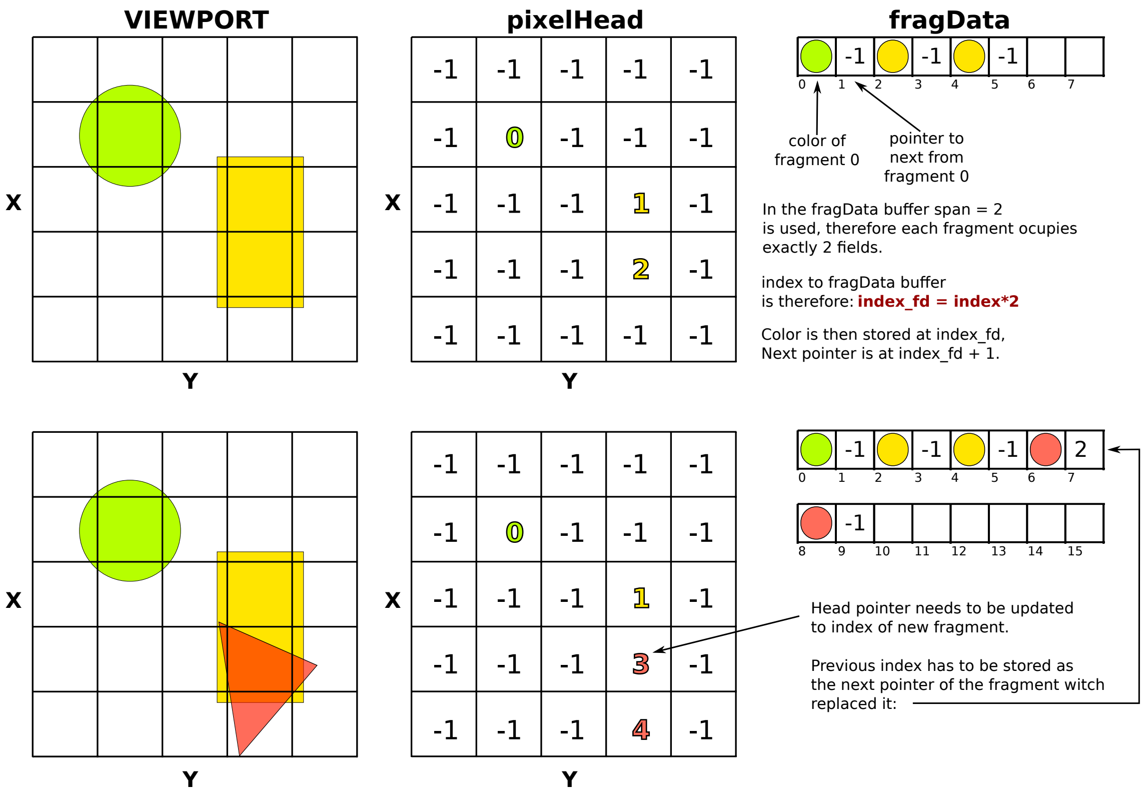

VIEWPORT
pixelHead
fragData
X
Y
color of fragment 0
pointer to next from fragment 0
In the fragData buffer span = 2 is used, therefore each fragment ocupies exactly 2 fields.
index to fragData buffer is therefore: index_fd = index*2
Color is then stored at index_fd, Next pointer is at index_fd + 1.
Head pointer needs to be updated to index of new fragment.
Previous index has to be stored as the next pointer of the fragment witch replaced it:

Figure 3.3: Shows how fragments are stored to the linked lists structure whene elements are rendered to the viewport.

Algorithm 5: creating the concurrent linked lists structure

```
Data: Geometry to be rendered, atomic counter $AC = 1$
Result: Unsorted concurrent linked lists
Depth Test OFF
Render the geometry
while fragment of coordinates $(X, Y)$ in view space to be processed exists do
    index = $AC$
    compute fragment colour according to selected shading and lighting model
    if $pixelHead(X, Y) == -1$ then
        pixelHead(X,Y) = index
        fragData(index*2) = colour
        fragData(index*2 + 1) = -1// since we are first fragment of this lists next pointer is empty.
    else
        nextPointIndex = pixelHead(X,Y)
        pixelHead(X,Y) = index
        fragData(index*2) = colour
        fragData(index*2 + 1) = nextPointIndex
    end
    $AC+ = 1$ // increase the atomic counter
    discard fragment // we do not want it to be seen yet.
end
```

As soon as we have the A-buffer constructed in the GPU's memory, the second pass is needed to sort pixels' linked lists. That can be achieved simply by rendering a full-screen quad and sorting lists in the fragment shader. While the original A-buffer[13] tored depth, transparency, pixel coverage and colour per fragment, it's GPU version23 stores such data to separate buffer and than uses pointers to speed up the work with the linked list structure.

Figure 3.3 shows how to store the fragments data to the memory of the GPU to form concurrent linked lists. To do so we need an atomic counter that will increase with each new rendered fragment to give us the fragment identifier. The algorithm of creating the linked lists is then showed in the Algorithm 5.

Now we have all samples (fragments) along each ray going through the center of each pixel stored in the concurrent linked lists. To be able to compose the final image we have to now sort the samples of each list. To do so the full-screen quad is rendered and for each pixel the HEADPOINTERS buffer is queried for the first fragment of the list. Next pointers in the FRAGMENTDATA buffer are then used to traverse the list until special value of -1 is found which indicates the last element in the list. Fragments are than sorted using insertion sort as Yang[23] or selection sort as Carnecky[12] (Even though selection sort should be more efficient due to number of writes over the insertion sort, no performance difference was found presumably because of caching[12]).

The biggest difference of this approach to the *peeling* mechanisms is that we only need to render the geometry once. When sorting and blending the collected samples along the rays we only access all the information already stored in GPU memory. Used structures are mentioned in greater detail in the Design and Implementation chapter

3.1.5 Illustration Buffer

Inspired by Yang[23], technique called *Illustration Buffer* was presented by Carnecky et al.[12]. It is the context that is desirable knowledge for all fragments to have. With that knowledge it is easier to determine the fragment role in the image, such as being a part of the silhouette, being occluded etc. Illustration buffer data structure is motivated by several image enhancements that modulate opacity based on specific image features that increase understanding of complex transparent objects notably. Such features are described in the Theory section in further detail.

To provide the information about the surrounding shape of all fragments, A-buffer constructed in the GPU memory[23] is extended. While in A-buffer method fragments know their neighbors only along the viewing ray, Illustration Buffer presents method of finding and connecting also neighbors that belong to the surrounding pixels. For pixel with coordinates (x, y) new four neighbors are found if exists in linked lists of pixels $(s + dx, y + dy)$ where $(dx, dy) \in \{(1, 0), (-1, 0), (0, 1), (0, -1)\}$.

After the neighbours being found the Illustration buffer can be used to traverse object surfaces to retrieve desired shape describing information such as gradients or distances to important features.

Figure 3.4 shows necessary structures for the *Illustration Buffer* and how the data are stored when new element is rendered. To retrieve the next available free index for inserted fragment we need to use global atomic counter, or several as is discussed in the implementation. In contrast to the previous *concurrent linked lists* we need to store much more information. Therefore we reserve several cells in the $fragData$ buffer per fragment and use the spanning mechanism for retrieving the index to this buffer as explained in figure 3.3. This is the list of structures we will need to construct and work with the Illustration Buffer:

- **pixelHead** of size $X \times Y$ where X, Y are the dimensions of the viewport. It stores ID of the first fragment in the list.
- **pixelCount** is also two-dimensional $X \times Y$ buffer storing lengths of the lists in each pixel. This buffer is however optional since we can control the end of the list by special value of the last next pointer. On the other hand we can use this information in the opacity modulation as shown further.
- **fragNext** is a one dimensional buffer where at frag_next(index) is stored id of the next fragment of id=index.
- **fragData** stores all data we need to work with the buffer. It stores for each fragment the colour, indices of its four geodesic neighbors and optionally other data we need for non local transparency which will be introduced further.
- **fragData2** is of the same layout as $fragData$ and it is used for ping pong computational schemes.

3.1.5.1 Neighbors Location by Carnecky

Now we assume we have already created the concurrent linked lists by Algorithm 5 and that samples are already sorted along the viewing ray. This will be essential to location of the neighbors.

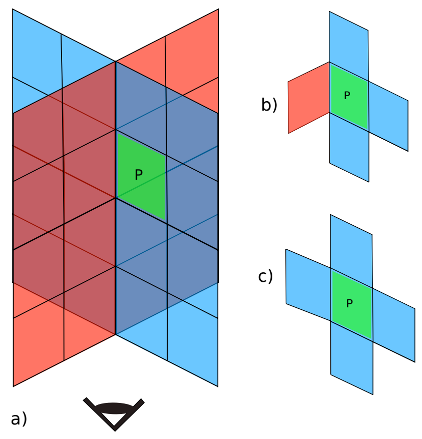


Figure 3.5: Using a perspective we show two crossing planes **a)**. When peeling the first layer and querying the neighboring pixels of P in such layer, we retrieve **b)**. When searching for neighbors of P in Illustrative buffer we want to find **c)**.

These neighbors however differ greatly from the neighbors in the layers. Consider situation depicted in figure 3.5 where we see found neighbors and peeled layers. It shows how greatly can differ terms *neighbors of P in peeled layer* and *neighbors of P on the surface*. This major difference will show vital for future use as shown later in this Chapter.

As shown in figure 3.5 the goal is to find geodesic neighbors on the same surface and not of the same layer in the peeling point of view. We denote fragments of current linked list L as $l_i, i \in \{1, ..., n\}$ where n is the number of fragments in L and neighboring list K with fragments $k_j, j \in \{1, ..., m\}$ where m is the number of fragments in K. While in peeled layers neighboring pixels will be both in *i-th* layer, for indices of geodesic neighbors

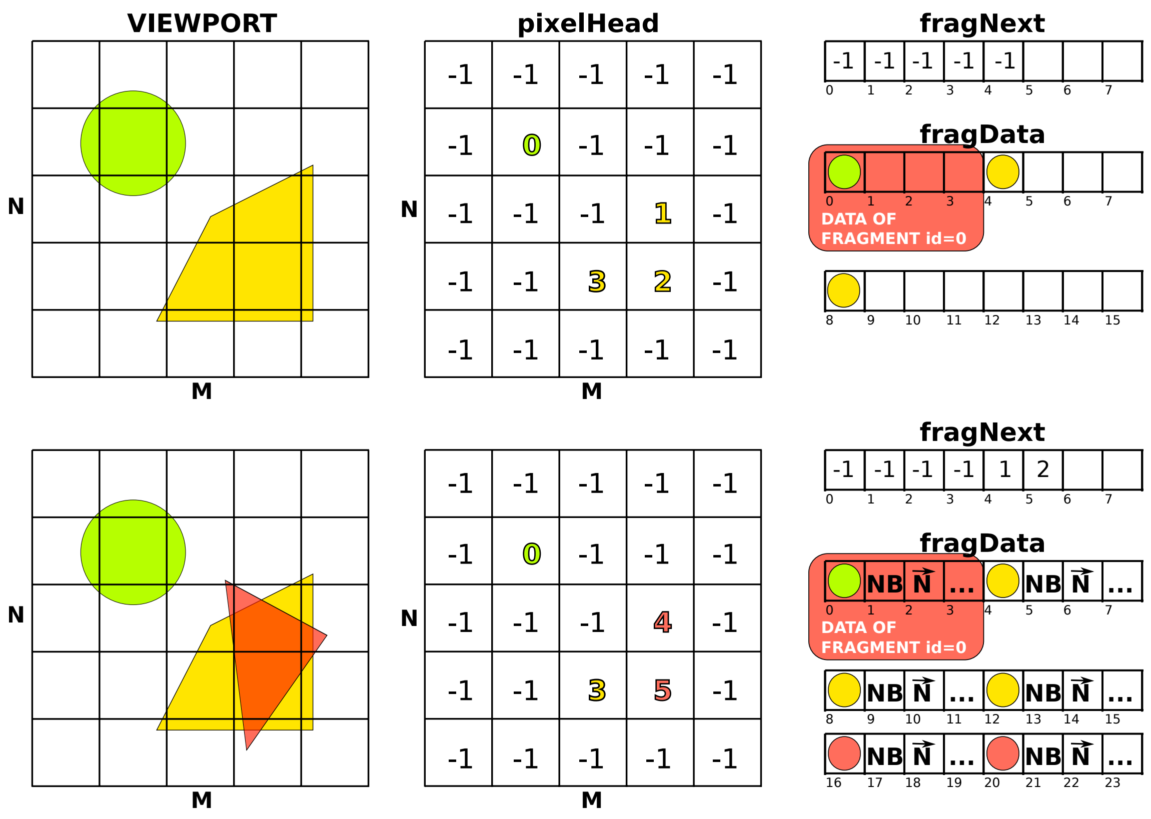


Figure 3.4: Principle of the concurrent lists is the same as in figure 3.3. Here however are next pointers stored separately from the data. Data are spanned by size of four here, giving space for colour, four surrounding neighbours (NB), normal (N) and some other data, which depend on the target use of the buffer. Buffer pixelHead if of a same size as the viewport ($X \times Y$). This figure does not consider the sorting of samples along the ray.

does not always apply that $i == j$. That means that to find the neighbor we need to traverse entire neighboring list. Carnecky et al. use a simple heuristic measure ϵ of the surface continuity for two fragments as shown in figure 3.6. They compute the ϵ_n as difference of fragments i, j normals n_i, n_j as:

$$\epsilon_n(i,j) = 1 - n_i + n_j$$

The eye distance ϵ_z is computed using the radius of a rendered object bounding spherer_{obj}, normal n_i, pixel coordinates x_i, eye distance z coordinate and finally the z_i gradient $\left(\frac{z_i}{dx_i}\right)$:

$$\epsilon_z(i,j) = \frac{1}{r_{obj}}\left[z_i + (x_j - x_i)\cdot\left(\frac{dz_i}{dx_i}\right) - z_j\right]$$

$$\epsilon(i,j) = w_z \cdot \epsilon_z(i,j) + w_n \cdot \epsilon_n(i,j)$$

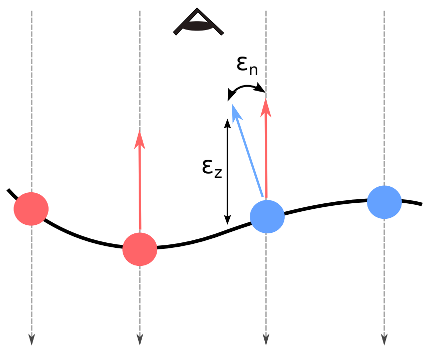


Figure 3.6: Meaning of ϵ_n and ϵ_z on the surface samples.

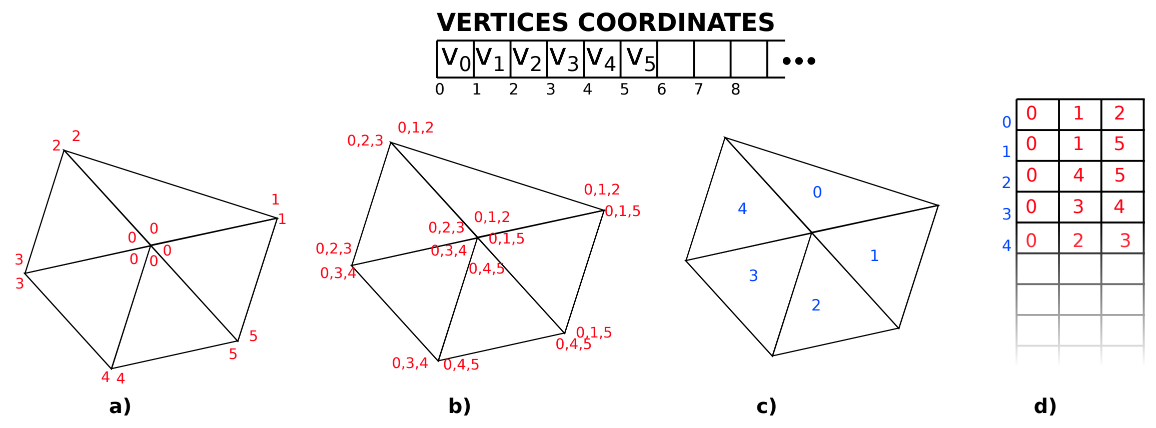


Figure 3.7: **a)** indexed geometry to prevent duplicate geometry to be sent to gpu. **b)** Every vertex knows indices of all vertices in given triangle. **c)** Every triangle has its own id. **d)** table maps IDs of the triangles (blue) to b) where every vertex knows all indices in its triangle.

Given this heuristic measure ϵ will be small for probably neighboring fragments of the same surface and large for fragments of different surfaces. For two neighboring lists A, B and fragment $f_i \in A$ they first try to find the best neighbor candidate c_i for f_i in B and than they traverse A to find if there is better neighbor for c_i then f_i in A. Even though they use component ID check to set $\epsilon = \infty$ to exclude fragments of different components (and therefore surfaces), this is rather inefficient since for location of one neighbor we have to traverse both neighboring and original lists.

3.1.5.2 Proposed Neighbors location

To overcome the inefficiency of method[12] presented by Carnecky et al. we propose new method motivated by indexed geometry. Given two neighboring lists A, B where list A is the current list and B is the list where neighbor is to be find, we propose auxiliary structure depicted in figure 3.7.

We are using indexed geometry to lower the load of informations mapped to GPU memory. We however extend the indices information so that every vertex knows indices of all vertices of the same triangle. This is shown in figure 3.7 **b)**. This would however be against the very principle of indexed geometry since we would replicate a lot of data. This can be solved as shown in figure 3.7 **c)** where every triangle has its unique ID attached and auxiliary table to map IDs to triangle indices as shown in figure 3.7 **d)**.

For fragments $f \in A$ of coordinates x_f, y_f and $g \in B$ of coordinates x_g, y_g then apply following rules:

1. f and g are not neighbors if f and g do not share any indices of the triangle they are part of.

2. f and g are neighbors and fragments of the same triangle if f and g share exactly 3 indices.

3. f and g are neighbors and fragments of two neighboring triangles if f and g share exactly 2 indices.

4. f and g are neighbors and fragments of two neighboring triangles if f and g share exactly 1 indices. This situation can happen e.g. for triangles with $ID = 1, ID = 4$ in figure 3.7.

Algorithm 6: proposed neighbor search

```
Data: two neighboring lists $A, B$, current fragment $f_i \in A$, indices of $f_i$ $indicesOfF$.
Result: Index to the linked list structure of $f_i$ neighbor.
for $b$ = 0; $b$ < count(B); $b$++ do
    fragB = B(b); indicesOfB = fragB.triangleIndices;
    for $k$ = 0; $k$ < 3; $k$++ do
        for $l$ = 0; $l$ < 3; $l$++ do
            if indicesOfB[k] == indicesOfA[L] then
                return fragB.ID; // Neighbor has been found since it shares at least one triangle
                index with $f_i$.
            end
        end
    end
end
```

The algorithm for neighbor search is then simplified to only one cycle through the neighboring list B and there is no need for the cycle through A afterwards.

drawbacks

Even-though this method is geometry motivated there can be artifacts caused by the rasterization process. Such artifacts occur when rendered triangles are smaller than pixel and neighboring fragments skip triangle(s) in the *isNeighbor* query. This error is shown in figure 3.9. With that knowledge we can higher the viewport resolution or lower the detail of the model to overcome this drawback in exchange for speed.

3.2 Comparison of OIT Solving Methods

We have shown several algorithms to solve the OIT problem as well methods to modulate opacity based on several shape properties. Not all of those methods can be applied to all presented algorithms solving OIT with the same effort or cannot be applied at all. This section summaries such comparison of the peeling methods, per pixel linked lists and the Illustration Buffer.

3.2.1 Modulation by Groups

Opacity modulation based on groups is trivial in all presented methods solving the OIT and therefore is presented separately. To all algorithms we simply pass the *group id* along with colour and normal to all fragments. Than we present the lookup texture with definitions of desired opacity for each group.

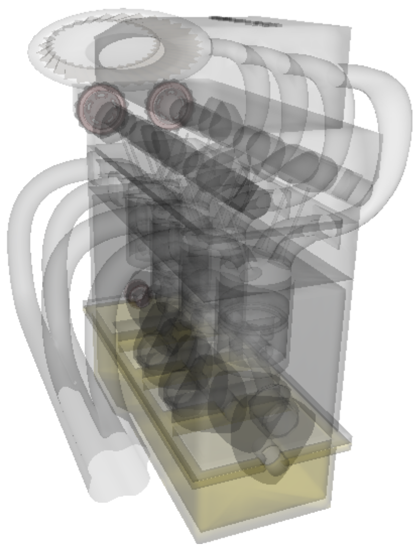


Figure 3.8: Every fragment has equal opacity.

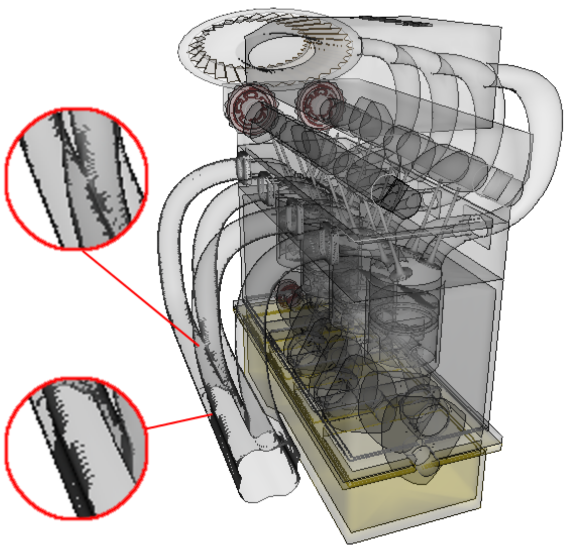


Figure 3.9: Fragment is fully opaque if the number of neighbors is less than four meaning it is part of the edge.

3.2.2 Modulation by Distance from Defined Plane/Area (Cut Motivated)

Modulation of opacity by distance from defined area or plane is rather easy since the principle is easily applicable to all mentioned algorithms it is presented separately as well.

In all mentioned algorithms we are able to pass to each fragment its position in camera space to prevent non linear coordinates scale given by the projection to view space. With this knowledge we can compute the distance to a *focus area* independently on the chosen OIT solving algorithm.

3.2.3 Depth Peeling and Dual Depth Peeling

The depth peeling mechanism is really straightforward to implement as well as its extension the dual depth peeling. It is not however as modifiable as we would want to as described in following text considering the opacity modulation.

Modulation by Differences in Shape

Please recall the figures 2.6 and 2.7 as well as the *curvature* computation first. For this problem to solve it is also important to understand the difference between neighbors in the peeled layer and neighbors of the same surface. This difference is vital for understanding following problem.

Given the situation in figure 3.10 a) we first need to find the neighbors in current layer. However since the first layer will consist of both terrain and the grid, we can't simply return surrounding pixels of the texture storing such layer. To solve this partially we can pass

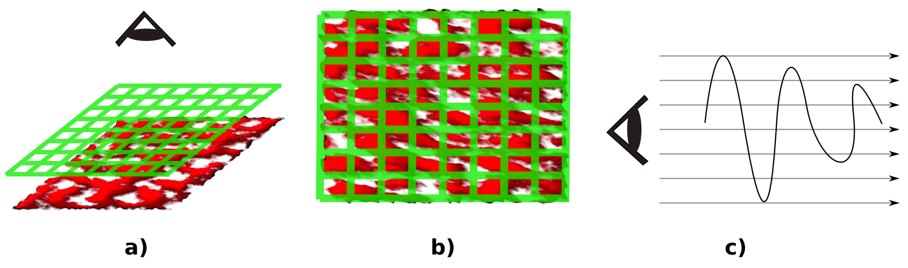


Figure 3.10: **a)** space arrangement of shaded 3D terrain (red) with green grid on the top. **b)** Projection of objects in a). **c)** points of the same surface in object's perspective do not have to be neighboring fragments.

the *componentID* along with the colour and normal to the shader. This will solve the cases where all the neighboring pixels have the same component ID. In case of different component ID we can't fetch the true neighbor on the surface since we have no way how to traverse all the layers. We can try to guess the curvature based on the other neighbors but we do not know if the surface really continues in that direction. This approach will therefore produce artifacts along the edges if curvature is used.

We could modify the algorithm to render to different texture for every peeling pass but this would not help us in cases depicted in figure 3.10 **c)** since we would still need to choose between several fragments with the same *componentID*. This can be solved by applying some eye-depth based heuristic with use of dynamic number of allocated textures given by number of layers. This approach is not really intuitive to implement and would require major change in the algorithm - storing every layer separately.

Modulation by Distance Between Samples along the Viewing Ray

While in all other presented cases the modification of the algorithm is the same for the original depth peeling and dual depth peeling, we need to use the original depth peeling to achieve the modulation by distance between layer samples.

When peeling the layers in the original depth peeling we do it in front to back manner. The distance from the last layer is computed as:

$$Distance(i) = |s_i.depth - s_j.depth|$$

where s_i is the current sample and s_j is the sample of previously peeled layer. We do not even store that information explicitly along with the colour since this information is stored in the depth buffer that we used to peel away previously processed layers.

Even though implementation of this is very simple and straightforward there is a problem of normalization the *Distance* value. To do that we would need the maximum distance of the whole peeling process but since depth peeling algorithms blend the intermediate results layer by layer we do not have that information until the algorithm finishes which is too late. The simplest solution of this problem might be user defined and controlled value which would have to be adjusted for each model separately.

The dual depth peeling algorithm would have to be modified to not to blend the layers on the fly in the same pass as the peeling occurs but to use one more texture to store the result. If layers are indexed in front to back manner, results of layer l_i would have to blended to previously processed layers in the peeling of the next layer l_{i-1}, when *Distance* between l_i and l_{i-1} is known.

Modulation by Distance from Important Shape Features in the Layer

This is the most problematic case of the peeling methods functionality extension. As we have seen in case of modulation by differences in shape finding the correct neighbors of the same surface is difficult and non-intuitive task requiring major changes of the original algorithm. Reason for that is that for successful diffusion of the distance information we do need to know the real neighbors on the surface, not the neighbors in the peeled layer.

Second problem is to successfully detect desired image features, the silhouette in this case. This problem was addressed by Nienhaus and Döllner in search for a method to find object blueprints using depth peeling[20]. They have combined the depth normal with normal buffer to create the *edge map* using discontinuities in the depth and normals to find the visible edges. The non-visible edges become visible after occluding geometry is peeled using the depth peeling mechanism.

In case of the opacity modulation by the distance from the closest silhouette of the same surface, we need to spread the distance information over the layer first. Considering the problem with location of the true neighbors on the surface, this problem becomes inherently difficult. If we for now think of the neighbors location issue as solved we could spread the distance information using method presented by Rong and Tan called the *jump flooding*[21]. Jump flooding allows to spread the information from the original seeds (the silhouette in our case) in $logn$ steps given the $n \times n$ grid size.

As we will see in the analysis of the Illustration Buffer, this kind of modulation is much more easier to implement and comprehend than in case of peeling algorithms. Therefore we do not advise using the depth peeling in this case.

We have also discussed the *silhouette enhancement* technique, which requires knowledge of the distance from the silhouette of occluding layer above the current layer (please see the Chapter 2 to recall the silhouette enhancement). Trying to implement such feature with depth peeling is also not advised for the same reasons.

3.3 Per Pixel Linked Lists

Per pixel linked lists are fast and intuitive method to solve the OIT. It is however clear that without extending the algorithm to the *Illustration Buffer* it is not applicable for any kind of modulation that requires the fragment neighbors to modulate the opacity.

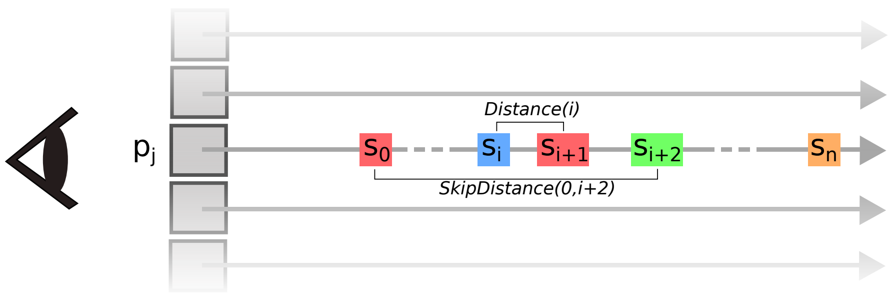


Figure 3.11: Samples along the view ray and distances between them in camera space. *Distance(i)* shows a distance to the next layer in front to back manner and *SkipDistance(i,k)* is a distance between samples s_i, s_k.

Modulation by Differences in Shape

This is the first case which per pixel linked lists are to simple to cover. Since we have no information about neighboring fragments of the same surface, we cannot differentiate surrounding normals to compute the curvature value.

However the case might not be lost entirely. We still could use the *illumination-based importance* to modulate opacity by the shape differences[14]. Even we will not get identical results, we will be able to see the shape differences thanks to its shading computed with respect to the eye vector. Places of highest lighting intensities will have the highest opacity.

For an extension of per pixel linked lists capable of this kind of modulation properly please refer to the next section where Illustration Buffer is discussed.

Modulation by Distance Between Samples along the Viewing Ray

Great advantage of the linked lists structure is that we can traverse the list anytime we need to after it is created. This does not apply for the depth peeling methods. This fact becomes important when distance between layers that are not successive.

$$Distance(i) = |s_i.depth - s_{i-1}.depth|$$

$$SkipDistance(i, j) = |s_i.depth - s_j.depth|$$

We can compute the *Distance(i)* value in the sorting pass since we already have loaded pointers to memory to each sample along the ray and it will be more effective due to cashing. Samples need to be sorted first of course. However in general case represented by *SkipDistance(i,j)* where for s_i, s_j applies that $|i - j| > 1$ this couldn't be solved by original depth peeling nor the dual depth peeling. In case of the linked list we need just one traverse of the list to get distances originating in s_i to all samples s_j where $j > i$.

Modulation by Distance from Important Shape Features in the Layer

This feature cannot be implemented using only concurrent linked lists. Please refer to the *Illustration Buffer* which enables traversing not only the list along the view ray but also to the neighboring fragments of the same surface as depicted in figure 3.12.

3.3.1 The Illustration Buffer

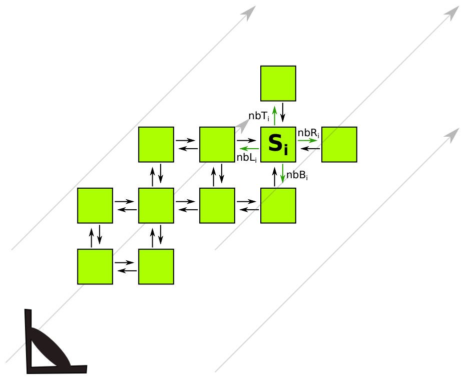


Figure 3.12: Every fragment s_i has four pointers $nbT_i, nbR_i, nbB_i, nbL_i$ to its geodesic neighbor in the top, right, bottom and left direction, if such neighbor exists. Fragment s_i still has the pointer to next fragment along the view ray, it is not visualized here for the sake of simplicity.

In solving OIT problem with opacity modulation based on non local transparency methods we find the *Illustration Buffer* to be the best of presented methods. This section explains how we can use the Illustration Buffer to explore what happens not only in properties of the current fragment but also in properties of its surroundings along the surface.

Modulation by Differences in Shape

Given the constructed *Illustration Buffer* it is easy to achieve this kind of modulation. All we have to do is to fetch neighbors indices of the current fragment and fetch their normals from the *fragData* buffer as well, as shown in algorithm 7.

Modulation by Distance Between Samples along the Viewing Ray

Since the Illustration Buffer is extension of the per pixel linked lists algorithm which already solved this issue please refer to previous section describing the mechanism inherited by Illustration Buffer.

Algorithm 7: Computing curvature from geodesic neighbors normals of fragment f_i

```
Data: current fragment $f_i$ and its normal $n$
Result: $Curvature = |\vec{n} - \vec{a}| + |\vec{n} - \vec{b}| + |\vec{n} - \vec{c}| + |\vec{n} - \vec{d}|$
vec4 neighbors = getNeighbors(fragData, i);
float Curvature = 0;
for int j = 0; j < 4; j++ do
    if exists(neighbors[j]) then
        neighborData = getData(fragData, neighbors[j]);
        vec3 neighborNormal = getNormal(neighborData);
        Curvature += $|\vec{n} - \vec{neighborNormal}|$;
    end
end
return saturate(Curvature); //Please refer to the Theoretical Background Chapter for
description of the saturate function.
```

Modulation by Distance from Important Shape Features in the Layer

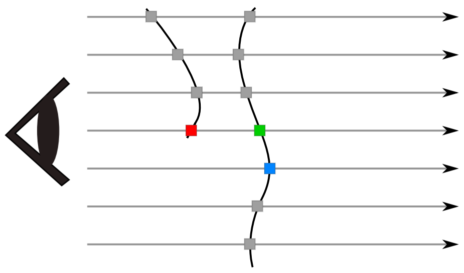


Figure 3.13: Fragment of the silhouette (red), fragment occluded by silhouette (green) and its neighboring fragment not occluded by above surface (blue).

Method described by Carnecky to modulate the opacity by the distance from the surface silhouettes[12] uses the Illustration Buffer structure. Transparency field β is introduced to have high values at surface silhouettes falling off with increasing distance from the silhouette. Binary value b_β is then used to identify the boundary fragments and is therefore set to 0 for the boundary fragments and 1 elsewhere. As initial conditions we set $\beta = 1$ and b_β for fragments with less than four neighbors (red fragment in figure 3.13). If fragment is occluded by a silhouette we set $\beta = 0$ and $b_\beta = 0$ and we set $\beta = 0, b_\beta = 1$ for all other pixels. Those values are stored as custom fragment properties in *fragData* buffer. To spread the initial values of β and b_β two possibilities are presented[12] - physical and non-physical approximation:

$$\frac{\partial}{\partial t}\beta = \lambda_\beta \Delta\beta \tag{3.1}$$

First is physically motivated process of homogeneous diffusion defined as (3.1) with a diffusion coefficient λ_α. Local operator (3.2) using a forward discretization of (3.1) is used to implement the diffusion process. The steady state of the diffusion is not met, forward discretization process is stopped after specified number of iterations.

$$\beta^{k+1} = \beta^k + b_\beta \lambda_\beta \Delta \beta^k \tag{3.2}$$

In equation (3.2) second order central finite difference approximation of Δ_β is used. Since without computing the error estimation the central finite difference produces errors that are not visually plausible we use discrete Laplace operator for $\Delta\alpha^k$ estimation from 8 surrounding neighbors. Please refer to the *Design and Implementation* Chapter for details.

This process is however not very effective for the diffusion of α modulated by distance from the silhouettes. Please see the *Results and Discussion* Chapter for comparison with following non-physical process:

$$\beta^{k+1} = max(\beta^k, max(\beta^k_{neighbor}) - \lambda_\beta) \tag{3.3}$$

We have been also examining the possibility of using already mentioned *Jump flooding* algorithm which would for $n \times n$ grid size distribute the distance in $logn$ steps[21]. This idea is unfortunately not applicable since we cannot simply solve the query: 'is f_i of screen coordinates k,l neighbor of f_j of screen coordinates m,n?', which is vital query asked in every step of the jump flooding algorithm. Situation is shown in figure 3.14. Problem is that we would have to employ some pathfinding[2] technique to answer that question which would be much more computationally expensive than presented diffusion processes.

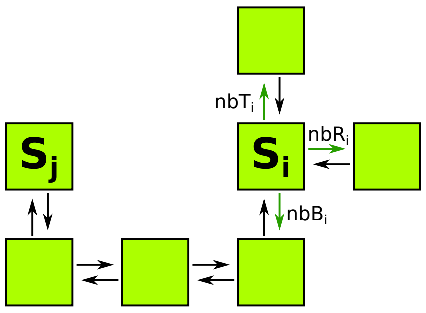


Figure 3.14: Can we simply answer the question 'is s_i and s_j on the same surface?' Answer is unfortunately negative as explained in the text.

The *silhouette enhancement* can be implemented in very same manner as the β field. The field γ (called the halo highlight field[12]) is therefore introduced along with the binary value b_γ. Value of γ will be high near the occluding edges and fall of with increasing distance from such edge. $\gamma = 0, b_\gamma = 1$ for every other fragment. We can use the very same diffusion as with the β field:

$$\gamma^{k+1} = max(\gamma^k, max(\gamma^k_{neighbor}) - \lambda_\gamma) \tag{3.4}$$

User defined variables λ_β and λ_γ in equations 3.3,3.4 affect the falloff of the β, γ values and results of such affecting are discussed in Chapter 5.

To use the values we have spread over the surfaces we need to modify the image composition equations. Considering the front to back blending procedure, colour c as the final pixel colour, colour c_i of current fragment and α_i is fragment initial transparency (not opacity), the composition algorithm is modified to Algorithm 8.

[2]Pathfinding is a method of finding a path from A to B in a graph.

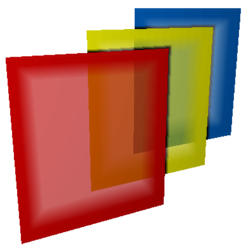


Figure 3.15: Hallo highlight using algorithm 8 produces dark hallo.

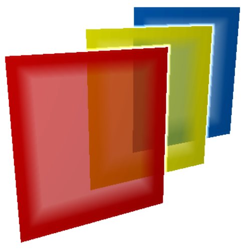


Figure 3.16: Hallo highlight produced by algorithm 9 with *colourOfHaloHighlight = white.*

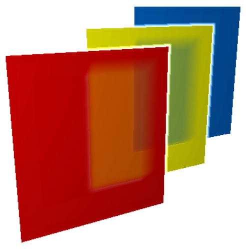


Figure 3.17: Same as in figure 3.16 with last layer opaque.

Algorithm 8: Composition of final pixel colour using fields β, γ

```
Data: linked list A
Result: Colour c of a pixel
1 $c = 0$
2 $\alpha = 1$
3 for int i = 0; i < size(A); i++ do
4     $\hat{\alpha_i} = (1 - \gamma_i)(\alpha_i + (1 - \alpha_i)\beta_i)$
5     $c = c + \alpha\hat{\alpha_i}c_i$
6     $\alpha = \alpha(1 - \alpha_i)$
7 end
8 $c = \alpha c_{background}$
```

Please note that algorithm 8 presented by Carnecky[12] is not complete and the γ field will not work not produce image that was presented in [12]. The $\hat{\alpha_i}$ will be going to zero for high gamma values and on the line 5 it will therefore produce dark halo highlight, not white as presented in the paper. We therefore suggest following change in the algorithm 9. Results of the composition using the algorithm 8 is shown in figure 3.15, while result of algorithm 9 is shown in figure 3.16. To produce the same result as in figure 2.11 we also set the last layer to be always opaque which can be seen in figure 3.17.

Algorithm 9: Composition of final pixel colour using fields β, γ

```
Data: linked list A
Result: Colour c of a pixel
1 $c = 0$
2 $\alpha = 1$
3 for int i = 0; i < size(A); i++ do
4     $c+ = \gamma * colourOfHaloHighlight$
5     $\hat{\alpha_i} = (\alpha_i + (1 - \alpha_i)\beta_i)$
6     $c = c + \alpha\hat{\alpha_i}c_i$
7     $\alpha = \alpha(1 - \alpha_i)$
8 end
9 $c = \alpha c_{background}$
```

3.4 Summary

In this Chapter algorithms based on peeling as well as on linked lists were introduced and compared based on the ease of use and most importantly on extendability by other opacity modulation techniques. Algorithms dual depth peeling and original depth peeling are intuitive to use in case of modulation by depth between samples along the ray, even in case of modulation by distance to defined focus area. It is however hard and not worth the effort to solve opacity modulation considering any non-local information. Therefore use of peeling algorithms on such techniques is not advised.

We have found the Illustration Buffer to be really powerful tool offering intuitive traverse of the object surfaces and therefore retrieving any non-local information easily.

Chapter 4

Design and Implementation

In this chapter implementation of the Illustration Buffer is presented as well as used technologies and overall structure of the application.

4.1 Used Technologies

In this section used technologies are described as is their purpose in our application. We use ANSI C++ and comply the C++11 standard as a bottom layer technology.

4.1.1 OpenGL and GLSL

OpenGL[4] is an environment for development of interactive 2D and 3D graphics applications. It offers powerful API for communication with the GPU, managing buffers, textures and much more. We use OpenGL to implement all the structures we need by the Illustration Buffer as well as the skeleton of the algorithm itself.

GLSL[3] is a shading language with direct support of OpenGL enabling management of all operations that occurs at programmable points in OpenGL rendering pipeline. We use GLSL to implement our shaders that implement most of the algorithm logic. Opengl version 4.4.0 NVIDIA 331.113 is used along with GLSL version: 4.40 NVIDIA via Cg compiler.

4.1.2 GLM

GLM[2] is a header only library for working with vector and matrix mathematics in graphics applications mostly. We can find definitions of new types with standard vector and matrix operations with overloaded operators for ease of use. We use this library to store vector and matrices data to pass to the GLSL shading languange.

4.1.3 RapidJSON

A fast JSON parser/generator for C++ with both SAX/DOM style API[7] is used to parse our configuration files stored in the JSON format.

4.1.4 QT Framework

QT Framework[5] is a framework for development of cross-platform applications as well as their user interface (UI). It is divided into several modules to minimize the unnecessary code for specific application. In our implementation we use only four modules of 5.3.0 QT Framework version:

- **Core** is a base of the QT framework offering many extensions of c++11 containers, functionality and it is necessary for development with QT.
- **Gui** module cares about most of the GUI elements.
- **Widgets** module extends the Gui module by adding widgets that we use for modal windows and better layout of the application.
- **Opengl** module offers basic opengl integration with OpenGL context management. It also offers a wrapper for all OpenGL commands since version QT5 but that offers only limited functionality of OpenGL ES 2.0. We therefore use a GLEW library for that purpose.

4.1.5 The OpenGL Extension Wrangler Library

The OpenGL Extension Wrangler Library (GLEW)[1] is used to manage OpenGL extensions as well as to provide access to OpenGL functionality we require. We use version 1.11.0 in our application.

4.2 Application Structure

First we describe classes that are most important for the algorithm. Since the application consists of many classes we show relation diagrams in parts. We do not describe shaders or processing workflow in this section.

`IllustrationBufferAlgorithm` class is responsible for creation of all buffers, textures and auxiliary structures before the algorithm starts. Then it consists of several methods each describing one stage of the algorithm. These are discussed further in this section.

`AlgTemplateObject` class serves as a renderer of the object. It also initializes attached algorithm and then calls methods of the algorithm when enabled.

`Context` is class passed to both `IllustrationBufferAlgorithm` and `AlgTemplate-Object` and is used as configuration container. It offers inner configuration as well as user parameters controlled by UI to other classes. Since there are 42 field members of the Context class we omit that information in all diagrams.

`Object` is a parent of `AlgTemplateObject` and is is responsible for the geometry and memory management for object geometry, materials and such.

`ObjMtlLoader` class implements simple parsing of OBJ format and it also supports materials and loading object groups. It loads informations about model name and path form JSON file providing simple way of importing OBJ models without changing the source code.

`ShaderProgramLoader` class loads the shaders configuration from attached JSON file. This allows us to add and remove shader programs dynamically without changing the code.

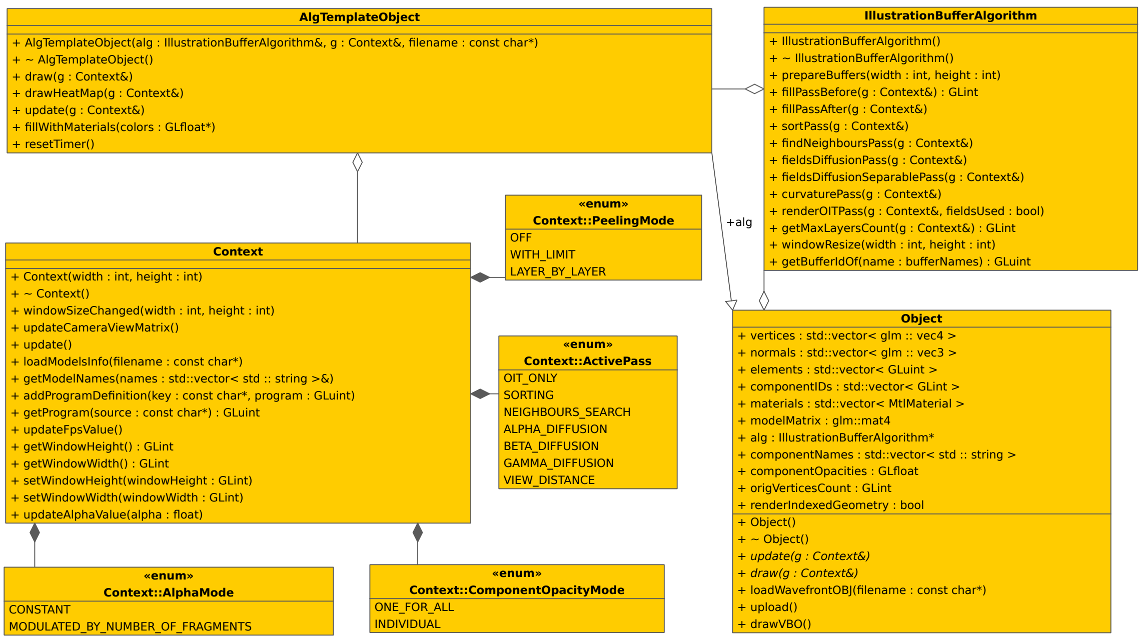

AlgTemplateObject
+ AlgTemplateObject(alg : IllustrationBufferAlgorithm&, g : Context&, filename : const char*)
+ ~ AlgTemplateObject()
+ draw(g : Context&)
+ drawHeatMap(g : Context&)
+ update(g : Context&)
+ fillWithMaterials(colors : GLfloat*)
+ resetTimer()
IllustrationBufferAlgorithm
+ IllustrationBufferAlgorithm()
+ ~ IllustrationBufferAlgorithm()
+ prepareBuffers(width : int, height : int)
+ fillPassBefore(g : Context&) : GLint
+ fillPassAfter(g : Context&)
+ sortPass(g : Context&)
+ findNeighboursPass(g : Context&)
+ fieldsDiffusionPass(g : Context&)
+ fieldsDiffusionSeparablePass(g : Context&)
+ curvaturePass(g : Context&)
+ renderOITPass(g : Context&, fieldsUsed : bool)
+ getMaxLayersCount(g : Context&) : GLint
+ windowResize(width : int, height : int)
+ getBufferIdOf(name : bufferNames) : GLuint
+alg
«enum»
Context::PeelingMode
OFF
WITH_LIMIT
LAYER_BY_LAYER
Context
+ Context(width : int, height : int)
+ ~ Context()
+ windowSizeChanged(width : int, height : int)
+ updateCameraViewMatrix()
+ update()
+ loadModelsInfo(filename : const char*)
+ getModelNames(names : std::vector< std :: string >&)
+ addProgramDefinition(key : const char*, program : GLuint)
+ getProgram(source : const char*) : GLuint
+ updateFpsValue()
+ getWindowHeight() : GLint
+ getWindowWidth() : GLint
+ setWindowHeight(windowHeight : GLint)
+ setWindowWidth(windowWidth : GLint)
+ updateAlphaValue(alpha : float)
«enum»
Context::ActivePass
OIT_ONLY
SORTING
NEIGHBOURS_SEARCH
ALPHA_DIFFUSION
BETA_DIFFUSION
GAMMA_DIFFUSION
VIEW_DISTANCE
Object
+ vertices : std::vector< glm :: vec4 >
+ normals : std::vector< glm :: vec3 >
+ elements : std::vector< GLuint >
+ componentIDs : std::vector< GLint >
+ materials : std::vector< MtlMaterial >
+ modelMatrix : glm::mat4
+ alg : IllustrationBufferAlgorithm*
+ componentNames : std::vector< std :: string >
+ componentOpacities : GLfloat
+ origVerticesCount : GLint
+ renderIndexedGeometry : bool
+ Object()
+ ~ Object()
+ update(g : Context&)
+ draw(g : Context&)
+ loadWavefrontOBJ(filename : const char*)
+ upload()
+ drawVBO()
«enum»
Context::AlphaMode
CONSTANT
MODULATED_BY_NUMBER_OF_FRAGMENTS
«enum»
Context::ComponentOpacityMode
ONE_FOR_ALL
INDIVIDUAL

Diagram 1: Diagram show relations between AlgTemplateObject, IllustrationBufferAlgorithm, Object and Context along with its enumerations.

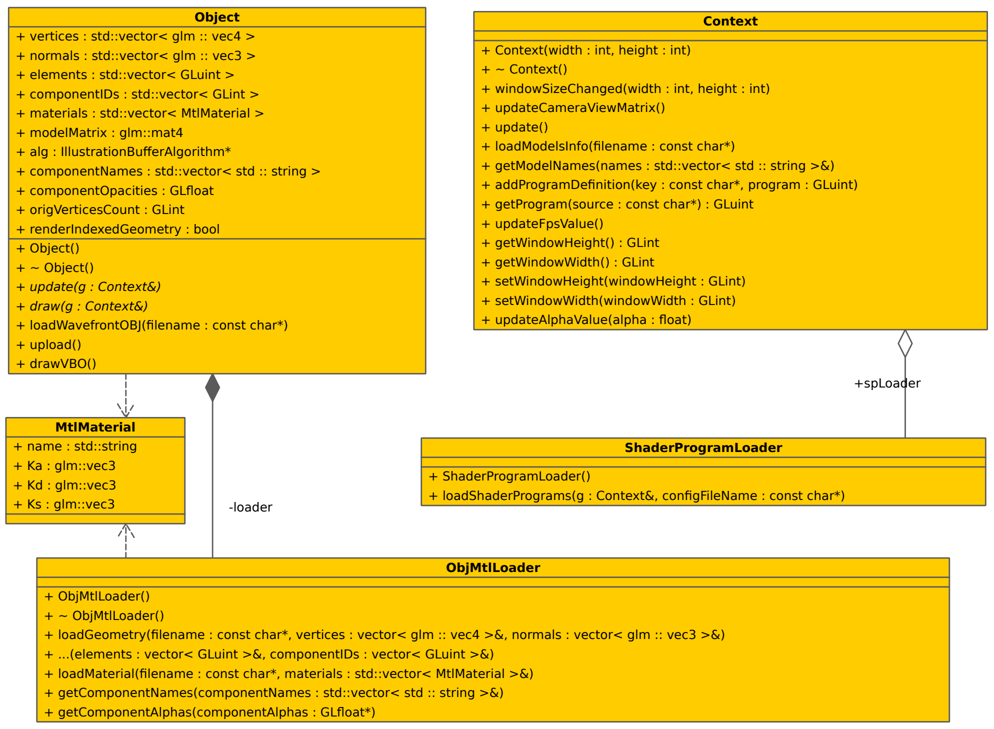

Object
+ vertices : std::vector< glm :: vec4 >
+ normals : std::vector< glm :: vec3 >
+ elements : std::vector< GLuint >
+ componentIDs : std::vector< GLint >
+ materials : std::vector< MtlMaterial >
+ modelMatrix : glm::mat4
+ alg : IllustrationBufferAlgorithm*
+ componentNames : std::vector< std :: string >
+ componentOpacities : GLfloat
+ origVerticesCount : GLint
+ renderIndexedGeometry : bool
+ Object()
+ ~ Object()
+ update(g : Context&)
+ draw(g : Context&)
+ loadWavefrontOBJ(filename : const char*)
+ upload()
+ drawVBO()
Context
+ Context(width : int, height : int)
+ ~ Context()
+ windowSizeChanged(width : int, height : int)
+ updateCameraViewMatrix()
+ update()
+ loadModelsInfo(filename : const char*)
+ getModelNames(names : std::vector< std :: string >&)
+ addProgramDefinition(key : const char*, program : GLuint)
+ getProgram(source : const char*) : GLuint
+ updateFpsValue()
+ getWindowHeight() : GLint
+ getWindowWidth() : GLint
+ setWindowHeight(windowHeight : GLint)
+ setWindowWidth(windowWidth : GLint)
+ updateAlphaValue(alpha : float)
+spLoader
MtlMaterial
+ name : std::string
+ Ka : glm::vec3
+ Kd : glm::vec3
+ Ks : glm::vec3
-loader
ShaderProgramLoader
+ ShaderProgramLoader()
+ loadShaderPrograms(g : Context&, configFileName : const char*)
ObjMtlLoader
+ ObjMtlLoader()
+ ~ ObjMtlLoader()
+ loadGeometry(filename : const char*, vertices : vector< glm :: vec4 >&, normals : vector< glm :: vec3 >&)
+ ...(elements : vector< GLuint >&, componentIDs : vector< GLuint >&)
+ loadMaterial(filename : const char*, materials : std::vector< MtlMaterial >&)
+ getComponentNames(componentNames : std::vector< std :: string >&)
+ getComponentAlphas(componentAlphas : GLfloat*)

Diagram 2: Diagram shows relations between AlgTemplateObject, IllustrationBufferAlgorithm, Object and Context along with its enumerations.

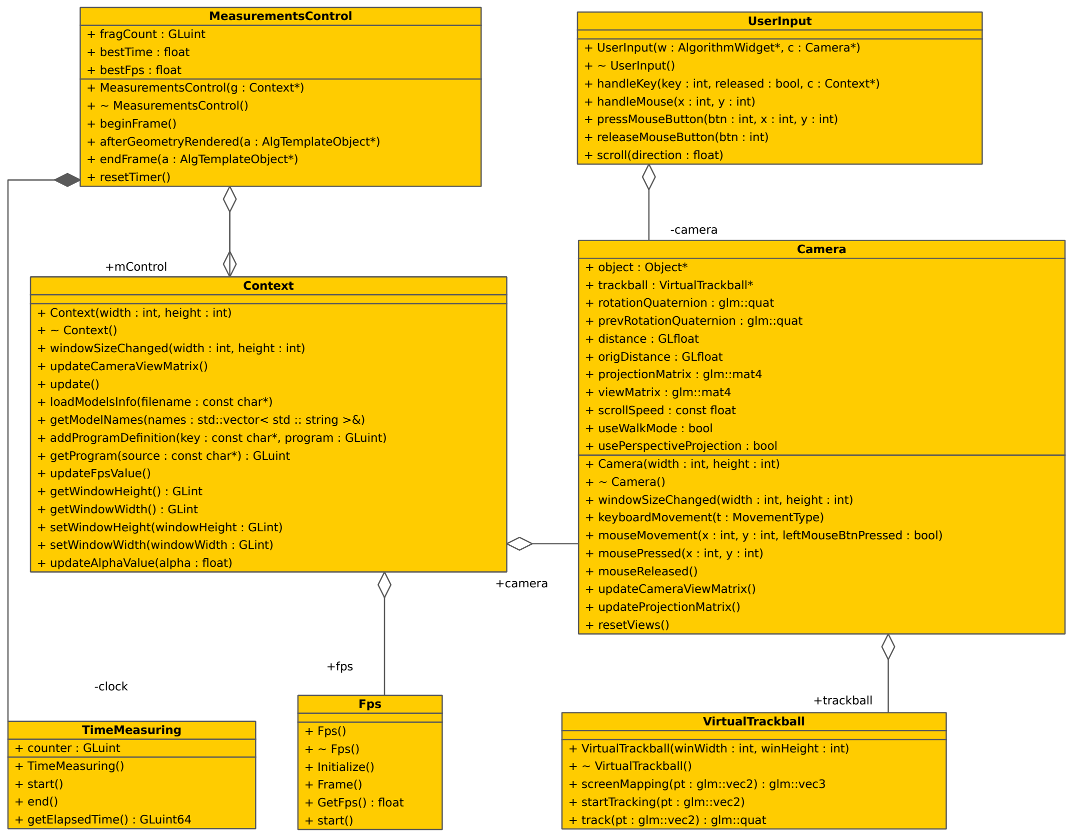

MeasurementsControl
+ fragCount : GLuint
+ bestTime : float
+ bestFps : float
+ MeasurementsControl(g : Context*)
+ ~ MeasurementsControl()
+ beginFrame()
+ afterGeometryRendered(a : AlgTemplateObject*)
+ endFrame(a : AlgTemplateObject*)
+ resetTimer()
UserInput
+ UserInput(w : AlgorithmWidget*, c : Camera*)
+ ~ UserInput()
+ handleKey(key : int, released : bool, c : Context*)
+ handleMouse(x : int, y : int)
+ pressMouseButton(btn : int, x : int, y : int)
+ releaseMouseButton(btn : int)
+ scroll(direction : float)
-camera
+mControl
Context
+ Context(width : int, height : int)
+ ~ Context()
+ windowSizeChanged(width : int, height : int)
+ updateCameraViewMatrix()
+ update()
+ loadModelsInfo(filename : const char*)
+ getModelNames(names : std::vector< std :: string >&)
+ addProgramDefinition(key : const char*, program : GLuint)
+ getProgram(source : const char*) : GLuint
+ updateFpsValue()
+ getWindowHeight() : GLint
+ getWindowWidth() : GLint
+ setWindowHeight(windowHeight : GLint)
+ setWindowWidth(windowWidth : GLint)
+ updateAlphaValue(alpha : float)
Camera
+ object : Object*
+ trackball : VirtualTrackball*
+ rotationQuaternion : glm::quat
+ prevRotationQuaternion : glm::quat
+ distance : GLfloat
+ origDistance : GLfloat
+ projectionMatrix : glm::mat4
+ viewMatrix : glm::mat4
+ scrollSpeed : const float
+ useWalkMode : bool
+ usePerspectiveProjection : bool
+ Camera(width : int, height : int)
+ ~ Camera()
+ windowSizeChanged(width : int, height : int)
+ keyboardMovement(t : MovementType)
+ mouseMovement(x : int, y : int, leftMouseBtnPressed : bool)
+ mousePressed(x : int, y : int)
+ mouseReleased()
+ updateCameraViewMatrix()
+ updateProjectionMatrix()
+ resetViews()
+camera
+fps
-clock
+trackball
TimeMeasuring
+ counter : GLuint
+ TimeMeasuring()
+ start()
+ end()
+ getElapsedTime() : GLuint64
Fps
+ Fps()
+ ~ Fps()
+ Initialize()
+ Frame()
+ GetFps() : float
+ start()
VirtualTrackball
+ VirtualTrackball(winWidth : int, winHeight : int)
+ ~ VirtualTrackball()
+ screenMapping(pt : glm::vec2) : glm::vec3
+ startTracking(pt : glm::vec2)
+ track(pt : glm::vec2) : glm::quat

Diagram 3: `Camera` class is attached to `UserInput` to receive commands and is accessible via `Context`. `VirtualTrackball` is used in case of orbiting camera mode, walking mode is implemented as part of the `Camera` class. `Fps` and `TimeMeasurements` are used for measurements.

`UserInput` captures all user input from mouse and the keyboard and adjusts Context and Camera variables accordingly.

`Camera` while model matrix is stored in the `Object`, camera is holding view and projection matrices. Quaternions are used for rotations of the camera. Camera is capable of both perspective and orthogonal projections, walk mode is also implemented here. It uses `VirtualTrackball` for modifications of the view matrix if orbiting mode is enabled.

`VirtualTrackball` class is notified when user clicks and starts dragging. It than for each frame computes the difference angle between the start point and the end point, both projected to the ball surface. Please refer to [24] for description of *virtual trackball* method.

`Fps` class uses `QElapsedTimer` from QT to compute time difference between last and current frame and turn it into the FPS value.

`TimeMeasurements` while `Fps` measures time using CPU clock, `TimeMeasurements` class measures time w.r.t. GPU. It uses opengl queries `GL_TIME_ELAPSED`, `GL_QUERY_RESULT_AVAILABLE`, `GL_QUERY_RESULT` to measure processing on GPU more precisely.

`MeasurementsControl` captures important data in time given by the measured task. Such logic is used from the AlgTemplateObject in the beginning of each frame, after the geometry is rendered and when frame ends.

4.2.1 Graphical User Interface

Graphical interface is built on top of QT Framework using *Qt Designer Form Class* allowing easy management of the elements using graphical designer provided by *QTCreator IDE*. Each *form* is attached to its own class where signals and events are captured and can be easily passed to the application logic. GUI consists from classes shown in Diagram 4. How the GUI of our implementation looks like can be seen in Appendix E.

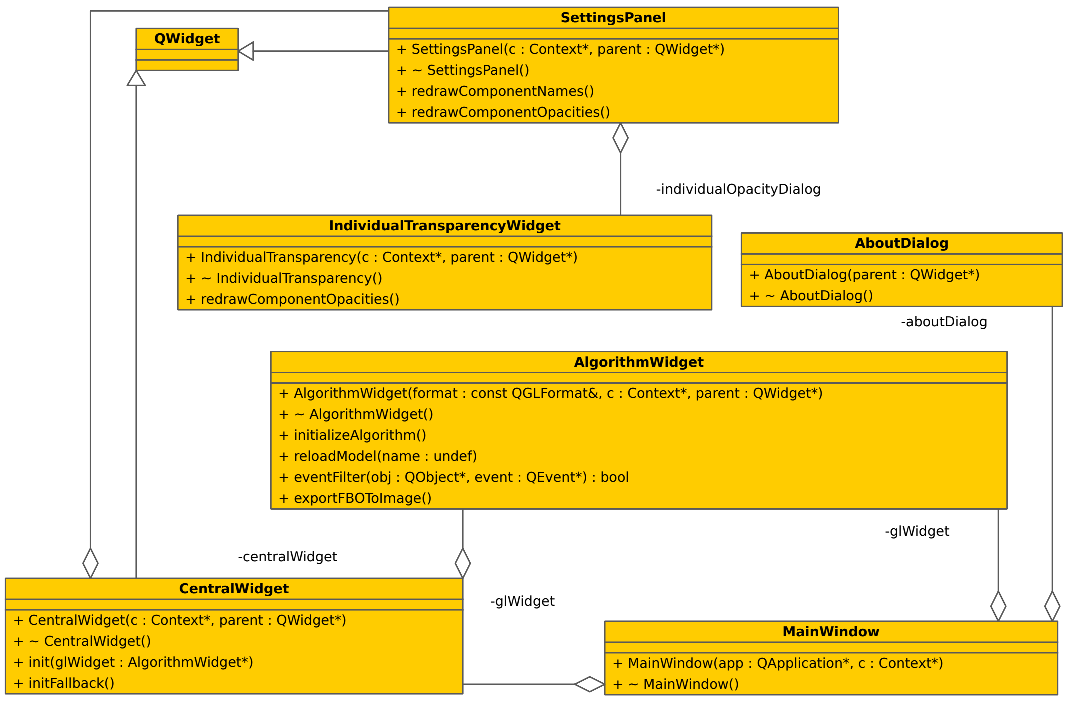

SettingsPanel
+ SettingsPanel(c : Context*, parent : QWidget*)
+ ~ SettingsPanel()
+ redrawComponentNames()
+ redrawComponentOpacities()
QWidget
-individualOpacityDialog
IndividualTransparencyWidget
+ IndividualTransparency(c : Context*, parent : QWidget*)
+ ~ IndividualTransparency()
+ redrawComponentOpacities()
AboutDialog
+ AboutDialog(parent : QWidget*)
+ ~ AboutDialog()
-aboutDialog
AlgorithmWidget
+ AlgorithmWidget(format : const QGLFormat&, c : Context*, parent : QWidget*)
+ ~ AlgorithmWidget()
+ initializeAlgorithm()
+ reloadModel(name : undef)
+ eventFilter(obj : QObject*, event : QEvent*) : bool
+ exportFBOToImage()
-glWidget
-centralWidget
-glWidget
CentralWidget
+ CentralWidget(c : Context*, parent : QWidget*)
+ ~ CentralWidget()
+ init(glWidget : AlgorithmWidget*)
+ initFallback()
MainWindow
+ MainWindow(app : QApplication*, c : Context*)
+ ~ MainWindow()

Diagram 4: There is one `MainWindow`, which is partitioned into `CentralWidget` and `SettingsPanel`. `CentralWidget` is a place for `AlgorithmWidget` which renders its content using OpenGL. Application uses two dialogs - `AboutDialog` and `IndividualTransparencyWidget`.

`MainWindow` is a descendant of `QMainWindow`. It creates all the widgets as shown in Diagram 4 as well the global menu. Global menu serves for change of the model, exporting images and toggling the `AboutWidget`.

`SettingsPanel` is tabbed interface used for the control of all variables that affect how currently selected stage works. It also consist of camera options, groups options and custom opacity per component settings.

`CentralWidget` is only a wrapper class for the AlgorithmWidget.

`AlgorithmWidget` class is responsible for the initialization of OpenGL context as well as for checking application requirements.

`IndividualTransparencyWidget` if enabled in the settings, this widget dynamically generates sliders for each component in rendered model. Sliders represent opacity value.

`AboutDialog` shows informations about the application, author and purpose.

4.2.2 Additional Notes to Application Structure

In our implementation class `IllustrationBufferAlgorithm` is responsible for the invocation of shaders that not only create the Illustration Buffer but are also used for variety of opacity modulations. While it would be more coherent to separate the buffer creation and algorithms using the Illustration Buffer to separate classes, it is all in one class in our implementation. Reason for that is optimization of the shader invocations. For example some data computation needed by future opacity modulation is possible to do in the same pass as the sorting of samples.

4.3 Creation of the Illustration Buffer

Implementation of Illustration Buffer algorithm of its stages is described in this section as well as used structures.

4.3.1 Used structures

To store the Illustration Buffer in the memory of GPU buffers for linked lists head pointers, next pointers, data for each fragment, indices of surrounding triangles as well as fragment counter as described in Chapter 3. Creation of such structures occurs in `prepareBuffers` method of `IllustrationBufferAlgorithm` class.

`GL_ATOMIC_COUNTER_BUFFER` is used to implement the counter of all rendered fragments to receive their unique id. It is clear that using only one counter for entire scene might become the bottleneck of the pipeline. We have therefore experimented also with partitioning the viewport into the grid as shown in image 4.1.

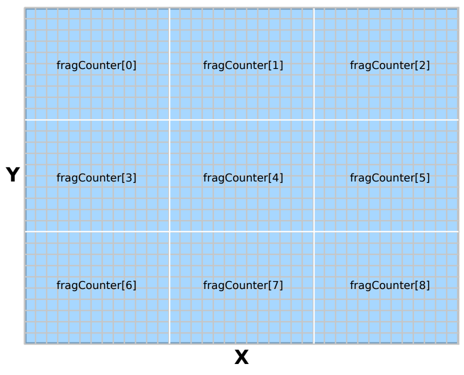


Figure 4.1: Viewport of dimensions X, Y divided into 9 regions each with its own atomic counter $fragCounter[i], i \in \{0..8\}$

Even though we get some improvement in speed, this method is not memory efficient. With such partitioning we have to reserve fixed amount of expected fragments count in the fragment data buffer. Even if we have enough memory, cashing will not be as effective because of big empty areas between each *fragCounter* address space in the fragment data buffer.

Since we need to not only read but also write to the buffers in the same shader invocation, traditional GL_TEXTURE_2D cannot be used in case of most structures we need. We therefore use OpenGL extension `ARB_shader_image_load_store`. This extension brings functions imageLoad(), imageStore() and also many atomic operations imageAtomic*().

Buffer *pixelHead* which stores pointers to first node of each per pixel linked lists is defined as GL_TEXTURE_2D with GL_R32I type of the same size as the viewport. To use is as *image* w.r.t. `ARB_shader_image_load_store` we need to bind it as a *ImageTexture*:

```
// In C++ source code
glTexImage2D(GL_TEXTURE_2D,0,GL_R32I,width,height,0,GL_RED_INTEGER,GL_INT,0);

// And in the shader:
uniform layout(binding = 0, r32i) coherent iimage2D u_pixelHead;
```

We use also buffer *pixelCount* which stores lengths of per pixel linked lists. It is therefore also 2D and we use the same format as for the buffer *pixelHead*.

We use GL_TEXTURE_BUFFER for buffers *fragData*,*fragData2*, *fragNext* and *fragElements*. These are one dimensional texture buffers created as follows (only fragData is shown):

```
    buffersIds[fragData] = 0;
    glGenBuffers(1, &buffersIds[fragData]);
    glBindBuffer(GL_TEXTURE_BUFFER, buffersIds[fragData]);
    glBufferData(GL_TEXTURE_BUFFER, fragCountMaxData, 0, GL_DYNAMIC_DRAW);
    glBindBuffer(GL_TEXTURE_BUFFER, 0);

    textureIds[fragData] = 0;
    glGenTextures(1, &textureIds[fragData]);
    glBindTexture(GL_TEXTURE_BUFFER, textureIds[fragData]);
    glTexBuffer(GL_TEXTURE_BUFFER, GL_RGBA32UI, buffersIds[fragData]);
    glBindTexture(GL_TEXTURE_BUFFER, 0);
```

Internal formats used by those buffers are: GL_RGBA32UI for *fragData*,*fragData2* and *fragElements* and GL_R32I for *fragNext*. Buffer *fragElements* consists of vertex indices of the triangle it belongs to. We need this for neighbors search is described in Chapter 3.

Table 4.1 shows what data we store when experimenting with the Illustration Buffer. We use the spanning mechanism for *fragData* buffer to store 4*4 unsigned integers per fragment. Four uvec4 are therefore reserved in *fragData* buffer per fragment. We describe the data stored further in this Chapter.

	uint	**uint**	**uint**	**uint**
fragID*spanSize	RGBA Color	Depth	LayerIndex	DistToNext Layer
fragID*spanSize + 1	Left	Right	Bottom	Top
fragID*spanSize + 2	$\alpha + 2b_\alpha$	$\beta + 2b_\beta$	$\gamma + 2b_\gamma$	
fragID*spanSize + 3	Normalized normal	Curvature		

Table 4.1: Table shows data stored per fragment. First column shows the position in the fragData buffer according to the spanning mechanism. SpanSize = 4 is used. One row of the table is represented as vector of 4 unsigned integers (uvec4).

4.3.1.1 Formats Packing

Note cells *RGBA Color* and *Normalized normal* in table 4.1, where we store vector data to unsigned integer. To save used memory space packing of 4 floats $f, f \in < 0, 1 >$ to one unsigned integer is used. For example color and alpha channel are stored to one unsigned integer as:

```
// packing formats in GLSL using bitwise shifts
   uint color32UI = (uint(color.r * 255) << 24) | (uint(color.g * 255) << 16)
    | (uint(color.b * 255) << 8) | uint(color.a * 255);
// For one float value to uint we can use built-in GLSL functions
    uint depth32UI = floatBitsToUint(abs(depth));
```

4.3.2 Buffer Filling

In our implementation creation of the Illustration Buffer is separated to two stages.

fillPassBefore method ensures that buffers *pixelCount* and *pixelHead* are reset to initial state. This can be done using `GL_PIXEL_UNPACK_BUFFER`. Other buffers can be simply rewritten by new data but for image to not interfere with previous results *pixelCount* and *pixelHead* need to be reset. Atomic counter used for fragments indices needs to reset as well. This can be done using following code snippet:

```
glBindBuffer(GL_ATOMIC_COUNTER_BUFFER, buffersIds[fragCount]);
GLuint* p=(GLuint*)glMapBufferRange(GL_ATOMIC_COUNTER_BUFFER,0,sizeof(GLuint),
                                     GL_MAP_WRITE_BIT |
                                     GL_MAP_INVALIDATE_BUFFER_BIT |
                                     GL_MAP_UNSYNCHRONIZED_BIT);

p[0] = ourInitialValue;
glUnmapBuffer(GL_ATOMIC_COUNTER_BUFFER);
glBindBuffer(GL_ATOMIC_COUNTER_BUFFER, 0);
```

Then all buffers are binded as uniforms to the current shader pass we call the *fill-Pass*. Since OpenGL cannot know what purpose of our buffer is we need to specify how the buffer will be accessed. That is done when binding the buffer using `GL_READ_WRITE`, `GL_READ_ONLY`, `GL_WRITE_ONLY` flags and memory qualifiers `coherent`, `volatile`, `restrict`, `read- only`, `writeonly` in the shader. It is responsibility of the developer to set these since write operations when using image load/store are not automatically coherent[6].

drawVBO method is then called in the `AlgTemplateObject` that renders the geometry using indexed geometry[24] and `GL_ARRAY_BUFFER` buffers that store model data in the memory of GPU and are uploaded only once before the algorithm start.

When geometry is rendered using the `fillPass` all fragments are discarded and stored in the Illustration Buffer:

```
vec4 color = computeShadingAndLighting(normal, eyeDirection);
color.a = u_alpha;

uint newFragId = atomicCounterIncrement(u_fragCount);
ivec2 coords = ivec2(gl_FragCoord.xy);

int prevFragId = imageAtomicExchange(u_pixelHead, coords, int(newFragId));
imageAtomicExchange(u_fragNext, int(newFragId), int(prevFragId));
imageStore(u_fragData, int(newFragId*u_dataSpanSize), pack(color, depth));
// Here all other data needed by current setup can be stored as well.
// Triangle indices for the neighbor search, fragment normal for curvature
// estimation, and others.

imageAtomicAdd(u_pixelCount, coords, 1); // increase number of processed
    fragments at these coordinates.
```

Now all the data are stored in presented buffers in GPU memory. Geometry therefore does not have to be rendered again and again as in case of the peeling methods which will turn out to be very important in measurements. For the Illustration Buffer to be complete we still have to sort accumulated fragments and find their geodesic neighbors along the surface.

4.3.3 Sorting

In our application two sorting methods are implemented. One sorting the linked list without using any auxiliary structures and second where array of fixed size is used for sorting.

Sorting the Linked List by Insertion sort can be done easily using two *fragNext* buffers. One to be filled initially and second that will be used for adding sorted fragments as shown in algorithm 10. Since our insertion sort differs in two used structures for next pointers we show used GLSL source code instead of pseudocode. We can see this procedure as a analogy to two linked lists A,B. A is unsorted and B consists only from copy of head in A. Then we remove node a from the front of A and insert them to B. To be able to remove $a.next$ from A and insert it to B, we would have to remember what was the original $a.next$ since inserting the a node to B may change its next pointer. In single linked lists this could be solved also by copies of the nodes instead of their removal from A. We solve this issue as mentioned by two buffers for the next pointers.

Sorting in Array of Fixed Size Number of accesses to the buffers is a bottleneck of the previous method. We load all values and their next pointers to static arrays of fixed size (64 in our implementation). This array is then sorted and results are stored back to the buffers.

Algorithm 10: creating the concurrent linked lists structure

```
Data: Buffers u_fragNext and u_fragNext2, u_fragData and u_pixelHead
Result: Sorted next pointers in the u_fragNext2 and fixed head pointer in u_pixelHead.
int sortedSize = 1;
int head = imageLoad(u_pixelHead, ivec2(gl_FragCoord.xy)).x;
float headDepth = uintBitsToFloat(imageLoad(u_fragData, head*u_dataSpanSize).y);
int new = imageLoad(u_fragNext, head);
float newDepth = uintBitsToFloat(imageLoad(u_fragData, new*u_dataSpanSize).y);
int newCounter = 1;
int curr,prev;
float currDepth;
while sortedSize < totalCount do
    if newDepth < headDepth then
        imageStore(u_fragNext2, new, ivec4(head,0,0,0));
        head = new;
        headDepth = newDepth;
        new = imageLoad(u_fragNext, new);
        newDepth = uintBitsToFloat(imageLoad(u_fragData, new*u_dataSpanSize).y);
        sortedSize++;
        continue;
    end
    prev = head;
    curr = imageLoad(u_fragNext2, head);
    currDepth = uintBitsToFloat(imageLoad(u_fragData, curr*u_dataSpanSize).y);
    int innerCounter = 0;
    while innerCounter <= sortedSize && currDepth < newDepth do
        prev = curr;
        curr = imageLoad(u_fragNext2, curr);
        currDepth = uintBitsToFloat(imageLoad(u_fragData, curr*u_dataSpanSize).y);
        innerCounter++;
    end
    imageStore(u_fragNext2, prev, ivec4(new,0,0,0));
    imageStore(u_fragNext2, new, ivec4(curr,0,0,0));
    new = imageLoad(u_fragNext, new);
    newDepth = uintBitsToFloat(imageLoad(u_fragData, new*u_dataSpanSize).y);
    sortedSize++;
end
imageStore(u_pixelHead, ivec2(gl_FragCoord.xy), ivec4(head,0,0,0));
```

4.3.4 Neighbors Location

Proposed indices motivated method is implemented to find neighbors along the surface. As described in 3.1.5.2 every vertex is passed all 3 vertex indices of incident triangle. We use `GL_ARRAY_BUFFER` to pass pre-processed indices to the vertex shader when geometry is rendered.

Indices are passed to the fragment shader as `flat uvec3 v_triangleIndices` so that its values are not interpolated between fragments and indices stay valid. This information is then stored per fragment in the *fragData* buffer.

FindNeighborsPass is the name of shader program that finds neighbors for all stored fragments if exist. First fullscreen quad is rendered to access the buffers by the pixel coordinates. Location of the neighbors is then trivial as described in algorithm 6 in 3.1.5.2. Refered algorithm will return ids of all geodesic neighbors if exist. We store those values into the *fragData* buffer as well.

4.3.5 Groups and Importance per Components

In our application two modes are available. In the default mode each component (part of the geometry) are set with the same initial opacity. In mode called *individual opacity* each component can have its own initial opacity value. In addition in both modes we can select which components are to be rendered.

Opacity shared by all components is implemented simply using uniform GLSL variable. Individual opacities `u_ComponentOpacities[512]` are passed to the shader as an uniform array where *componentID* is used to address it. We would use same mechanism for u_AllowedComponents[512], there is however a hardware limitation considering the number of uniform values that can be passed in one shader invocation. We therefore merge two arrays to one in following manner: there is no need to have binary array u_AllowedComponents[512] for components to be used. If a value of `u_ComponentOpacities[componentID]` is equal to zero, its fragments are not stored to the linked lists at all. If other than zero, fragments are stored and individual opacity is set. This not only saves number of used uniform values and stores only necessary - visible fragments to per pixel linked lists.

4.3.6 Visualization of the Illustration Buffer

Several tools to visualize and debug the Illustration Buffer are implemented. First we use the *transfer function* to visualize the depths of per pixel linked lists as a *heatmap*. This is simply done by copying the *pixelCount* buffer to CPU and finding the maximum depth to normalize all depths to range $< 0, 1 >$. Normalized value is then used as a coordinate to one dimensional texture that represents the transfer function. Results of such visualization are shown in figure 4.2

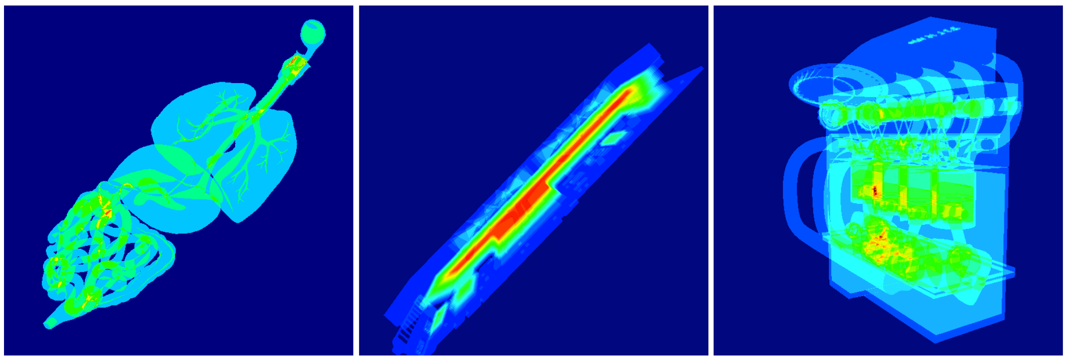


Figure 4.2: Heatmap gradient shows the linked lists size color coded in following manner: for gradient blue->green->yellow->red lowest sizes are blue and longest red.

To visualize found neighbors and correct order of fragments along the ray *peeling* of the Illustration Buffer is implemented. Two modes can be used in our application:

Peeling limit is a limiting number to the final pixel compositing equation. For example if set to 3, only first three fragments in each linked list are used for the computation of final color.

Layer by Layer is a mode where for number L only *L-ith* fragments are shown if exist.

4.4 Non Local Transparency

Non local transparency modulation methods presented by Carnecky et. al.[12] are implemented to demonstrate Illustration Buffer flexibility and ease of use. We have however used transparency field α in different way than in[12] as described further.

4.4.1 Transparency Fields

Let β, γ be transparency fields accompanied by binary fields b_β, b_γ as introduced in Chapter 3. Carnecky et. al. also use fields α, b_α for initial transparency. These were designed for complex flow surfaces with discontinuous initial transparency α in [12]. They use described physically motivated diffusion process to smooth the α, b_α values since any discontinuity in the transparency could be mistaken for a surface contour. We however do not have discontinuous initial transparency considering only one surface, we only use different initial values per surface which do not suffer from mentioned perception problem.

To test the difference between physical and non physical diffusion process we therefore modify the α field presented in [12] to be set fully opaque on the surface contour. To set fields α, β, γ along with binary values $b_\alpha, b_\beta, b_\gamma$ several prerequisites have to be met:

1. **fragmentIndex**, $fragmentLayerIndex \in \{1, ..., n\}$, where n is number of fragments in given linked list and $fragmentLayerIndex$ is increasing with the distance from the camera. We compute and store this simply during the *sortPass* as custom fragment data.

2. **boundaryN** is a number of directions for which current fragment is a boundary. This can be easily computed in the *findNeighborsPass* when all existing neighbors are known. BoundaryN is set to zero and increased by one with every direction where neighbor does not exist.

3. **indexSmallerThanNeighbours** is a binary field that is true if *fragmentIndex* is smaller than at least one *fragmentIndex* of its neighbors. This indicates fragment adjacency to the contour. (Please refer to Chapter 3 and figure 3.13)

4. **indexGreaterThanNeighbours** is also binary field that is true if fragment lies underneath some silhouette fragment and its *fragmentIndex* is therefore bigger than of one of its neighbors. (Please refer to Chapter 3 and figure 3.13)

With these values known we can finally set fields α, β, γ with respect to rules defined in Chapter 3:

```
if(boundaryN > 0){
  beta, aplha = 1.0f;
  b_alpha, b_beta = 1.0f;
}
if(layerIndexGreaterThanNeighbours){
  beta = 0.0f;
  b_beta = 0;
}
if(layerIndexSmallerThanNeighbours){
  gamma = 1.0f;
  b_gamma = 0;
}
```

Note that we have set the fields $\alpha, \beta = 1.0f$ at the surface boundary. While it may seem as a duplicate information we use than different diffusion techniques for both fields and compare the results as well as results of their combination. To save memory we store both values α and b_α as $\alpha + 2b_\alpha$ without loss of precision. All discussed fields are implemented in the end of the *findNeighborsPass* and stored to *fragData* buffer to positions discussed in *Used structures* section.

4.4.2 Diffusion Process

We now have fields α, β, γ initialized. Physical and non physical approximations can be computed. Since these approximations differ significantly when used alone, method of combining the physical and altered nonphysical diffusion process is proposed. We can motivate ourselves with figure 4.4, where we can see how inefficient is transport of values using discrete Laplace operator and artifacts caused by nonphysical process proposed by [12].

To overcome the read/write collisions during this pass, *ping pong computational scheme* is implemented in the *fieldsDiffusionPass* where the diffusion occurs. This is simply done by switching buffers *fragData* in input and *fragData2* in pass output in every iteration.

4.4.2.1 Physical Process

0.5	1	0.5
1	-6	1
0.5	1	0.5

Figure 4.3: Discrete Laplacian convolution kernel for 2D signals

As described we use *discrete Laplace operator* for the $\Delta\alpha^k$ estimation. Discrete Laplace operator's *convolution* kernel is commonly used in image processing. Kernel shown in figure 4.3 is in our implementation not applied to pixels as in image processing but to neighboring fragments in the Illustration Buffer.

This is simply implemented as a sum of the neighboring fragments with corresponding coefficient given by the convolution kernel shown in figure 4.3. Best results was achieved by normalization of final value by $\frac{1}{4}$. Second order central finite difference proposed by Carnecky et. al. is also implemented but not used due to high amount of visual errors. For its use please uncomment line with `#define CENTRAL_DIFFERENCE` in the `smoothFieldsPassLaplacian.frag` shader file.

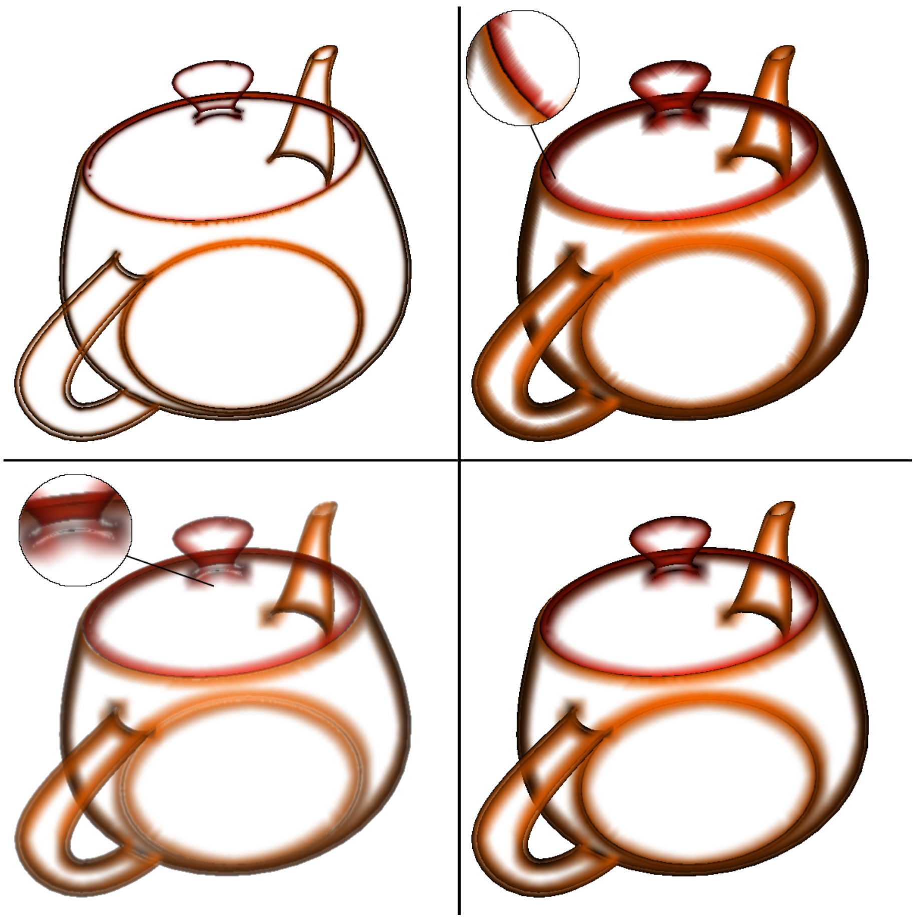


Figure 4.4: All images were created by 15 iterations of diffusion process. **Top left:** is result of convolution with discrete Laplace operator of α. **Top right:** is non physical approximation of β diffusion as in [12], suffering from any contour discontinuity. **Bottom left:** is result of applying discrete gaussian filter on β. We can see artifact called *ringing* in place of former contour which is typical for such kernels. **Bottom right:** is method used by us where result is combination of blurred non physical β diffusion process and output of discrete Laplacian operator on α.

4.4.2.2 Nonphysical Process

Nonphysical diffusion process introduced before is very straightforward to implement. In iteration over all fragments of a linked list we simply compute the β field for all existing neighbors of current fragment f_i as:

```
// GLSL code using built-in functions intBitsToFloat and max
// beta_diffusion = beta(f_i) initially
float bn=uintBitsToFloat(fieldDatan.y);//beta of currently processed neighbor
beta_diffusion = max(beta_diffusion,bn - lambda_beta);
```

In the end of the process *beta_ diffusion* value is in correct range due to maximizing process. Identical process is used for computation of *gamma_ diffusion* for silhouette enhancement.

4.4.2.3 Proposed α, β Diffusion

As we can see in figure 4.4, we get great quality (no box-like artifacts) for a price of very inefficient values transport in case of discrete Laplacian operator. Quite opposite is a result of the nonphysical diffusion of β field, where we get values quickly distributed but quality is very poor. We therefore use a combination of those techniques with minor modification.

First α field is processed by a discrete Laplacian operator and β is processed by the nonphysical process. To overcome the visual errors we propose to use the Gaussian filter to blur the result of β diffusion to lower amount discontinuities in such result. We implement this using user-defined number of blurring passes using 2D Gaussian filter kernel.

$$G(x) = e^{-\frac{x^2}{2\sigma^2}} \tag{4.1}$$

Since equation (4.1), with standard deviation σ controlling the filter width is both separable and radially symmetrical, its 2D version can be separated to two one-dimensional filters to speed up the process[19]. Field β is first blurred in the horizontal direction and result of such convolution is then blurred horizontally. In our implementation we use standard deviation $\sigma = 0.85$ and size of the 1D filter $size = 5$ (current fragment + 2 neighbors on both sides).

We can see in figure 4.4 that such blurring produces another type of visual error called *Ringing artifact*[24]. We therefore combine this blurred image with computed field α which has in principle high values exactly in places of the ringing artifacts. This is of course no coincidence since Laplacian operator is sensitive to *high-frequency* input which is a place where the *ringing artifacts* occur. Final image is composed using algorithm 8 where α field is used instead of the initial opacity α_i.

4.4.2.4 Automatic Transparency Field Setup

To get visually pleasant results we would have to adjust value λ_β manually with every change of diffusion iterations. To overcome this we allow *auto* mode in our application as the default mode. Value λ_β in the *auto* mode are computed simply as:

$$\lambda_\beta = 0.95/DIterations;$$

where 0.95 is a best value for our scenes found experimentally and *DIterations* is a number of diffuse iterations. For an *auto* mode we have found sufficient to set values $\lambda_\alpha = 1$ and $\lambda_\gamma = 0.45$.

4.5 Modulation by Distance Along the Ray

After sorting the fragments along the ray when creating the Illustration Buffer, single traverse of the linked list will give us the distances between the fragments along the ray. We store this information in the $fragData$ buffer.

To turn the distance to α channel, distances need to be normalized. We do this by uniform variable `u_distanceNorm` controlled by the user. Value α is then simply computed as:

$$alpha = clamp(distance/(u\_distanceNorm), 0.0, 1.0);$$

Please note that proper way to do this would be to find the maximum distance between fragments across all linked lists and use it for the normalization. If user-defined parameter λ_d is then defined in range $< 0, 1 >$, there is no need to clamp the final value:

$$alpha = (distance/u\_distanceNorm) * \lambda_d;$$

Finding the maximum is however not implemented since it would require additional computation time and we found the single user-defined variable *u_ distanceNorm* to be sufficient since we aim to keep the application as interactive as possible. For automatic normalization we can find the maximum by doing the modified *parallel prefix scan* with maximizing prefixes instead of summing[19].

4.5.1 Combined with Modulation by Differences in Shape

Fragment curvature is found during the `curvaturePass`. We use curvature computed from 4 geodesic neighbors as well as 8 adding also neighbors on the diagonals. For all fragments in the linked list curvature is computed as:

```
// neighbors is an array with 8 neighbor IDs if exist, zero otherwise.
// norm_n has to be unpacked from the unsigned int as discussed before
float curvature = 0f;
for(int n = 0; n < 8; n++){
  if(neighbours[n] > 0){
    uvec4 s = imageLoad(u_fragData, int(neighbours[n])*u_dataSpanSize+3);
    vec3 norm_n = vec3(s.x >> 16, (s.x >> 8) & 0xFF,(s.x) & 0xFF)/255.0;
    vec3 diff = norm_n - normal;
    curvature += length(diff);
  }
}
```

4.6 Final Rendering Pass

All gathered informations are composed together in the final rendering pass `renderOIT`. We again iterate over all fragments in each linked list and composite the final pixel color as discussed in Chapter 3. To enable the user to understand the purpose of each transparency field, following combinations are to be applied: (α), (β), (α and β), (α and γ), (β and γ) and finally (α and β and γ). Field is simply set zero if not used and no changes need to be done to the compositing equation implemented as:

```
src.rgb += gamma * vec3(1.0f); // white color of silhouette enhancement
alpha_r = (alpha + (1.0f - alpha)*beta);
finalFragmentColor.rgb += finalFragmentColor.a * alpha_r * src.rgb;
finalFragmentColor.a *= (1.0f - alpha);
finalFragmentColor.a *= (1.0f - beta);
```

Simple version of the compositing is used for the modulation by distance along the ray and the curvature effect:

```
alpha = clamp(max(distance,curvature*u_curvatureEffector),0.0,1.0);
finalFragmentColor.rgb += finalFragmentColor.a * (alpha * src.rgb);
finalFragmentColor.a *= (1.0f - alpha);
```

Chapter 5

Results and Discussion

In this Chapter results and measurements of our implementation of the Illustration Buffer are presented along with measurements of compared algorithms. For measurements of the *Per Pixel Linked Lists* and the *Illustration Buffer* we use our own implementation while *Depth peeling* and *Dual depth peeling* methods are measured using the *NVIDIA Graphics SDK 10*[8] where only more precise measuring was implemented.

While our implementation is cross-platform application and developed on Linux, NVIDIA Graphics SDK 10 is compilable only under the Visual C++ Compiler and therefore all measurements are conducted on a Windows machine with following configuration:

- **CPU:** Intel Core$^{\text{TM}}$2 Duo CPU E6850 @ 3.00GHz, 64KiB L1 cache, 4MiB L2 cache
- **Memory:** 4 × 2GiB DIMM DDR Synchronous 59392 MHz
- **GPU:** GeForce GTX 660, 2048 MB GDDR5 @ 6.0 Gbps, OpenGL 4.3
- **OS:** Windows 8 64-bit
- **Compiler:** GCC 4.9.1
- **OpenGL:** version: 4.4.0 NVIDIA 331.113
- **GLSL:** version: 4.40 NVIDIA via Cg compiler

Very precise GPU timers are used to measure application rendering time in nanoseconds. Queries `GL_TIME_ELAPSED`, `GL_QUERY_RESULT_AVAILABLE` and `GL_QUERY_RESULT` are used. For measurements of the total rendering time we use `Fps` class using `QElap-sedTimer`.

Twelve varied models are used to measure the Illustration Buffer characteristics. Each model is tested in two positions - one general and one with maximum possible linked lists length. All tested views along with the heatmaps visualizing linked lists depths are shown in figure 5.1. Please note that in all measurements image resolution 600 × 600 is used if not stated otherwise.

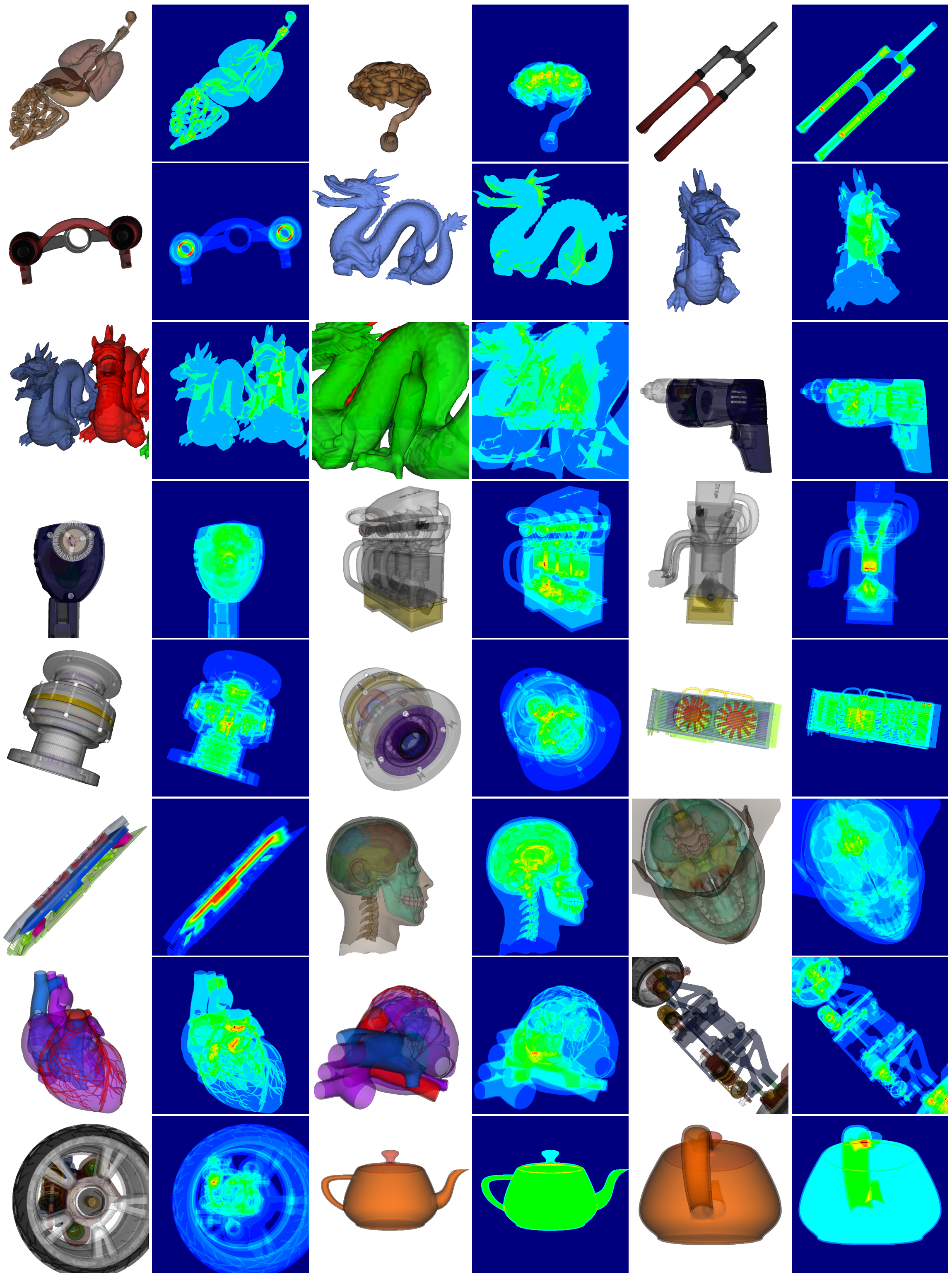


Figure 5.1: Measured projections of models along with the heatmaps indicating pixels with highest length of linked list as red. Models are sorted alphabetically by name.

5.1 The Illustration Buffer Creation

During the Illustration Buffer creation we examine relations between number of vertices, rendered fragments (not pixels!), lengths of the linked lists storing the data along the ray and of course rendering time on the GPU and rendering time combined with the CPU workload. Please note that choosing the sorting method also causes big performance impact. For simplicity we show this on *concurent per pixel linked lists* alone in the next section. To be able to examine the relations fully creation process is split to several stages by functionality.

- *Fill* stage renders the geometry and fills the buffers in GPU memory necessary for future Illustration Buffer creation. No pixels are rendered to the *framebuffer*.
- *Sort* stage sorts the fragments along the view ray. No pixels are rendered to the *framebuffer*.
- *FindNeighbors* stage is responsible for finding existing geodesic neighbors along the surface for each stored fragment, if exists. It also sets initial transparency fields. No pixels are rendered to the *framebuffer*.
- *Total* stage combines all before and composes final pixel colors.

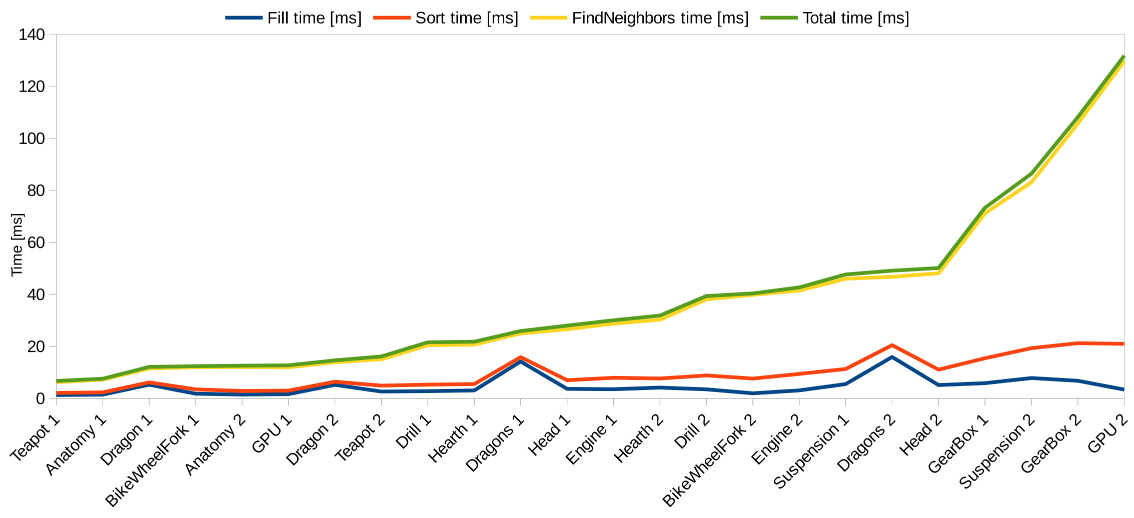


Figure 5.2: GPU Time in plit stages of the Illustration Buffer creation process. Models on the horizontal axis is sorted by the *Total* rendering time.

Results of the split stages measurements are shown in table 5.1. Table 5.2 then shows the `Total` rendering time in greater detail. These informations are necessary to decode what is happening in figure 5.2.

When *Fill* and *Sort* stages are compared we see two spikes for `Dragons 1,2` which are scenes with the greatest number of vertices. These show perfectly how application stays vertex bounded during the sort for big geometry instances with small linked lists lengths. Spike for the `Dragons 2` shows how workload of the sort stage is a lot bigger than for `Dragons 1`, where maximum linked list length was smaller.

	Fill		**Sort**		**FindNeighbors**		**Total**	
MODEL NAME	**TIME [ms]**	**FPS**	**TIME [ms]**	**FPS**	**TIME [ms]**	**FPS**	**TIME [ms]**	**FPS**
Anatomy 1	1,442	485	2,313	395	7,192	134	7,563	126
Anatomy 2	1,458	527	2,847	324	12,035	81	12,572	80
BikeWheelFork 1	1,790	510	3,459	274	11,995	83	12,390	82
BikeWheelFork 2	1,947	468	7,612	113	39,816	26	40,351	26
Drill 1	2,775	335	5,272	180	20,430	50	21,551	47
Drill 2	3,475	271	8,808	109	38,069	28	39,334	28
Engine 1	3,519	266	7,911	121	28,751	37	30,020	35
Engine 2	3,073	304	9,400	103	41,438	26	42,659	26
GearBox 1	5,851	165	15,437	65	71,184	17	73,291	17
GearBox 2	6,765	143	21,213	48	105,697	13	108,145	12
GPU 1	1,634	498	3,000	311	11,944	82	12,710	77
GPU 2	3,365	278	20,969	49	129,777	11	131,769	11
Head 1	3,666	258	6,987	138	26,569	39	27,981	38
Head 2	5,118	186	11,047	88	48,076	23	50,113	22
Hearth 1	3,047	309	5,517	173	20,675	50	21,785	47
Hearth 2	4,143	228	7,652	127	30,256	36	31,838	34
Suspension 1	5,485	178	11,293	87	45,992	24	47,641	24
Suspension 2	7,807	124	19,305	53	83,112	15	86,381	15
Teapot 1	1,265	648	2,088	437	6,225	155	6,699	147
Teapot 2	2,626	350	4,896	194	15,028	67	16,112	62
Dragon 1	5,250	184	6,130	158	11,493	86	12,101	83
Dragon 2	5,210	185	6,399	152	13,926	71	14,616	68
Dragons 1	14,167	71	15,836	65	24,961	42	25,894	41
Dragons 2	15,915	65	20,456	51	46,741	25	49,083	24

Table 5.1: Illustration Buffer creation split to stages *Fill, Sort, FindNeighbors, Compose.*

Biggest workload occurs during the *FindNeighbors* stage where we no longer see the spikes propagating to the stage time. This implies that application is not vertex bounded in most cases. Other properties has to be therefore examined. Table 5.2 shows measurements of the Illustration Buffer creation. Table is visualized in figure 5.5. Even though this figure is hard to read it implies some very interesting observations. For better visualization please refer to figure 5.4 where parallel coordinates are used. Since figure 5.4 does not provide the information about the current model, we provide both graphs for completeness.

First we can see that rendering time is highly affected by maximum length of all linked lists storing the information along the ray since shape of the lines is almost identical. When carefully examining views `GPU 2` and `Suspension 2` we see that even though `GPU 2` has lower number of fragments and vertices, smaller length of the longest linked list, the rendering time is higher than of `Suspension 2`. Reason for this behavior is captured in figure 5.3.

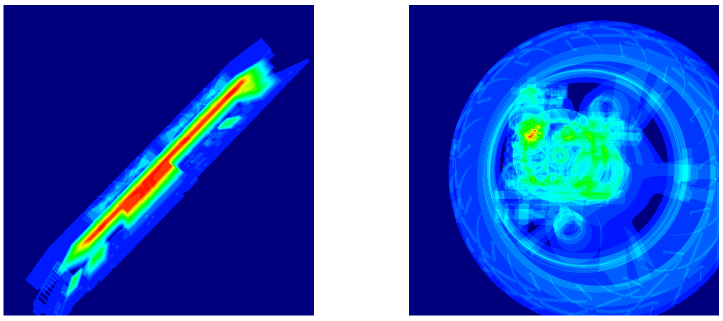


Figure 5.3: Heatmaps of the linked lists lengths of `GPU 2` and `Suspension 2`.

Is is clear that even though the longest linked list of `Suspension 2` is longer than in case of `GPU 2`, coverage of the longest linked lists is much lower. We can see this fact also in 5.5 by examining the dashed line. This line shows percentage coverage of the linked list lengths that are bigger than $\frac{2}{3}$ of the longest linked list, please mind the secondary vertical axis. We denote this coverage as Ψ. This also perfectly demonstrates the complexity of GPU parallelization process and optimizations being done by the hardware. Figure 5.4 shows how important is the Ψ property for the final rendering time, which is not so apparent from figure 5.5. Lines are in most cases ordered and without crossings in the last segment of the graph thus Ψ characteristics is the most important parameter affecting the final speed along with the maximum list lengths.

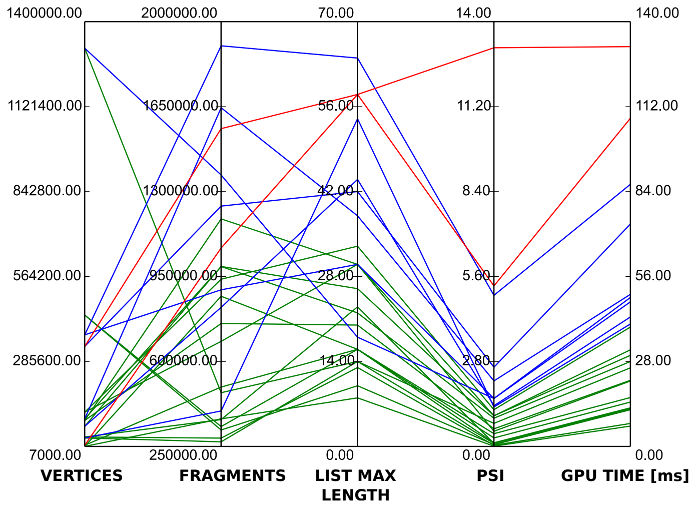


Figure 5.4: Parallel coordinates visualizing the same information as the figure 5.5. Time space is split to thirds. Results of the first third are green, second third blue and last third results are red.

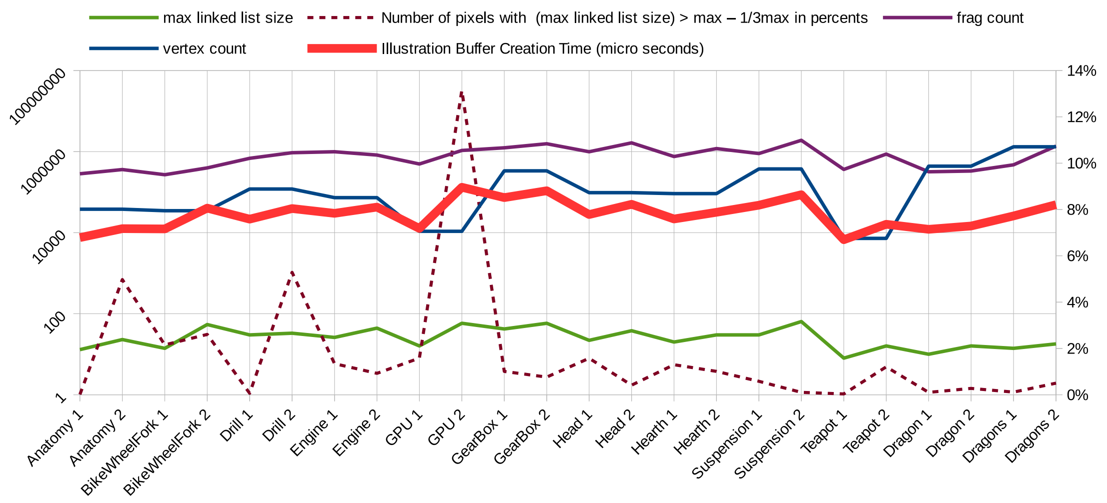


Figure 5.5: Visualized table 5.2. Secondary Y axis is used for the dashed line representing Ψ, primary Y axis (on the right) is then used for all other variables using logarithmic scale.

MODEL	VERTICES	FRAGMENTS	MAX |LIST|	Ψ [%]	TIME [ms]	FPS
Anatomy 1	37963	284196	13	0,01%	7,562848	126
Anatomy 2	37963	360567	23	4,98%	12,571584	80
BikeWheelFork 1	34786	268946	14	2,16%	12,390464	82
BikeWheelFork 2	34786	396110	54	2,61%	40,350754	26
Drill 1	119328	682863	30	0,06%	21,550688	47
Drill 2	119328	938931	33	5,29%	39,334015	28
Engine 1	73125	991435	26	1,34%	30,020128	35
Engine 2	73125	822796	44	0,93%	42,659233	26
GPU 1	10796	494564	16	1,58%	12,710432	77
GPU 2	10796	1067557	58	13,14%	131,768768	11
GearBox 1	334954	1240161	42	1,01%	73,290848	17
GearBox 2	334954	1559269	58	0,76%	108,144798	12
Head 1	97141	991103	22	1,58%	27,980576	38
Head 2	97141	1645353	38	0,42%	50,11261	22
Hearth 1	92026	756464	20	1,30%	21,785376	47
Hearth 2	92026	1187402	30	1,01%	31,837536	34
Suspension 1	372827	895210	30	0,58%	47,640865	24
Suspension 2	372827	1900548	64	0,10%	86,380898	15
Teapot 1	7231	363598	8	0,04%	6,69888	147
Teapot 2	7231	868698	16	1,21%	16,111521	62
Dragon 1	437645	317648	10	0,10%	12,100896	83
Dragon 2	437645	331854	16	0,28%	14,615904	68
Dragons 1	1312935	471328	14	0,12%	25,893728	41
Dragons 2	1312935	1368308	18	0,50%	49,082592	24

Table 5.2: Illustration Buffer creation - Table show used models, number vertices, fragments and max size of the linked lists in given view, resulting in time measured on GPU, and FPS where CPU overhead is considered as well. Column Ψ shows percentage coverage of the worst third of the linked lists lengths. In other words it shows coverage of the yellow and red pixels in the heatmaps in percents (Pixels with no linked lists stored are not considered).

Another method measurements can be found in Appendix B, where change of resolution is examined with other parameters fixed. Measurements, where one variable is changed with others fixed however can't be simply done with the coverage Ψ and linked list lengths. This would require special models designed for this very purpose thus such measurements were not conducted. Please note that by changing the resolution maximal length of linked lists does not need to stay fixed due to the rasterization process. It will however stay almost stable in most cases.

We can observe in figure B.1 that difference between GPU times between consequent resolutions depends on the same aspects of the scene as discussed in this section. That is percentage coverage Ψ, maximal length of all linked lists and finally number of fragments.

5.1.1 Sorting Methods Comparison

Here measurements of the two implemented sorting methods are presented. Since *per pixel linked lists* are part of the Illustration Buffer solution we present results on the *per pixel linked lists* only for simplicity.

It is very clear from graph in figure 5.6 and table 5.3 that dynamic version of the sort is winning in all cases over sorting in static array. This measurements might be however more interesting and producing different results even when GPU with bigger memory was used. We have used 3 arrays of size 64. One for IDs, second for the depths and third for the distances between layers. In case of our GPU it was more expensive to allocate such arrays than much bigger amount of texture reads and writes, which could differ on hardware where invocation of the fragment shader for one pixel would have more memory available.

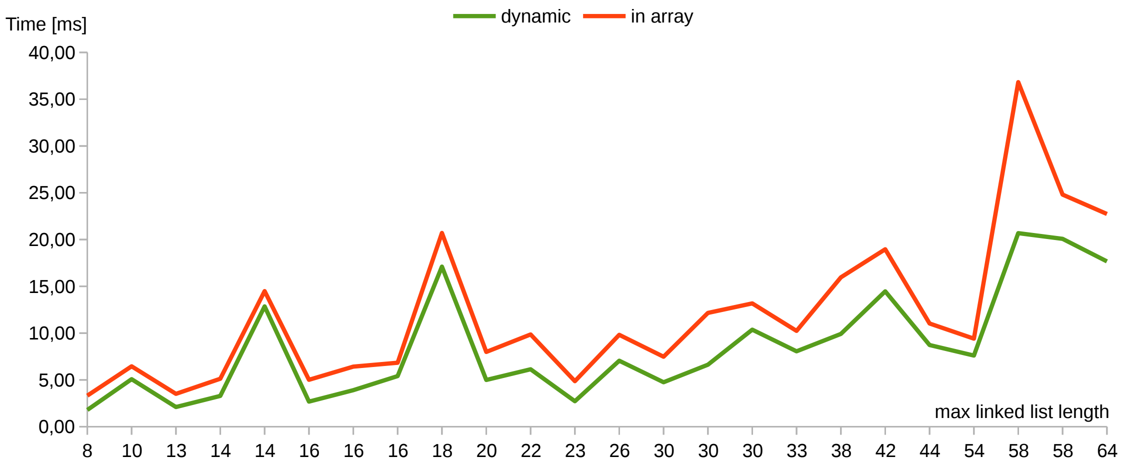


Figure 5.6: Speed comparison of the dynamic sort and sorting in static array. Bigger spikes are caused by the Ψ parameter as discussed before.

	Common characteristics				**Dynamic**		**In array**	
MODEL	**VERTICES**	**FRAGS**	**MAX |LIST|**	**Ψ [%]**	**TIME [ms]**	**FPS**	**TIME [ms]**	**FPS**
Teapot 1	7231	363598	8	0,04%	1,79	466	3,31	287
Dragon 1	437645	317648	10	0,10%	5,07	187	6,45	149
Anatomy 1	37963	284196	13	0,01%	2,10	418	3,51	262
BikeWheelFork 1	34786	268946	14	2,16%	3,29	285	5,12	187
Dragons 1	1312935	471328	14	0,12%	12,85	79	14,49	69
GPU 1	10796	494564	16	1,58%	2,68	343	5,01	189
Teapot 2	7231	868698	16	1,21%	3,90	241	6,42	150
Dragon 2	437645	331854	16	0,28%	5,41	176	6,83	141
Dragons 2	1312935	1368308	18	0,50%	17,11	60	20,70	50
Hearth 1	92026	756464	20	1,30%	4,99	189	7,99	121
Head 1	97141	991103	22	1,58%	6,14	156	9,85	101
Anatomy 2	37963	360567	23	4,98%	2,72	336	4,86	196
Engine 1	73125	991435	26	1,34%	7,05	135	9,82	101
Drill 1	119328	682863	30	0,06%	4,75	197	7,47	129
Hearth 2	92026	1187402	30	1,01%	6,62	145	12,16	81
Suspension 1	372827	895210	30	0,58%	10,37	94	13,18	77
Drill 2	119328	938931	33	5,29%	8,05	119	10,24	96
Head 2	97141	1645353	38	0,42%	9,91	98	15,96	64
GearBox 1	334954	1240161	42	1,01%	14,46	67	18,96	54
Engine 2	73125	822796	44	0,93%	8,73	111	11,02	90
BikeWheelFork 2	34786	396110	54	2,61%	7,60	115	9,40	99
GPU 2	10796	1067557	58	13,14%	20,68	50	36,82	30
GearBox 2	334954	1559269	58	0,76%	20,07	50	24,80	43
Suspension 2	372827	1900548	64	0,10%	17,66	57	22,74	44

Table 5.3: Comparison of the dynamic sort method with sort in the array. Table is sorted by the `MAX |LIST|` for convenience.

5.1.2 Comparison with Other Methods

We now know how the Illustration Buffer behaves. This section provides speed comparison with the remaining methods compared in this thesis: *depth peeling*, *dual depth peeling* and *concurrent per pixel linked lists*.

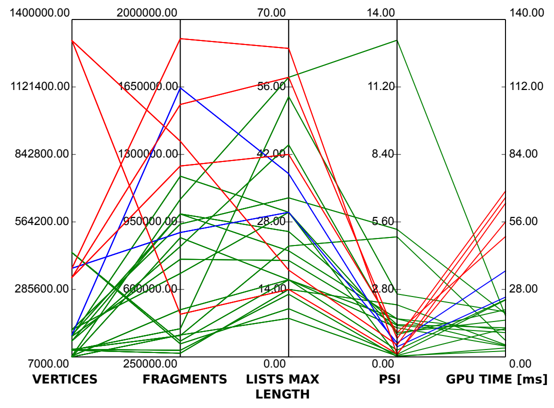


Figure 5.7: Depth peeling

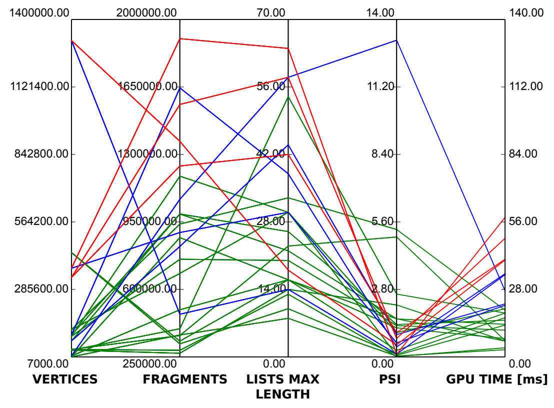


Figure 5.8: Dual depth peeling

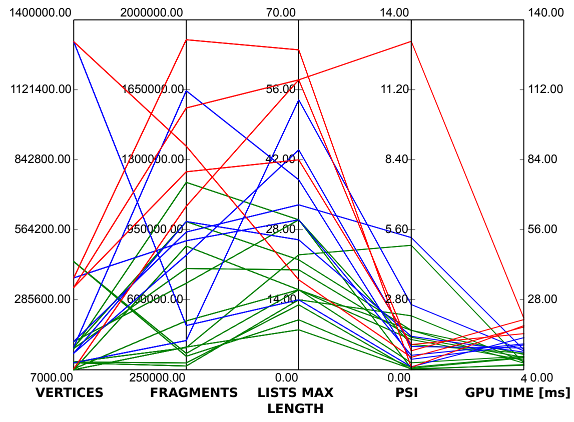


Figure 5.9: Concurrent per pixel linked lists

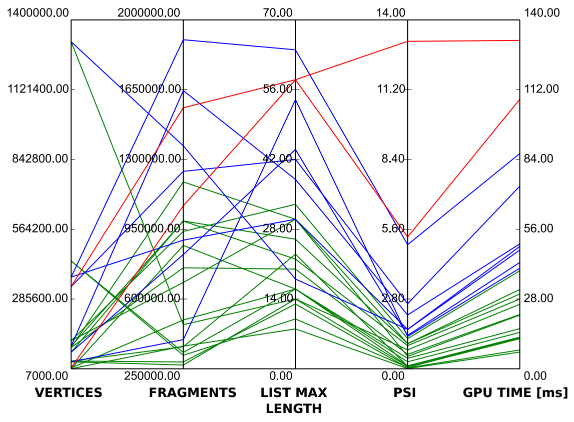


Figure 5.10: Illustration Buffer

We have used implementation of the peeling methods from the NVIDIA Graphics SDK 10, only our own GPU time measuring system was added to their implementation. Concurrent per pixel linked lists we on the other hand measured in our own implementation since it is a sub-problem of the Illustration Buffer construction.

Figures 5.7, 5.8, 5.9 and 5.10 show such comparison using parallel coordinates. Measured data can be also found in table 5.4. We can see that *linked lists* absolutely win in speed. Another observation is that only *Illustration Buffer* is really affected by Ψ, which is caused by its *FindNeighbors* stage. We can also see that overhead on the fragment shader is not big for first three methods and they are vertex bounded in most cases. This is logical for the *peeling methods* since we need to render the geometry in each peeling pass. It might be however surprising for the *per pixel linked lists*, where even though the sorting procedure must occur on all fragments overhead is small thus application stays vertex bounded. This is however completely different in case of the *Illustration Buffer* where processing of the *FindNeighbors* stage is vital for the final rendering time.

Another really interesting finding is that *dual depth peeling* is in fact much slower than original *depth peeling* algorithm. It is not an error in measurements, *dual depth peeling* as stated by the authors[9] 'may speed up performance by 2x for geometry bound applications'. Which is true, only for models `Dragon,Dragons` dual depth peeling actually speeds up the rendering process and slows down otherwise.

	Depth peeling		**Dual depth peeling**		**Linked lists**		**Illustration Buffer**	
MODEL	**TIME [ms]**	**FPS**	**TIME [ms]**	**FPS**	**TIME [ms]**	**FPS**	**TIME [ms]**	**FPS**
Anatomy 1	3,71456	269	3,780544	265	2,10	418	7,562848	126
Anatomy 2	6,845984	146	6,763392	148	2,72	336	12,571584	80
BikeWheelFork 1	5,002528	200	6,605408	151	3,29	285	12,390464	82
BikeWheelFork 2	18,693695	54	18,202721	55	7,60	115	40,350754	26
Drill 1	11,776256	85	13,186496	76	4,75	197	21,550688	47
Drill 2	17,932705	56	19,386368	52	8,05	119	39,334015	28
Engine 1	12,098816	83	15,221184	66	7,05	135	30,020128	35
Engine 2	19,601856	51	21,992928	46	8,73	111	42,659233	26
GPU 1	4,749152	211	6,540608	153	2,68	343	12,710432	77
GPU 2	17,111233	58	27,447712	36	20,68	50	131,768768	11
GearBox 1	49,930943	20	40,493729	25	14,46	67	73,290848	17
GearBox 2	68,777283	15	49,199966	20	20,07	50	108,144798	12
Head 1	11,10448	90	13,515168	74	6,14	156	27,980576	38
Head 2	24,807137	40	34,500385	29	9,91	98	50,11261	22
Hearth 1	10,079136	99	11,115456	90	4,99	189	21,785376	47
Hearth 2	15,244064	66	19,986464	50	6,62	145	31,837536	34
Suspension 1	35,680351	28	21,393408	47	10,37	94	47,640865	24
Suspension 2	63,731583	16	57,904831	17	17,66	57	86,380898	15
Teapot 1	2,470528	405	2,968192	337	1,79	466	6,69888	147
Teapot 2	4,463328	224	7,612128	131	3,90	241	16,111521	62
Dragon 1	23,64992	42	16,269793	62	5,07	187	12,100896	83
Dragon 2	23,782944	42	18,152449	55	5,41	176	14,615904	68
Dragons 1	56,015232	18	34,098175	29	12,85	79	25,893728	41
Dragons 2	66,103233	15	40,214657	25	17,11	60	49,082592	24

Table 5.4: Comparison of the *depth peeling, dual depth peeling* and *concurrent per pixel linked lists.* Please note that data `vertices, fragments, list max size` and Ψ are shared by all methods and can be found for example in table 5.2.

5.1.3 Memory consumption

Unfortunately, memory consumption is the biggest weakness of the Illustration Buffer. While in Depth Peeling and Dual Depth Peeling structures are of fixed size without any relation to the number of rendered fragments (except for the absolute size of the viewport, of course), structures *fragData* and *fragNext* of the Illustration Buffer are growing linearly based on the number of fragments. Demand on the memory is of course growing when we expand the amount of data stored per one fragment.

5.2 Results of the Opacity Modulation

In this section performance of methods used to modulate the opacity is presented as well as the results of such renders.

5.2.1 Speed Comparison

GPU time is captured in table 5.5 and visualized in figure 5.11. We show the GPU time of one pass only! It usually takes several passes to get pleasant results as discussed further. Please note that we omit other dependencies in figure 5.11 for simplicity, please refer to figure 5.10 where parallel coordinates are used to visualize such relations.

Diffusion and blurring stages are examined first. We see that curves of both are very much alike in shape but diffusion process is much more expensive. While both processes are similar, the implementation differs greatly. While blurring Gaussian filter is separable to horizontal and vertical pass, Laplacian filter used in diffusion process cannot be separated thus it needs to access more texture data in one iteration. It is also clear that performance cost of the blurring process is relatively small thus we can use it to enhance the diffusion process.

While it may seem that process of finding the surface curvature along with the distance between samples is more expensive than previously discussed stages, it is not. We need several passes of both diffusion and blurring processes while the distance-curvature pass will be performed only once to get data we need for final composition. 8 neighbors were used to find the surface curvature in this measurement.

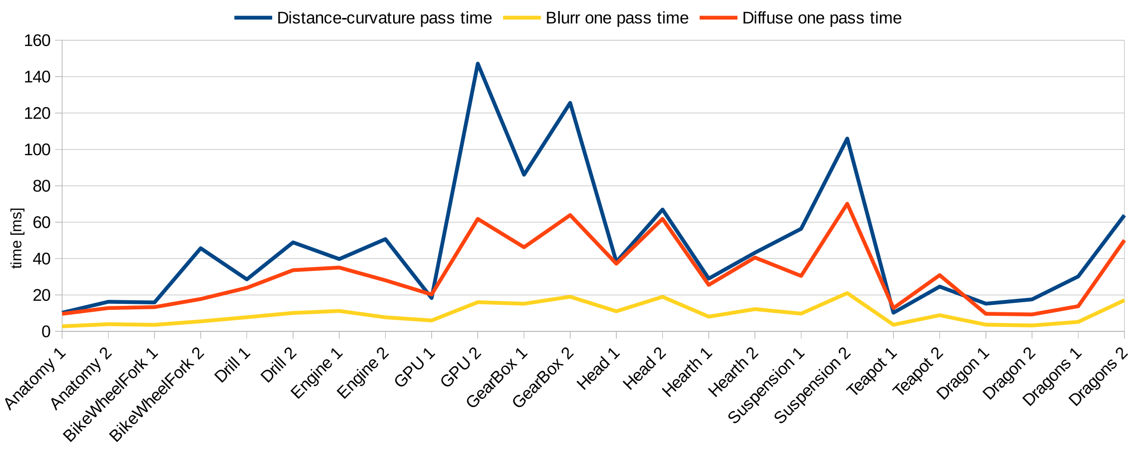


Figure 5.11: Speed comparison of the single pass of the diffusion process, blurring by Gaussian separable filter and of distance-curvature search, where distances between layers are found as well as the surface curvature.

However as we show in visual comparison, curvature computed from 4 geodesic neighbors is usually sufficient, which will yield lower processing time.

Only measurements of one pass are provided. This is simply because we would get linear progression of the time for many iterations of presented methods. Such progression is not really interesting nor surprising since dataset topology (neighbors) does not change during the process.

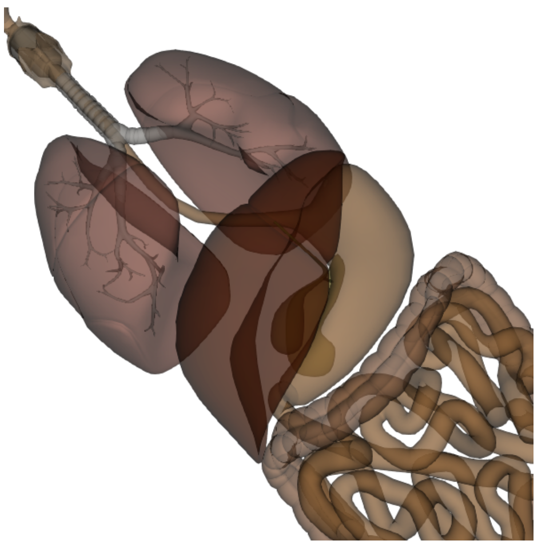
	Common characteristics				**DIFFUSION**	**BLURR**	**DIST-CURV**
MODEL	**VERTICES**	**FRAGS**	**MAX |LIST|**	**Ψ [%]**	**TIME [ms]**	**TIME [ms]**	**TIME [ms]**
Anatomy 1	37963	284196	13	0,01%	9,630	2,771	10,244
Anatomy 2	37963	360567	23	4,98%	12,809	3,933	16,257
BikeWheelFork 1	34786	268946	14	2,16%	13,325	3,581	15,895
BikeWheelFork 2	34786	396110	54	2,61%	17,775	5,484	45,671
Drill 1	119328	682863	30	0,06%	23,919	7,745	28,554
Drill 2	119328	938931	33	5,29%	33,609	10,109	48,878
Engine 1	73125	991435	26	1,34%	35,054	11,194	39,724
Engine 2	73125	822796	44	0,93%	28,032	7,693	50,682
GPU 1	10796	494564	16	1,58%	20,245	5,981	18,279
GPU 2	10796	1067557	58	13,14%	61,811	16,001	147,159
GearBox 1	334954	1240161	42	1,01%	46,219	15,173	86,068
GearBox 2	334954	1559269	58	0,76%	63,914	19,020	125,594
Head 1	97141	991103	22	1,58%	37,152	11,019	38,196
Head 2	97141	1645353	38	0,42%	61,848	18,971	66,959
Hearth 1	92026	756464	20	1,30%	25,532	8,062	28,990
Hearth 2	92026	1187402	30	1,01%	40,554	12,199	43,196
Suspension 1	372827	895210	30	0,58%	30,447	9,739	56,365
Suspension 2	372827	1900548	64	0,10%	70,155	20,998	105,973
Teapot 1	7231	363640	8	0,04%	12,777	3,618	10,228
Teapot 2	7231	868698	16	1,21%	30,894	8,861	24,599
Dragon 1	437645	317648	10	0,10%	9,624	3,669	15,210
Dragon 2	437645	331854	16	0,28%	9,303	3,240	17,553
Dragons 1	1312935	471328	14	0,12%	13,778	5,192	30,169
Dragons 2	1312935	1368308	18	0,50%	50,068	17,090	63,829

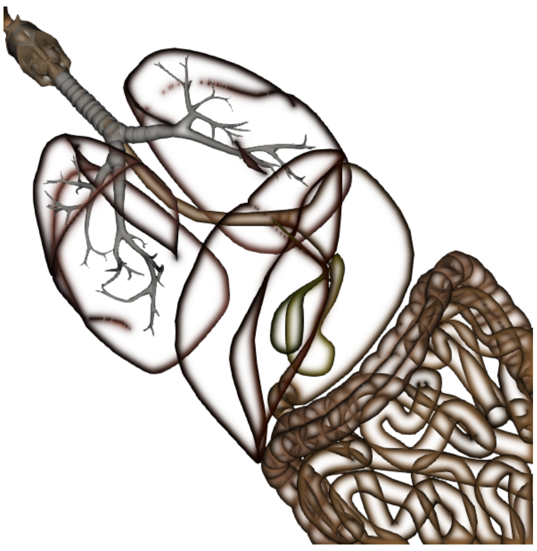
Table 5.5: Speed comparison of the single pass of the diffusion process, blurring by Gaussian separable filter and of distance-curvature search, where distances between layers are found as well as the surface curvature.

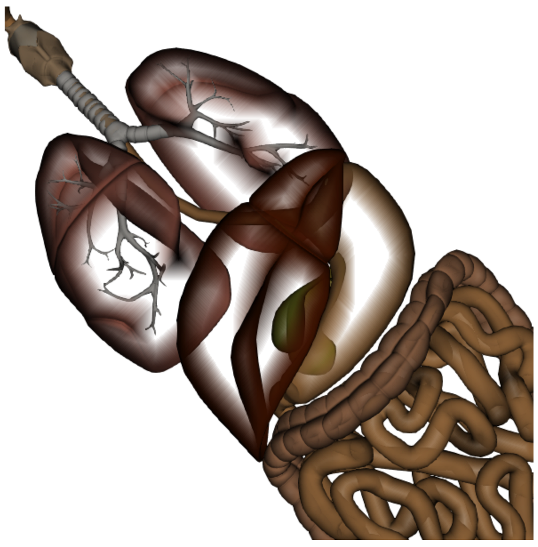
5.2.2 Visual Comparison

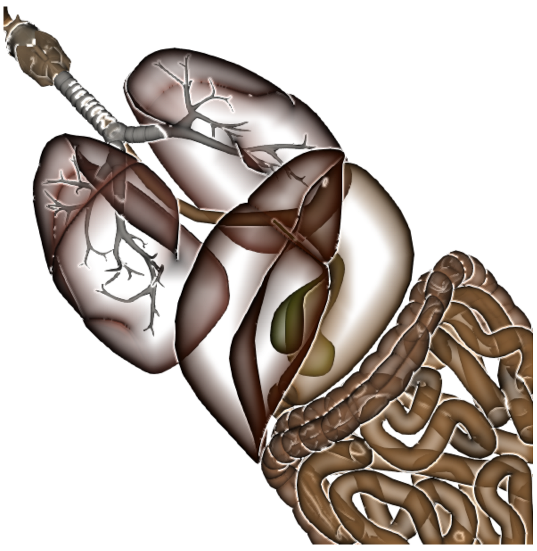
Selection of output images rendered by our application is presented in this section. We also present FPS in time of rendering and the values of variables used. Since this thesis does not address the human perception in greater detail and focus on the technical aspects, we leave the conclusion of this images to the reader. We provide this list to remind the parameters used in this thesis and algorithms to better understand following images:

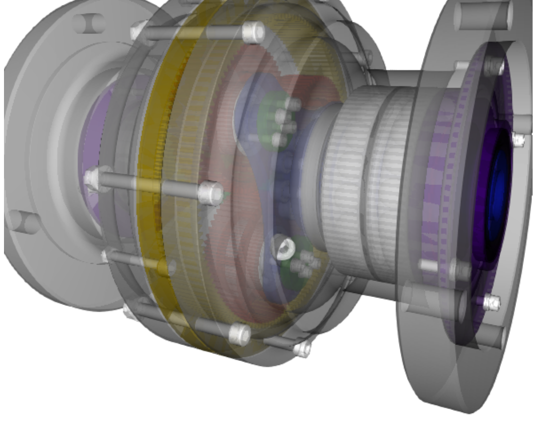
- *Curv* is a short for curvature.
- *Dist* is a short for method based on distance between samples along the ray.
- δ_c is a user parameter defined to affect the curvature effect on the final image.
- α, β, γ are transparency fields defined in Chapter 3.
- $\lambda_\alpha, \lambda_\beta, \lambda_\gamma$ are parameters to affect transparency fields α, β, γ. If Caption of an image does not state such value, automatic mode was applied as described in Chapter 4.
- *di* is a number of iterations used during the diffusion process.
- *bi* is a number of iterations of the Gaussian filter.
- $focusRegion$ is a distance between samples along the ray of higher importance.

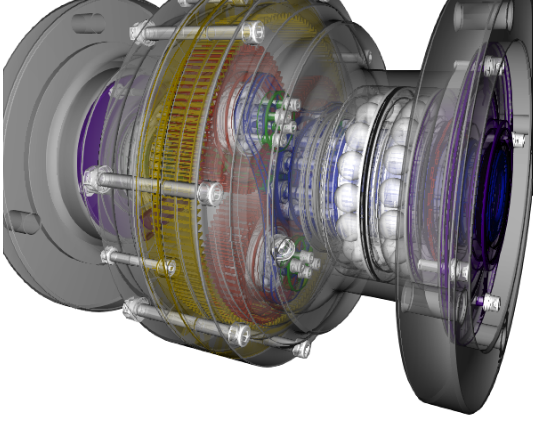

OIT Only, $\alpha = 0.5$, 269 FPS

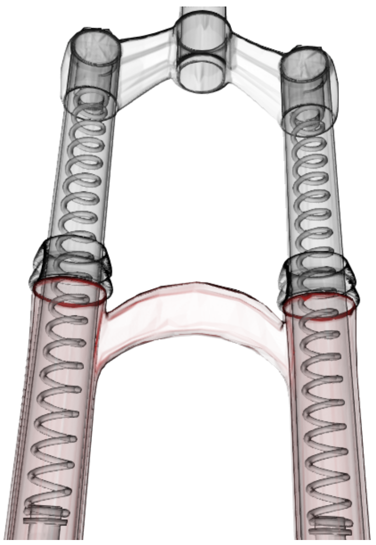

used α, $di = 25$ 12 FPS


used α, β, $di = 25$ 12 FPS

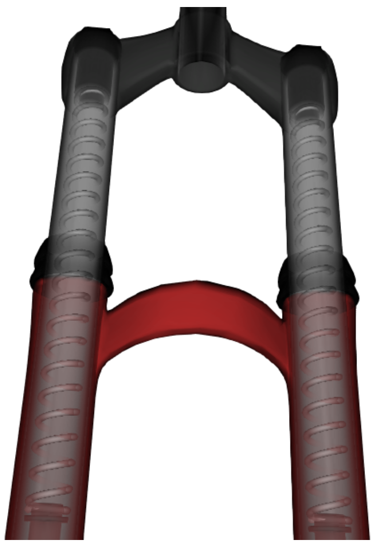

used α, β, γ, $di = 25$, $bi = 10$, 9 FPS


OIT Only, $\alpha = 0.5$, 34 FPS

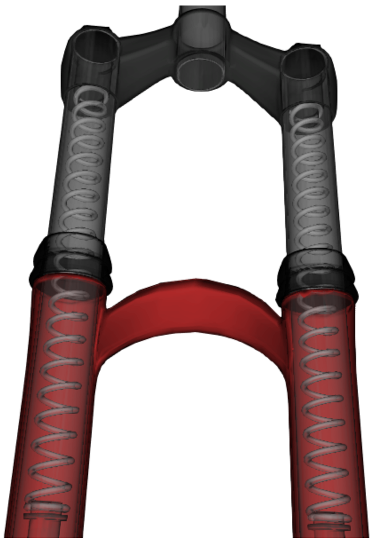

Curv-Dist, $focusRegion = 0.7$,$\delta_c = 1.8$, 9 FPS



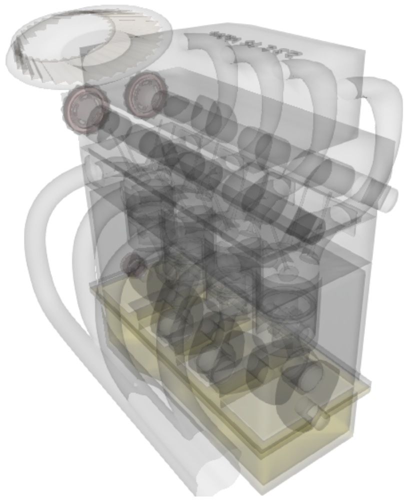
Curv Only, $\delta_c = 0.6$, 46 FPS



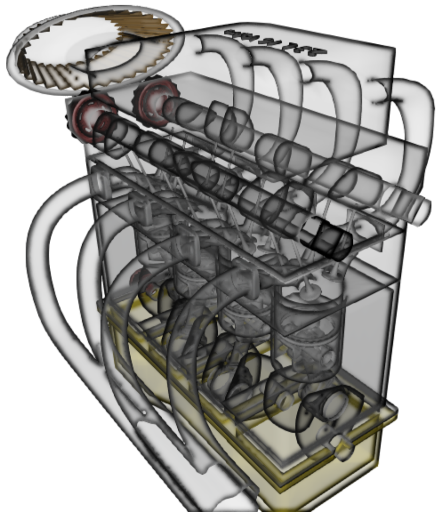
Dist Only, $focusRegion = 0.1$, 46 FPS



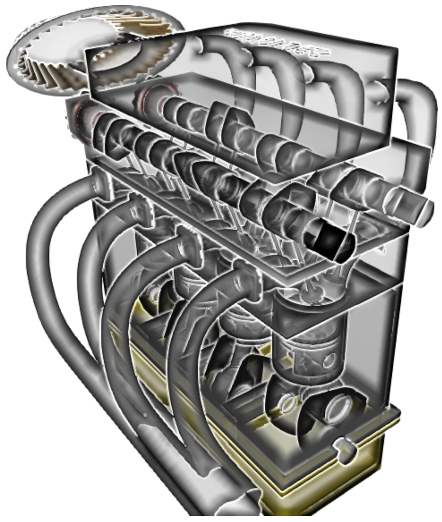
Dist-Curv, $focusRegion = 1.1$, $\delta_c = 0.9$, 46 FPS



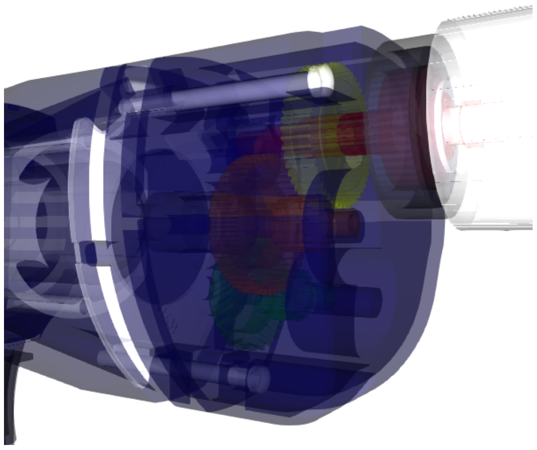
Curv Only, $\delta_c = 0.6$, 46 FPS

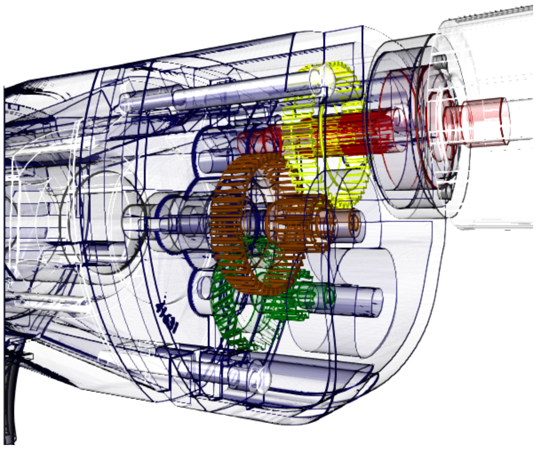


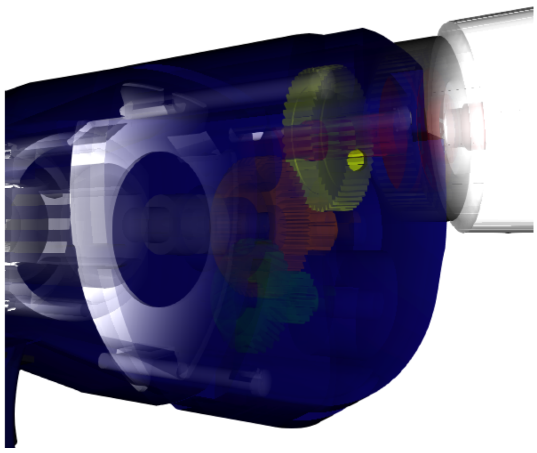
OIT Only, $\alpha = 0.13$, 11 FPS

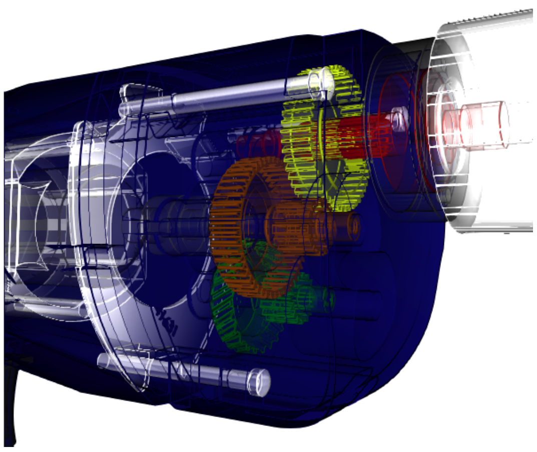


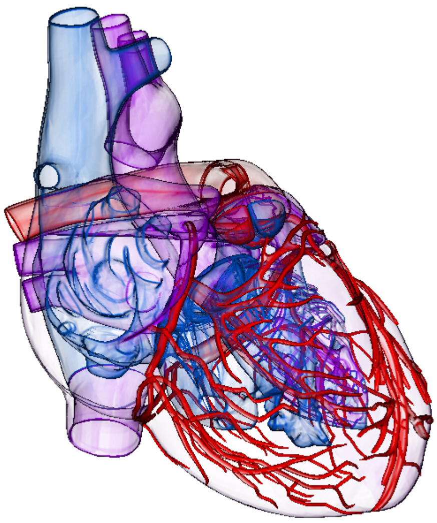
used α, β, γ, $di = 10$, $bi = 2$,$\alpha = 0.13$, 9 FPS

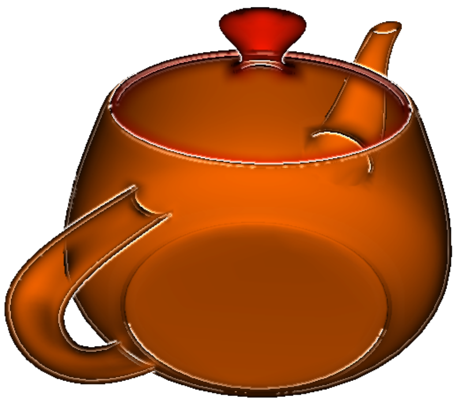

OIT Only, $\alpha = 0.34$, 69 FPS

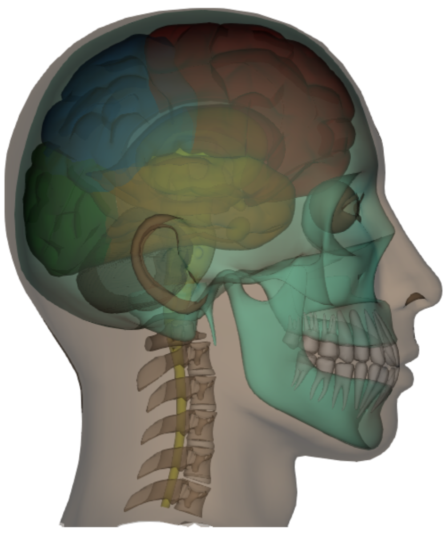

Curv Only, $\delta_c = 1.8$, 23 FPS

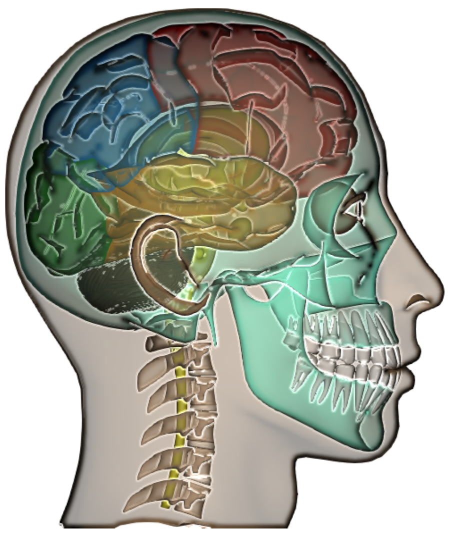

Dist Only, $focusRegion = 0.9$, 23 FPS

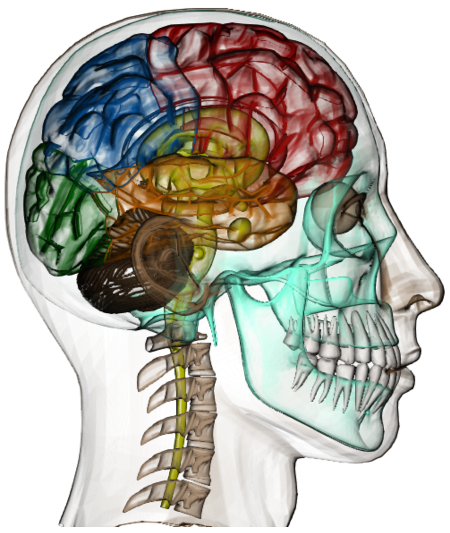

Dist-Curv, $focusRegion = 0.9$, $\delta_c = 1.8$, 23 FPS

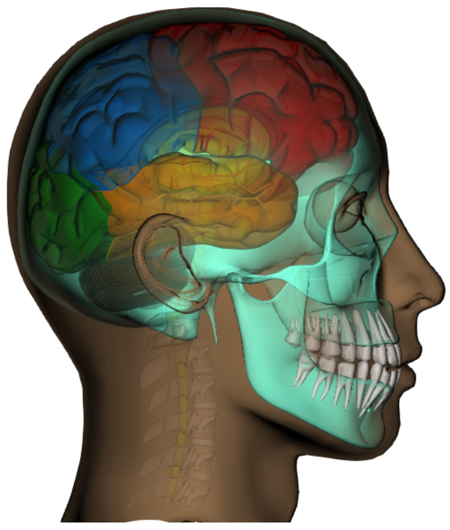

Curv Only, $\delta_c = 1.2$, 21 FPS


used α, β, γ, $di = 10$, $bi = 5$,$\alpha = 0.13$, 18 FPS


OIT Only, $\alpha = 0.4$, 79 FPS


used α, β, γ, $di = 5$, $bi = 5$, $\alpha = 0.16$, 9 FPS


Curv Only, $\delta_c = 1.8$, 24 FPS


Dist-Curv, $focusRegion = 0.5$, $\delta_c = 1.8$, 24 FPS

Chapter 6

Conclusion

This thesis addressed the approaches of the internal structure visualization for complex 3D objects. Theoretical background for the problematics of rendering semi transparent objects and techniques of the opacity modulation based on several object features was presented. Comparison of the OIT solving algorithms was provided and the Illustration Buffer was found the most flexible algorithm considering discussed opacity modulation techniques thus was examined in further detail.

We implemented the Illustration Buffer algorithm using OpenGL and GLSL technology. All discussed opacity modulation technique were implemented on top of the Illustration buffer. We have found the construction of the buffer to be the hard part of the implementation, application of the opacity modulation methods was rather easy.

We have tested our implementation on 24 scenes consisting of 12 geometrical models varying between 7K to 1.3M vertices, 26K to 1.9M fragments and 8 to 64 maximal lengths of the per pixel linked lists using 600x600 image resolution. Impact of increasing the number of fragments was measured by changing the resolution.

Our results show that the *Concurrent per Pixel Linked Lists* are the best choice over other methods considering speed. *Peeling methods* are on the other hand more memory efficient and their memory consumption does not grow with the number of fragments like in case of the *Illustration Buffer* and *Concurrent per Pixel Linked Lists.* The Illustration Buffer was found to be the slowest but most usable algorithm. Now it should be easy for the reader to choose the algorithm according to his/her demands.

Construction times of the Illustration Buffer vary between 6ms to 131ms in our measurements. We have found the number of fragments and most importantly the maximum length of the per pixel linked lists and coverage of the longest lists to be the most performance affecting variables.

We have introduced geometry motivated method for searching the geodesic neighbors along the surface which speeds up the search process but produces artifacts for models with great detail (so that several triangles are render to one pixel.). This can be however solved by model preprocessing in exchange for greater performance and interactivity.

6.1 Future Work

We have seen that search for the neighbors is a bottleneck of the Illustration Buffer creation process. Methods of this process optimization and enhancement should be researched to provide better interactivity even on very complex models. As we could see in section *Visual Comparison* where final renders of our application were shown, our implementation is not *Z-Fighting* aware and therefore produces errors where the geometry is poorly defined or is defined in scale that is too small.

Mainly technical aspects of the opacity modulation were addressed in this thesis and testing on human participants should be done to better understand the human perception of complex internal structures by using varying opacity modulation techniques.

Bibliography

[1] GLEW - The OpenGL Extension Wrangler Library. <http://glew.sourceforge.net/>. Accessed: 2014-12-24.

[2] GLM - OpenGL Mathematics. <http://glm.g-truc.net/0.9.6/index.html>. Accessed: 2014-12-24.

[3] GLSL - OpenGL Shading Language. <https://www.opengl.org/documentation/glsl/>. Accessed: 2014-12-24.

[4] OpenGL - The Industry's Foundation for High Performance Graphics. <https://www.opengl.org/>. Accessed: 2014-12-24.

[5] QT Framework - Qt Project. <http://qt-project.org/>. Accessed: 2014-12-24.

[6] OpenGL - Image Load Store. <https://www.opengl.org/wiki/Image_Load_Store>. Accessed: 2014-12-24.

[7] Rapid JSON - A fast JSON parser/generator for C++ with both SAX/DOM style API. <https://github.com/miloyip/rapidjson>. Accessed: 2014-12-31.

[8] NVIDIA Graphics SDK 10. <https://developer.nvidia.com/opengl>. Accessed: 2014-12-24.

[9] BAVOIL, L. – MYERS, K. Order independent transparency with dual depth peeling. Technical report, NVIDIA Corporation, 02 2008.

[10] BAVOIL, L. et al. Multi-fragment effects on the GPU using the k-buffer. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, s. 97–104. ACM, 2007.

[11] BRUCKNER, S. et al. Illustrative context-preserving volume rendering. In *EuroVis*, s. 69–76, 2005.

[12] CARNECKY, R. et al. Smart transparency for illustrative visualization of complex flow surfaces. *IEEE Trans. Vis. Comput. Graph.* 2013, 19, 5, s. 838–851.

[13] CARPENTER, L. The A -buffer, an Antialiased Hidden Surface Method. *SIGGRAPH Comput. Graph.* January 1984, 18, 3, s. 103–108. ISSN 0097-8930. doi: 10.1145/964965.808585. Dostupné z: <http://doi.acm.org/10.1145/964965.808585>.

[14] MOURA PINTO, F. – FREITAS, C. M. Importance-aware composition for illustrative volume rendering. In *Graphics, Patterns and Images (SIBGRAPI), 2010 23rd SIBGRAPI Conference on*, s. 134–141. IEEE, 2010.

[15] EVERITT, C. Interactive order-independent transparency. Technical report, NVIDIA Corporation, 06 2001.

[16] KRUGER, J. – SCHNEIDER, J. – WESTERMANN, R. Clearview: An interactive context preserving hotspot visualization technique. *Visualization and Computer Graphics, IEEE Transactions on.* 2006, 12, 5, s. 941–948.

[17] MESHKIN, H. Sort-independent alpha blending. 2007.

[18] MYERS, K. – BAVOIL, L. Stencil routed A-buffer. In *ACM SIGGRAPH*, 7, 2007.

[19] NGUYEN, H. *Gpu Gems 3.* Addison-Wesley Professional, first edition, 2007. ISBN 9780321545428.

[20] NIENHAUS, M. – DöLLNER, J. Blueprints - Illustrating Architecture and Technical Parts using Hardware-Accelerated Non-Photorealistic Rendering. In HEIDRICH, W. – BALAKRISHNAN, R. (Ed.) *Graphics Interface*, 62 / *ACM International Conference Proceeding Series*, s. 49–56. Canadian Human-Computer Communications Society, 2004. Dostupné z: <`http://dblp.uni-trier.de/db/conf/graphicsinterface/graphicsinterface2004.html"`>. ISBN 1-56881-227-2.

[21] RONG, G. – TAN, T.-S. Jump flooding in GPU with applications to Voronoi diagram and distance transform. In *Proceedings of the Symposium on Interactive 3D Graphics and Games*, s. 109–116. ACM Press, 2006.

[22] VASILAKIS, A. – FUDOS, I. Z-fighting Aware Depth Peeling. In *SIGGRAPH Posters.* ACM, 2011.

[23] YANG, J. C. et al. Real-time Concurrent Linked List Construction on the GPU. In *Proceedings of the 21st Eurographics Conference on Rendering*, EGSR'10, s. 1297–1304, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association. doi: 10.1111/j.1467-8659.2010.01725.x. Dostupné z: <`http://dx.doi.org/10.1111/j.1467-8659.2010.01725.x`>.

[24] ŽÁRA, J. et al. *Moderní počítačová grafika.* Computer Press, 2004. ISBN 9788025104545.

[25] ČMOLíK, L. *Interactive Illustrative Visualization of 3d Models.* PhD thesis, Faculty of Electrical Engineering, Czech Technical University in Prague, July 2011.

Appendix A

List of Abbreviations

1D	One-Dimensional
2D	Two-Dimensional
3D	Three-Dimensional
ANSI	American National Standards Institute
API	Application Programming Interface
CPU	Central Processing Unit
FPS	Frames per second
GDC	Game Development Conference
GLEW	The OpenGL Extension Wrangler Library
GLSL	OpenGL Shading Language
GPU	Graphics Processing Unit
GUI	Graphical User Interface
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
MVP	Model View Projection
OBJ	Wavefront .obj File Format
OIT	Order Independent Transparency
QT	Cross-platform Application and UI Framework
RGBA	Red Green Blue Alpha Format
SDK	Software Development Kit
UI	User Interface
VBO	Vertex Buffer Object

Appendix B

Additional Measurements Data

	400x400		450x450		500x500		550x550	
model	#frags	[ms]	#frags	[ms]	#frags	[ms]	#frags	[ms]
Anatomy 1	126296	4,71	159666	5,09	197269	5,52	238644	6,71
Anatomy 2	160267	7,81	202807	8,44	250522	9,75	302912	11,51
BikeWheelFork 1	119486	6,64	151306	7,97	186828	8,54	226038	11,16
BikeWheelFork 2	176012	25,14	222792	28,07	275204	29,03	332948	30,77
Drill 1	303673	12,42	384597	14,61	474041	16,45	573959	20,03
Drill 2	417460	21,55	528164	25,73	651967	30,35	788422	34,68
Engine 1	440439	16,81	557607	19,71	688829	22,26	833030	26,58
Engine 2	365743	24,23	462718	28,11	571715	31,53	691365	38,55
GPU 1	219870	7,28	278235	8,68	343496	9,43	415536	11,26
GPU 2	474450	61,33	600549	81,68	741436	83,67	897034	102,54
GearBox 1	550972	43,57	697378	48,73	860962	55,39	1041905	66,89
GearBox 2	693358	61,88	877294	68,70	1083141	82,11	1310306	92,44
Head 1	440563	14,95	557542	17,73	688352	20,62	833145	24,71
Head 2	731184	26,97	925545	32,02	1142755	37,66	1382749	41,62
Hearth 1	336200	12,41	425523	14,70	525262	16,66	635657	19,73
Hearth 2	527587	17,28	667923	20,32	824387	23,35	997826	27,83
Suspension 1	397860	27,82	503386	32,00	621602	37,81	752308	42,39
Suspension 2	844522	52,52	1069506	61,71	1320028	64,70	1597364	68,17
Teapot 1	161566	3,37	204534	4,18	252578	4,51	305504	5,79
Teapot 2	386128	8,03	488538	10,03	603308	11,31	729882	13,80
Dragon 1	141126	8,64	178682	9,53	220612	10,05	266784	11,48
Dragon 2	147526	10,00	186578	11,10	230448	12,11	278944	13,47
Dragons 1	209316	19,92	264930	21,43	327276	22,33	395986	24,51
Dragons 2	608124	32,35	769554	35,96	950294	39,87	1149646	44,81

Table B.1: Impact of changing the number of fragments with other variables fixed. First row denotes used viewport resolution.

	600x600		650x650		700x700	
model	#frags	[ms]	#frags	[ms]	#frags	[ms]
Anatomy 1	284196	7,82	333476	8,71	386880	8,79
Anatomy 2	360567	13,04	423164	14,87	490751	16,85
BikeWheelFork 1	268946	12,60	315752	13,70	366012	14,69
BikeWheelFork 2	396110	42,24	464712	42,01	539200	46,34
Drill 1	682863	21,95	802118	24,96	929220	28,22
Drill 2	938931	40,02	1101728	44,11	1277522	49,58
Engine 1	991435	30,38	1163484	34,37	1349344	37,95
Engine 2	822796	43,10	965506	47,64	1119805	51,28
GPU 1	494564	13,00	580287	14,83	673177	16,44
GPU 2	1067557	132,28	1253030	132,23	1453378	148,61
GearBox 1	1240161	73,63	1455451	81,61	1687934	89,25
GearBox 2	1559269	102,70	1830104	107,29	2122659	108,57
Head 1	991103	28,28	1163565	32,60	1349534	35,01
Head 2	1645353	44,81	1931097	45,51	2239816	48,29
Hearth 1	756464	22,39	887865	25,17	1029590	28,07
Hearth 2	1187402	32,16	1393473	37,34	1616192	39,81
Suspension 1	895210	48,69	1050801	53,92	1218220	59,23
Suspension 2	1900548	75,55	2230808	80,17	2587586	87,69
Teapot 1	363598	7,27	426754	7,82	494974	8,59
Teapot 2	868698	16,76	1019654	19,14	1182474	21,16
Dragon 1	317648	12,34	372736	13,69	432214	14,15
Dragon 2	331854	15,19	389544	16,17	451678	17,24
Dragons 1	471328	26,28	552926	28,15	641244	29,50
Dragons 2	1368308	47,10	1606040	47,33	1862570	47,27

Table B.2: Impact of changing the number of fragments with other variables fixed. First row denotes used viewport resolution.

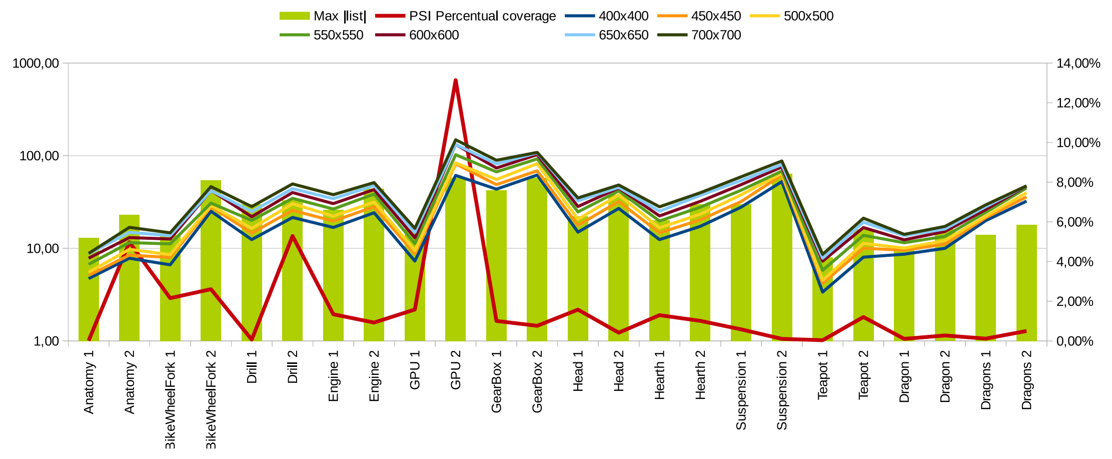


Figure B.1: Visualization of tables B.1 and B.2
. PSI is Ψ as defined before.

Appendix C

Installation Guide

Our application needs graphic card supporting at least OpenGL of version 4.2. It also requires extensions `GL_shader_image_load_store, GL_shader_atomic_counters`. The application is compilable on both Linux and Windows. Following minimal software requirements have to be met:

Library	Minimal version
OpenGL	4.2
GLEW	1.11.0
Qt	5.3
GLM	0.9.5
QMake	3.0
GCC	4.9.1
CLANG	3.5.0

Table C.1: Minimal application requirements. Either CLANG or GCC is required, not both.

Windows libraries 32 bit compiled by `MinGW` are included on the DVD. On linux simply install packages for your distribution including the library. Easiest way is to compile the code is using QtCreator IDE which automatically detects the environment and performs necessary commands. If using this method, please set the root directory of our structure as a run directory so that `resources` are available. We now describe how to build application manually:

1. Run qmake IllustrationBufferProjectQT5.pro to generate makefiles.
2. Run make
3. Run make install (this will automatically copy the resources folder to your executable.)
4. Run compiled application.

Entire project configuration for both Windows and Linux platforms is present in the `IllustrationBufferProjectQT5.pro` file which makes this project easily usable.

Appendix D

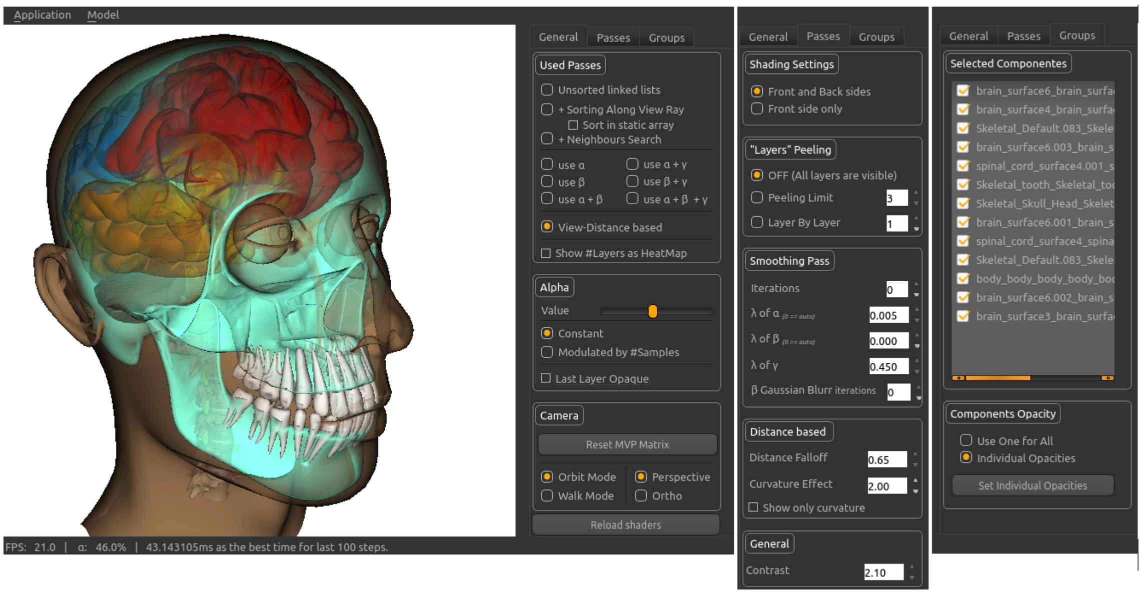
DVD Content

This is the structure of attached DVD. Please note that not all models presented as renders in this thesis are part of the `models` folder since some models provided by the supervisor were not allowed to distribute, only publish the results by it's license. Models that are included are free to use.

`DVD`	root directory
`bat`	scripts for measurements
`bin`	application windows executable
`platforms`	necessary QT libs for windows
`resources`	copy of the resources for win exe
`doc`	documentation, thesis PDF and video
`doxygen`	documentation of the source code
`latex`	source code of the thesis text
`screenshots`	screenshots of the application
`libs`	libraries
`resources`	non compilable resources
`config`	configuration JSON files
`gui`	styles of the GUI
`images`	Images for the GUI
`models`	models and materials
`shaders`	GLSL shaders
`textures`	texture for the heatmap
`src`	C++11 source code
`gui`	classes and ui files for the GUI
`measuring`	classes used for the measurements
`graphs`	python files to generate parallel coord. graphs
`tmp`	empty directory, used for temporal build data

Appendix E

User Manual



FPS: 21.0 | α: 46.0% | 43.143105ms as the best time for last 100 steps.

Used Passes
- [ ] Unsorted linked lists
- [ ] + Sorting Along View Ray
  - [ ] Sort in static array
- [ ] + Neighbours Search
- [ ] use α
- [ ] use β
- [ ] use α + β
- [ ] use α + γ
- [ ] use β + γ
- [ ] use α + β + γ
- [✓] View-Distance based
- [ ] Show #Layers as HeatMap

Alpha
- Value
- [✓] Constant
- [ ] Modulated by #Samples
- [ ] Last Layer Opaque

Camera
- [✓] Orbit Mode
- [ ] Walk Mode
- [✓] Perspective
- [ ] Ortho

Shading Settings
- [✓] Front and Back sides
- [ ] Front side only

"Layers" Peeling
- [✓] OFF (All layers are visible)
- [ ] Peeling Limit: 3
- [ ] Layer By Layer: 1

Smoothing Pass
- Iterations: 0
- λ of α (0 == auto): 0.005
- λ of β (0 == auto): 0.000
- λ of γ: 0.450
- β Gaussian Blurr iterations: 0

Distance based
- Distance Falloff: 0.65
- Curvature Effect: 2.00
- [ ] Show only curvature

General
- Contrast: 2.10

Selected Componentes
- [✓] brain_surface6_brain_surfa
- [✓] brain_surface4_brain_surfa
- [✓] Skeletal_Default.083_Skele
- [✓] brain_surface6.003_brain_s
- [✓] spinal_cord_surface4.001_s
- [✓] Skeletal_tooth_Skeletal_to
- [✓] Skeletal_Skull_Head_Skelet
- [✓] brain_surface6.001_brain_s
- [✓] spinal_cord_surface4_spina
- [✓] Skeletal_Default.083_Skele
- [✓] body_body_body_body_bo
- [✓] brain_surface6.002_brain_s
- [✓] brain_surface3_brain_surfa

Components Opacity
- [ ] Use One for All
- [✓] Individual Opacities

Figure E.1: Graphical User Interface of the main window with `Settings panel` on the right. All three tabs are shown.

Main GUI of our application can be seen in figure E.1. Settings panel on the right enables full control of active stages, camera modes and most importantly working with the variables that affect the result as described in this thesis. We also provide the *Contrast* modifier to simply enhance image colors by adjusting the contrast.

Main menu can be seen on the top bar. Application section provides actions *Export to PNG*, *About* and *Quit*. Please note that export of the framebuffer to a PNG file is supported only on Linux due to Qt bug. On windows image will be generated but colors will be wrong. Model section provides list of all models loaded to the application.

E.1 Controls

Application and algorithm's functions are controlled via the settings panel and global menu. Following controls are used for movement in the scene:

- **Orbit Camera Mode** - mouse is used to rotate the model when left mouse button is pressed. Scrolling wheel can be used to zoom.
- **Walk Camera Mode** - Keys `W,A,S,D` are used to move forward, backward, left and right. Keys `Q,E` are used for control of the elevation. Mouse can be used to adjust the *look at* direction.

E.2 Configuration

Application is configurable via JSON config files. Following snippet shows how the config file looks like for models. First is a name as will appear in the application and second is the path to a file. Material files are loaded automatically, its name has to be the same as of the `.obj` file though, only with `.mtl`.

```
{
    "Anatomy" : "resources/models/anatomy.obj",
    "Teapot"  : "resources/models/teapot.obj"
}
```

```
{
  "fillPass" : {
        "programNumber" : 1,
        "vert" : "resources/shaders/basic.vert",
        "frag" : "resources/shaders/fillPass.frag"
    },
    "sortPass": {
      "programNumber": 2,
        "vert" : "resources/shaders/sortPass.vert",
        "frag" : "resources/shaders/sortPass.frag"
    },
    "renderOITPass": {
        "programNumber": 3,
        "vert" : "resources/shaders/sortPass.vert",
        "frag" : "resources/shaders/renderOIT.frag"
    },
}
```

E.3 Measurements and Graphs

We also provide several `.bat` files that can be used for batch measurements of all provided models on the Windows platform for convenience.

Note that the application source code has to be recompiled with uncommented line 25: `#define MEASURING_TIME` in `Context.h` header file. For optimal results please check the comments in the shaders, where several defines are used for measurements of the given task only.

- **IllustrationBufferConstruction.bat** Illustration Buffer construction.
- **IllustrationBufferResolutions.bat** Changing the resolution from 400x400 to 700x700
- **IllustrationBufferDiffusion.bat** Measurement of one diffusion pass GPU time
- **IllustrationBufferDiffusionGauss.bat** Measurement of one blurring pass GPU time
- **LinkedLists.bat** Measures only creation of *per pixel linked lists*.
- **IllustrationBufferViewDistance.bat** Is used for measurements of finding the curvature and distance between samples.

These scripts runs the application with the same MVP matrices as we have used in our measurements and will generate txt file with the results.

When the application is compiled with `#define MEASURING_TIME` as stated above, following arguments can be used:

arg 1: Number for measured task (1 = per pixel linked lists, 2 = Illustration Buffer construction, 3 = Diffusion, 4 = Diffusion and Gaussian filter, 5 = Distance Curvature)
arg 2 : counter of the model (e.g. 2 if this model is measured second time in given view)
arg 3 : model name as will appear in generated report
arg 4 : model path
arg 5 - 21 : rows of the model view matrix
arg 22 - 37 : rows of the projection matrix
arg 38 : optional dimmension of the viewport. Is set 600x600 by default.

Python scripts we have used to generate graphs with parallel coordinates are also included on attached DVD in `src/graphs` directory.